SmartFuzz: Making Smart Contract Fuzzing Smarter via State-Aware Feedback

ANONYMOUS AUTHOR(S)*

With the rapid development of blockchain and smart contracts, many decentralized applications (DApps) have emerged and attracted a huge amount of funds due to their convenient interactive services. However, the increasing market value of DApps has also attracted many malicious actors. In recent years, hundreds of security incidents have occurred, resulting in billions of dollars in financial losses. Among their causes, vulnerability exploits serve as one of the main reasons for highlighting the importance of vulnerability detection. In particular, as smart contracts utilize more state variables to support more complex functionalities, *state-relevant vulnerabilities* that can only be triggered in specific *vulnerable state* have also emerged. Furthermore, these vulnerabilities are both difficult to discover and frequently appear in DApps, which presents new challenges for vulnerability detection.

To address this challenge, we propose a state-aware fuzzer, SmartFuzz , that aims to explore smart contract states and increase the probability of reaching *vulnerable state*. Our main idea is a state-aware seed scheduling mechanism that reserves seeds reaching new states and mutates them to explore more states. To improve the effectiveness of state exploration, we implement the seed scheduling with the following key insights: (1) we construct the state space consisting of essential variables (e.g., variables related to sensitive operations) to compress the exploration state; (2) we design a novel strategy to identify interesting states, thereby improving the efficiency of exploration. To evaluate the performance of SmartFuzz , we adopt it on a dataset consisting of 116 DApps from security incidents. The experimental results demonstrate that the fuzzer with state-aware seed scheduling achieves 13% higher state coverage than that with code-coverage-guided seed scheduling. Meanwhile, SmartFuzz effectively detects state-relevant vulnerabilities due to its state exploration strategy.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Smart contract, Fuzzing, Vulnerability detection

ACM Reference Format:
Anonymous Author(s). 2018. SmartFuzz: Making Smart Contract Fuzzing Smarter via State-Aware Feedback. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

With the rapid development of blockchain, smart contracts and their decentralized applications (DApps) have been increasingly adopted in various fields, such as finance and gaming [35], providing convenient interactive services. In recent years, DApps have attracted a large number of users and funds, and their total market value once exceeded one trillion dollars [2]. However, as DApps flourish, the increasing market value also attracts many hackers to attack. In the past few years, hundreds of security incidents have occurred, such as vulnerability exploits and rug pull [40], which have caused more than 30 billion dollars in financial losses. In particular, according to SlowMist statistics [6], the vulnerabilities hiding in smart contracts are the leading cause of security incidents. For example, Parity Wallet is drained of millions of dollars due to its lack of access control

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

vulnerability [1]. This situation makes vulnerability detection an essential task in protecting smart contracts.

Smart contracts are inherently *stateful* programs, meaning they maintain a dynamic state and execute various actions based on that state. As smart contracts evolve to support more complex functionalities that involve multiple state-related actions, the detection of vulnerabilities becomes increasingly challenging. Given the characteristic of smart contracts, we classify smart contract vulnerabilities into two main categories:

- *state-irrelevant vulnerability* does not depend on the contract's dynamic state to exploit, such as timestamp dependency bugs and mishandled exception bugs [24].
- *state-relevant vulnerability* are influenced by the smart contract's state, such as price manipulation attacks [22].

State-relevant vulnerabilities are generally more complex and difficult to detect compared to state-irrelevant ones because they can only be triggered under specific dynamic states. State-relevant vulnerabilities pose a significant challenge in smart contract security, as they are often deeply hidden and can be overlooked during typical security assessments.

Challenges. Fuzz testing [41] is a dynamic testing technique that generates test cases and executes smart contracts in various states, making it a popular approach for detecting smart contract vulnerabilities. Researchers have proposed many evolutionary fuzzing techniques to improve efficiency and effectiveness in vulnerability detection [16, 26, 29, 32, 34]. A critical aspect of these techniques is the fitness metric of the fuzzer, which significantly influences fuzzing performance. This metric allows for the evaluation of seed quality, enabling the prioritization of fuzzing resources towards seeds with higher fitness values, thus optimizing the fuzzing budget allocation [36]. However, existing fitness metrics are still limited in helping fuzzers detect vulnerabilities, especially state-relevant ones. We elaborate their limitations as follows:

- *Code Coverage* is widely used in many smart contract fuzzers [16, 26, 34]. It gives higher fitness values to the seeds that cover more code, so that guides fuzzers to explore more code. However, for stateful programs like smart contracts, increasing code coverage does not guarantee the discover of vulnerabilities.
- *Distance* is another common metric used to identify seeds likely to reach unexplored areas, such as branch conditions. This metric typically relies on condition operations in smart contracts. However, it becomes inefficient if vulnerabilities do not reside in branches with specific conditions. Both code coverage and distance focus on expanding code exploration, which does not guarantee effective vulnerability discovery [32]. Moreover, the same code can execute differently depending on the contract's state (discussed further in Section 3).
- *Vulnerability* is an emerging fitness metric designed to guide fuzzers directly towards seeds likely to trigger vulnerabilities. It requires static analysis to locate likely vulnerable code or uses machine learning to find vulnerable transaction sequence. For example, RLF [32] uses reinforcement learning to improve the efficacy of vulnerability detection. However, this approach heavily depends on prior knowledge, such as high-quality labeled data for training, which restricts its scope in identifying diverse vulnerabilities.

Therefore, we need to make the smart contract fuzzing *smarter* to efficiently and effectively reveal vulnerabilities, especially for state-relevant vulnerabilities. As mentioned above, we note that smart contracts are *stateful* programs and state-relevant vulnerabilities require some specific states to trigger. Thus, we aim to make the fuzzer cover as many different states as possible to increase the likelihood of achieving the state to trigger vulnerability. Referring to code-coverage-guided seed scheduling that reserves the seeds that cover new code for mutation, we can directly reserve and mutate the seeds that reach new states to explore more states. However, this strategy would

reserve too many redundant seeds and invalidate the state exploration (explained in Section 5.3). To this end, we improve the efficacy of state exploration with two key insights: (1) we construct the state space consisting of essential variables to compress the exploration space; (2) we use a novel strategy to identify interesting states to improve the efficiency of state exploration. Based on them, we propose our state-aware seed scheduling to reserve and mutate the seed that reaches interesting states so as to make the fuzzer efficiently explore smart contract states.

In this paper, we propose SmartFuzz to efficiently explore smart contract states and detect smart contract vulnerabilities, especially state-relevant cases. At each fuzzing iteration, we use *state selector* to extract the essential variables, including *common*, *sensitive* and *token* variables. In particular, we design a caching mechanism to reduce the overhead of state extraction for fuzzing. After that, we use *state explorer* to identify interesting states. Specifically, we divide the state space of variables into multiple areas and identify an interesting state if it is located in an area that has not been explored. If a seed reaches an interesting state, we compress and reserve it in the seed corpus for subsequent mutation.

In experiments, we adopt SmartFuzz on a dataset consisting of 116 DApps from historical security incidents. The experimental results demonstrate the efficacy of state-aware seed scheduling that achieves 13% more state coverage and 2% more code coverage than code-coverage-guided seed scheduling. In particular, the *state explorer* plays an essential role in improving the state coverage that helps SmartFuzz explore 37% more states. In addition, SmartFuzz successfully generates the exploits for 11 vulnerabilities in DApps.

We summarize the main contributions of this paper as follows:

- We propose a state-aware seed scheduling strategy that improves the efficacy of state exploration of smart contracts.
- We implement a fuzzing framework, SmartFuzz , that aims to explore smart contract states and detect vulnerabilities, especially state-relevant vulnerabilities.
- We perform a comprehensive evaluation of SmartFuzz that demonstrates the performance of SmartFuzz in testing smart contracts.

2 Background

2.1 Smart Contract

Smart contracts are Turing-complete programs running on the blockchain that enable trusted execution in a decentralized environment [4]. They are always written in Solidity [7], the most popular programming language for smart contracts. Through compilation, we can obtain the bytecode of the smart contract and deploy it on the blockchain. After that, the smart contract has its own *account state* stored on the blockchain. In general, the account state consists of the balance of ether and state variables stored in the *storage* (mapping *bytes32* to *bytes32*) [7]. Once receiving the transaction, the smart contract with the original state s is automatically executed within Ethereum Virtual Machine (EVM), which can be viewed as a state transfer function ($S \times T \rightarrow S$) mapping a state $s \in S$ and a transaction $t \in T$ to a new state $s' \in S$. After that, the new state s' will be stored in blockchain if the transaction does not revert. In particular, smart contracts can also emit new transactions (namely internal transactions) to call other contracts, making the execution logs form a call flow tree [33].

In addition to bytecode, the compilation also provides the Application Binary Interface (ABI) of the smart contract, which can serve as the interface of functions for the fuzzer to generate test cases. According to *stateMutability*, smart contract functions can be divided into three parts: (1) *View* functions only read but do not alter the state variables in contracts, which are commonly used to query smart contract state, such as the function *balanceOf* in ERC20 contracts [3]. In particular,

```
contract Example {
    uint256 public constant bar = T;
    uint256 public rate = 1;
    mapping(address => uint256) public balances;
    function deposit() payable public {
        balances[msg.sender] += msg.value;
    }
    function withdraw() public {
        uint256 amount = (rate * balances[msg.sender]) /bar;
        payable(msg.sender).transfer(amount);
        balances[msg.sender] = 0;
    }
    function increase(uint256 x) public {
        require(x <= rate);
        rate += 1;
    }
    function decrease(uint256 x) public {
        require(x >= rate);
        rate -= 1;
    }
    function getReturnRate() view public returns (uint256){
        return rate/bar;
    }
}
```

Fig. 1. A simplified smart contract with state-relevant vulnerability.

the variables with the *public* property [7] will be automatically generated their corresponding *view* functions by the Solidity compiler, so that we can query the value of these state variables by calling their *view* functions. (2) *Pure* functions do not read or modify any state variables in contracts. (3) Other functions can read and modify the state variables in contracts.

2.2 Fuzzing

Fuzzing is one of the most popular automated techniques for detecting vulnerabilities, both in the field of traditional programs and smart contracts [25]. The core idea of fuzzing is to rapidly generate a vast number of seeds (test cases) and automatically execute the target program. Specifically, the test cases of smart contracts are transaction sequences; each transaction includes sender, receiver, amount of ethers, function parameters, etc. To improve the performance of fuzzing, researchers have proposed various fitness metrics for seed scheduling, such as code coverage and distance, etc. Fuzzers use fitness metrics to assess the quality of seeds and allocate more energy to high-quality seeds. During the fuzzing procedure, we can catch program crashes or use oracles [27] to analyze execution logs and identify vulnerabilities that do not crash the program. In particular, since smart contracts revert to their original states (instead of crashing) when encountering errors, we need to use oracles to detect vulnerabilities in smart contracts.

3 Motivation

In this section, we use a motivating example to illustrate the limitations of existing works in detecting state-relevant vulnerabilities and demonstrate our insights on how to address these limitations.

3.1 Motivating Example

Fig. 1 gives a simplified smart contract with three variables, including `rate`, `balances`, and `bar`, which equals a constant value `T`. In addition, there are four critical functions that help maintain the state of the smart contract. The function `deposit` allows users to deposit their ether in the

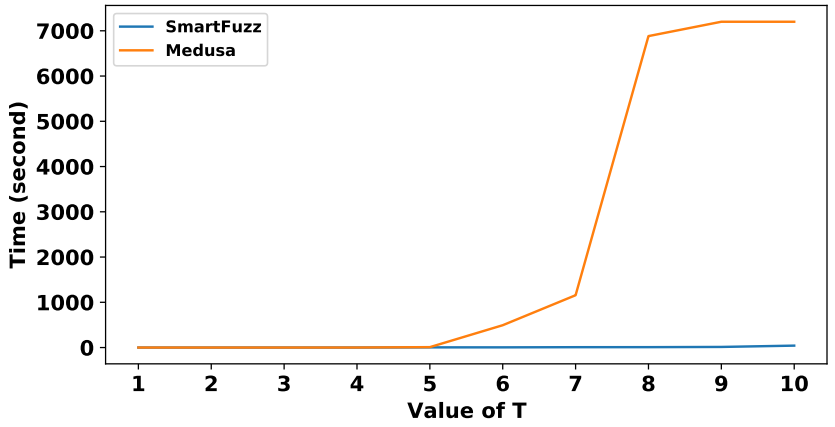

Fig. 2. Time (seconds) to detect the bug with different values of T.

contract and increase the corresponding balance, while the function `withdraw` allows users to retrieve their deposited ether at a certain rate (i.e., $\frac{rate}{bar}$). Meanwhile, this contract contains two functions to change a critical state variable (i.e., `rate`). The function `increase` increases the `rate` by one if the argument `x` is smaller than `rate`. And, the function `decrease` decreases the `rate` by one if the argument `x` is greater than `rate`.

However, there exists a state-relevant vulnerability hiding in the contract. By reaching a specific state, attackers are allowed to extract the excess ether. Assuming that the contract has 10 ether and T = 1, the exploit (i.e., the transacction seqeuence to trigger bugs) should be [`deposit{value:1 ether}`, `increase(0)`, `withdraw`]. In this transaction sequence, we first call `deposit()` function with one ether, then call `increase` function with valid argument (i.e., $x \leq 1$). Consequently, the state variable `rate` is changed to two, thus $\frac{rate}{bar} = 2$. In the end, we can call `withdraw` function to extract two ether from the contract, compared to one ether that we deposit. In general, once the contract is under the state where `rate` is larger than `bar` (marked as *vulnerable state*), anyone can withdraw more ether than they have deposited.

3.2 Our Insights

Limitations of existing research. As the configured constant value `T` increases, the difficulty for fuzzers to automatically generate exploits grows exponentially. This is because fuzzers must trigger the `increase` function `T` times more than the `decrease` function to reach the *vulnerable state*. However, current smart contract fuzzers, such as Medusa [10], which employs a code-coverage-guided seed scheduling, require significantly more time or even fail to achieve that, as demonstrated by the experimental results shown in Fig. 2. Next, we elaborate on limitations based on the fitness metrics of fuzzers:

- *Code Coverage* metric makes the fuzzers cover more code and quickly reach the maximum coverage in this example contract, but cannot help the fuzzer find the *vulnerable state*.
- *Distance* metric fail to provide effective feedback to the fuzzers, since the function `withdraw` does not contain any condition operation for calculating the amount of ether being transferred.
- *Vulnerability* metric requires the characteristics of the vulnerability as prior knowledge (e.g., the location of the vulnerability), which is hard to define since the vulnerability is related to multiple functions.

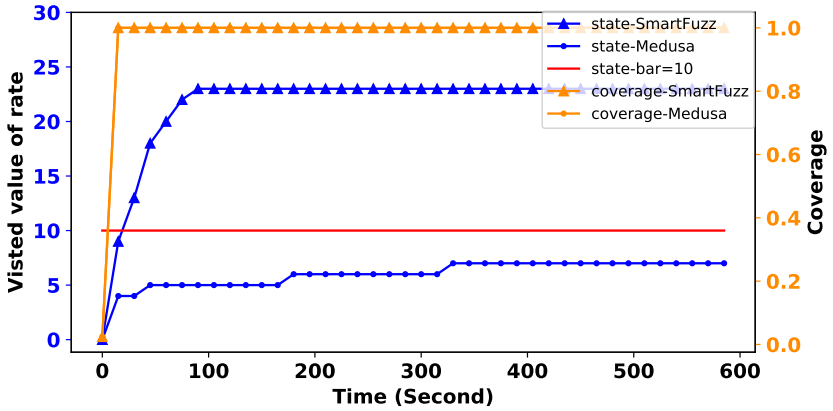

Fig. 3. Reached value of `rate` along fuzzing period.

In general, the state-relevant vulnerability in this example brings new challenges to existing smart contract fuzzers. Unfortunately, these vulnerabilities appear in many smart contracts, and many of them have been exploited in the past, which motivates us to propose a more effective fitness metric to enhance the performance of fuzzers.

Key insights. To effectively trigger state-relevant vulnerabilities, it is crucial to increase state coverage rather than merely focusing on other metrics like code coverage. In this study, we identify the state variable `rate` as a key factor in reaching the *vulnerable state*. Therefore, we analyze its values during the fuzzing process. We set $\tau = 10$ and track the values of `rate` over time, as shown in Fig. 3. For the *vulnerable state* to be reached, the value of `rate` must exceed 10.

Notably, while the code-coverage-guided fuzzer, Medusa, achieves 100% code coverage quickly, its effectiveness drops once no new code paths remain. This limitation arises because the fitness metric in Medusa is solely based on code coverage, without considering the states of the smart contract. In contrast, guiding the fuzzer to explore a broader range of states significantly enhances the likelihood of reaching the *vulnerable state*.

Instead of merely covering new code, one potential strategy is to employ a seed scheduling method to prioritize transactions reaching new interesting states. In this example, we define the exploration scope using the state variable `rate` and consider transactions that result in new values of `rate` as interesting. This strategy allows the fuzzer to efficiently generate transactions that progressively explore new interesting values of states, quickly reaching the *vulnerable state*, as demonstrated by the experiment results of SmartFuzz in Fig. 2 and Fig. 3.

In order to achieve high efficacy in exploring smart contract state, there are two key challenges for integrating seed scheduling strategy: (1) how to select variables to construct state space; (2) how to identify the discovery of new interesting state. SmartFuzz addresses these challenges with a novel method, detailed in Section 4.

4 SmartFuzz

This section introduces our methodology for designing SmartFuzz . We first briefly introduce the main idea of our state-aware fuzzing (Section 4.1). Then, we present the architecture of SmartFuzz (Section 4.2) and its two crucial building blocks: state extractor (Section 4.3) and state explorer (Section 4.4).

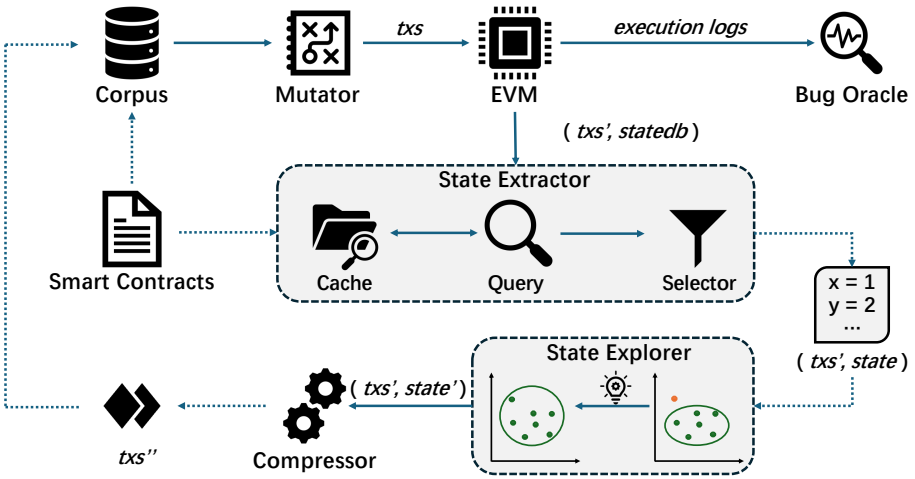

Fig. 4. Overview of SmartFuzz .

4.1 State-Aware Fuzzing

In this paper, our aim is to guide the fuzzer in exploring more smart contract states. We note that the code-coverage-guided seed scheduling is that reserves the seeds covering new code and mutates them to cover more code. Similarly, we attempt to reserve the seeds that reach a new state for subsequent mutation, where the definition of a new state is as follows:

Definition 4.1. Given that S is the set of all states, $s \in S$ denotes a state. $S' \subseteq S$ denotes the set of reached states, and $S' = \bigcup s_i$ where s_i is the reached state. For a state s', we define it as a new state if $s' \,!\in S'$.

We notice that the new state s' is generally located at the boundary of S', which can help to reach other new states. As the variable `rate` in the example contract (Fig. 1), we assume the condition that $S' = \{0 \leq \texttt{rate} \leq N-1\}$ and we just find a new seed *txs* to make the smart contract reach a new state ($\texttt{rate} = N$). In this case, mutating *txs* has a higher probability of exploring a wider range of states (i.e., the state that $\texttt{rate} > N$) compared to the previous seeds. Therefore, we aim to design a state-aware seed scheduling that reverses and mutates the seeds that make the smart contract reach a new state for state exploration.

However, the state variables of the smart contract are stored in a *storage* that maps *bytes32* to *bytes32*, which makes the complete state space of the smart contract nearly infinite. If we directly regard the whole *storage* as the state space and identify a new state by checking whether there exists a slot storing a new value, we will reserve many seeds that reach homogeneous states (marked as *homogeneous seeds*), which causes state-aware seed scheduling invalid.

Therefore, we design an effective strategy to identify new and interesting states that enhance the quality of seeds, whose key insights are as follows:

- *State space*: constructing the state space with essential variables to compress the exploration state (Section 4.3.2).
- *State identification*: identifying the new and interesting state to reduce redundant and homogeneous seeds to improve the efficiency of state exploration (Section 4.4).

```
interface Example {
    function bar() view returns (uint256);
    function rate() view returns (uint256);
    function balances(address) view returns (uint256);
    function getReturnRate() view public returns (uint256)
}
```

Fig. 5. Interface of *view* functions in Example contract.

4.2 Architecture of SmartFuzz

We propose a state-aware fuzzer, namely SmartFuzz , and present its architecture in Fig. 4. Given the ABIs of the target smart contracts, SmartFuzz initialize the *Corpus*, which is responsible for saving seeds (i.e., transaction sequences). At each fuzzing iteration, a seed is selected from *Corpus* and mutated to a new transaction sequence txs. Then, given the initial account states, *EVM* executes the transactions in txs one by one. After each transaction is executed, *EVM* outputs the transaction sequence has been executed and the new account states, marked as (txs', $statedb$). Based on them, *State Extractor* queries the essential variables as the state for further analysis ($state$). Moreover, *State Explorer* identifies if $state$ is a new and interesting state. If yes, *State Explorer* updates the reached scope of the state space, and *Compressor* removes the redundant transactions in txs to obtain a compressed transaction sequence (txs''), which will be saved in *Corpus* for subsequent mutation and testing. In addition, *Bug Oracle* is responsible for detecting smart contract vulnerabilities according to the execution logs of *EVM*.

4.3 State Extractor

State Extractor is responsible for extracting the smart contract states after each transaction is executed. We first illustrate the basic mechanism for querying variables (*State Query* in Section 4.3.1), based on which we next describe how we select essential variables for the construction of state space (*State Selector* in Section 4.3.2). In addition, we design a cache mechanism to improve the efficiency of extracting state for reducing its overhead to fuzzing 4.3.3.

4.3.1 State Query. In this part, we aim to design a versatile state query mechanism to extract the smart contract state during the testing.

After each transaction is executed, *EVM* outputs the *statedb*, which consists of the ether balance and state variables of smart contracts. For ether balance, we can directly query it from *statedb* by invoking its function *GetBalance* with the smart contract's address as the parameter [1]. For the state variable, it is stored in *storage*, a data structure that maps *bytes32* to storage slots of type *bytes32*. As a result, we cannot directly query the value and type of the state variable due to the lack of its corresponding storage slot location. Next, we discuss how to query state variables in contracts.

To recover the value of the state variable, previous research commonly uses the storage layout to locate the corresponding storage slot and recover its type and value [15, 23]. However, the storage layout is derived from the source code of the smart contract, which limits the application scope of state recovery (i.e., it is only suitable for the smart contract with source code). Therefore, a more versatile approach is needed to query state variables in smart contracts.

Variables From View Functions. We notice that there exist *view* functions in smart contracts that allow us to query the smart contract state. For a better understanding, we use the example contract in Fig. 1 as an example; the interfaces of their *view* functions are presented in Fig. 5. Among these *view* functions, we divide them into two parts for discussion: (1) the functions that directly return the values of state variables, which are commonly automatically generated

[1] https://github.com/ethereum/go-ethereum/blob/master/core/state/statedb.go#L329

Table 1. State space.

Category	Remark
Common Variables	Variables from the *view* functions with no parameter and non-constant.
Sensitive Variables	Variables from the *view* functions with parameters and related to sensitive operations (e.g., transfer, selfdestruct).
Token Variables	The balance of ether and ERC20 token of sender and contracts.

by the compiler according to the state variables with the *public* property. For example, the state variable `balances` (`mapping(address => uint256)`) has its corresponding *view* function with interface `balances(address) returns (uint256)`. Thus, we can query state variables with the *public* property by calling their *view* functions. (2) the functions that read state variables and perform further calculation for return values (e.g., the function `getReturnRate()` that returns the value of $\frac{rate}{bar}$), which provide more interesting cases in addition to state variables. In general, although we cannot query the state variables with *private* property, the variables from *view* functions still provide sufficient variables for subsequent analysis.

View-Function-based State Query. Considering that the fuzzer needs the ABI of the smart contract to generate transactions, we can also query variables of state based on the *view* functions in ABI (without the source code of the smart contract). To this end, we choose to serve the return value of *view* functions as the variables of the smart contract state. For better understanding, we denote $addr$, $func$, $param$ as the address of the smart contract, the *view* function, and its parameter, respectively. And, $var = EVM.call(addr, func, param)$ denotes that the variable var records the return value of the *view* function.

4.3.2 State Selector. In this part, our objective is to construct a state space consisting of the essential variables related to the important state (e.g., *vulnerable state*) or sensitive operations (e.g., token transfer), so that the fuzzer can effectively detect vulnerabilities when exploring the state space. Meanwhile, we also need to reduce the redundant variables in the state space to prevent the fuzzer from exploring the useless state area. To achieve this goal, we construct the state space by extracting *Common*, *Finance* and *Token* variables from *view* functions, which are listed in Table 1. For better understanding, we use $state$ to record the variables in the state space, which maps the name to the value of a variable.

Common Variables. For the *view* function with no parameter, it always returns the values with *Value Type* [7] (e.g., *address*, *uint256*). We serve them as the variables in the state space since they always represent the common and basic property of the smart contract. For example, we make the function `rate()` as a variable, and we record $state[rate()] = EVM.call(addr, rate)$ for extraction. In particular, some *view* functions return the values of constant (e.g., the function `bar()`) that never change during testing, we discard them to reduce redundant variables.

Sensitive Variables. For the *view* function with parameters, it always reads the state variables with dynamic types (e.g., *dynamic array*, *mapping*) [7], whose return value relies on the input parameter. As a result, we cannot iterate over all possible parameters to extract variables, otherwise the state space will be too large to explore. To address this problem, we heuristically reserve the var that is related to the sensitive operations in smart contract (e.g., token (ether) transfer, *selfdesturt*), which are often associated with smart contract vulnerabilities (e.g., Price Manipulation). Specifically, we originally call the *view* function with some interesting parameters (e.g., the address of the sender and the contract) with a tracer to record their loaded slots, marked as an array $slotArray = [(var_0, param_0, slots_0), ...]$. After the transaction including sensitive operations is executed, we reserve the var_i as an essential variable if $slots_i \cap slots_{tx} != \emptyset$, where $slots_{tx}$ is the loaded slots during the execution of the transaction. In particular, the sensitive variables will be dynamically included in the state space during the testing procedure.

Algorithm 1: Workflow of querying variable from *view* function with cache mechanism.

```
input  : txs', func, param, cache, statedb, tracer
output : var
id := hash(addr, func, param);
isUpdated := false;
/* Check if the variable is updated.                                  */
if id ∈ cache then
    for slot := range cache[id].SlotValueMap do
        if cache[id].SlotValueMap[slot] != statedb.GetState(addr, slot) then
            isUpdated = true;
            break;
        end
    end
end
if id ! ∈ cache || isUpdated then
    /* Call view function.                                            */
    EVM := newEVM(statedb, tracer);
    cache[id].Variable = EVM.call(addr, func, param);
    cache[id].SlotValueMap = tracer.getSlotValueMap();
end
var = cache[id].Variable;
```

Token Variables. In addition to the variables from the *view* functions of the smart contract, we also record the balances of ether and ERC20 tokens that represent the financial properties of smart contracts. For ether, we directly read the balance of *addr* from account state *statedb*. For each related ERC20 token *token*, we obtain the balance of the token by calling the function `balanceOf` with $param = addr$ in token contract, i.e., $EVM.call(token, balanceOf, addr)$.

4.3.3 State Cache. As mentioned above, we perform the state query after each transaction is executed, which means that we have to call the same *view* functions many times during the testing procedure. Obviously, these repeated calls will cause too much overhead and significantly reduce the efficiency of fuzzing.

We notice that the core logic of the *view* function is reading specific slots in *statedb*, based on which it performs some calculations and returns values. In particular, if the values in specific slots are not changed, the *view* function returns the same values. Considering that directly reading the *storage* slots in *statedb* is much faster than execution of the *view* function, we can decrease the times of calling the *view* functions by caching the return values, specific *storage* slots, and their values, based on which we only call the *view* functions if the values in *storage* slots are changed.

The whole workflow for querying the variable from the *view* function with the cache mechanism is presented in Algorithm 1. Given an input tuple $(addr, func, param)$ that represents the address, the *view* function and the function parameters of the smart contract, we first calculate the hash value of the tuple to obtain the cache identifier *id*. After that, we call the *view* function in two conditions: (1) the *id* has not been stored in *cache*; (2) the *id* has been stored in *cache* and there exists a *slot* that satisfies:

$$cache[id].SlotValueMap[slot] ! = statedb.GetState(addr, slot),$$

Algorithm 2: Workflow of identifying new and interesting value of the variable.

```
input  : variable, valueSet, updateBar, value
output : newValueFlag
newValueFlag := false;
if IsNumeric(variable) then
    if len(valueSet) == updateBar then
        /* Update division of value scope.                                  */
        minV, maxV := min(valueSet), max(valueSet);
        valueSet.Parts = {[Min, minV), (maxV, Max]};
        valueSet.Parts.extend({[a_i, a_{i+1}] | a_i = minV + ((maxV − minV) × i)/10, i = 0, 1, ..., 9});
    end
    part := valueSet.Parts.Match(value);
    if !part.IsReached then
        part.IsReach = true;
        newValueFlag = true;
    end
end
if IsNonNumeric(variable) then
    if value ! ∈ valueSet then
        newValueFlag = true;
    end
end
valueSet.Add(value);
```

where $cache[id].SlotValueMap$ records the loaded slots and their values of the last call with a data structure mapping *slot* to *value* in type *bytes32*, $statedb.GetState(addr, slot)$ returns the value (type *bytes32*) that is stored in the *slot* of *storage* in address *addr*. If one of the above conditions is satisfied, we apply $(addr, func, param, statedb)$ to *EVM* with a tracer to record loaded storage slots and their values, based on which we store the new return value of the *view* function in $cache[id].SlotValueMap.Variable$ and update $cache[id].SlotValueMap$. Finally, we can obtain the cached return value of the *view* function by querying $cache[id].SlotValueMap.Variable$. In this way, we can significantly improve the efficiency of querying variables during the fuzzing procedure.

4.4 State Explorer

State Explorer is responsible for identifying a new and interesting state compared with previously reached states based on the above-constructed state space. Specifically, we need to identify whether the variable has a new value compared to its previous values for each variable in *state*. However, if we identify a new value by directly comparing it with previous values, we would identify many new and homogeneous states. As the variable `balances[msg.sender]` in example contract (Fig. 1), it would be frequently modified to a huge number of values during fuzzing, but whether the function `withdraw` transfers ether depends only on whether `balances[msg.sender]` is zero. That is, the values of `balances[msg.sender]` greater than 0 are homogeneous, but are non-homogeneous to zero. Consequently, we would reserve too many redundant seeds and make seed scheduling invalid.

To address this problem, we design a heuristic identification mechanism to reduce the number of homogeneous cases within new values of the variable. Next, we describe the mechanism according to the type of the variables: (1) *Numeric Variables* that with numeric types (e.g., *uint256*); (2) *Non-numeric variables* that with non-numeric types (e.g., *string*, *address*). The whole identification mechanism is presented in Algorithm 2.

4.4.1 Numeric Variables. We notice that the homogeneous values of the variable are always in the same part, while the non-homogeneous values are divided into multiple parts by some specific values. As the variable `rate` in example contract (Fig. 1), when the value of `rate` is in $[1, \mathtt{T})$, the amount of ether withdrawn is less than `balances[msg.sender]`; when the value of `rate` is larger than `T`, the contract reaches *vulnerable state* and we could withdraw more ether than our deposit. In this example, the value of `T` divides the value scope of `rate` into two parts, and the `withdraw` function performs substantially different behavior as the value of `rate` belongs to different parts. It motivates us to divide the value scope of the variable.

Therefore, we attempt to divide the value scope of the variable into multiple parts, based on which we identify the new and interesting value of the variable. We assume that the value scope of the variable is $[Min, Max]$, which is determined by the type of the variable (e.g., the variable with *uint256* type has value scope $[0, 2^{256}]$. During the testing procedure, we use a *valueSet* to reserve the reached values of the variable. Once the capacity of *valueSet* reaches *updateBar*, we perform division on the value scope. Specifically, we obtain the maximum and minimum values of *valueSet* as $maxV$ and $minV$, based on which we divide the value scope into three parts: $[Min, minV)$, $[minV, maxV]$, $(maxV, Max]$. For the part $[minV, maxV]$, we further divide it into 10 equal parts $\{[a_i, a_{i+1}] | a_i = minV + \frac{(maxV - minV) \times i}{10}, i = 0, 1, ..., 9\}$. After that, if a new value belongs to a part that is never reached, we regard the value as a new and interesting value and mark the part as the reached part.

In particular, we redivided the range of values of the variable if the capacity of *valueSet* reaches some specific values, to avoid a deadlock in state exploration.

4.4.2 Non-numeric Variables. Since the value scope of non-numeric variables cannot perform division like numeric variables, we directly identify a new value by comparing it with previous values. Considering that the non-numeric variable in the smart contract does not involve many values, this identification mechanism is still effective in general situations.

4.5 Compressor.

In this paper, our strategy for exploring state space is to reserve the transaction sequence (txs') that reaches a new and interesting state ($state'$) for subsequent mutation. Within the sequence, we notice that some transactions have no effect on reaching $state'$ (marked as *invalid transactions*), which may reduce the efficiency in state exploration. Therefore, we design a compressor to remove the *invalid transactions* from txs' for $state'$, whose workflow is presented in Algorithm 3. First, we set the compressed sequence txs'' to txs'. In each iteration, we generate a possible sequence txs_p by removing one transaction from txs'' and check if txs_p reaches $state'$. If yes, we set txs'' to txs_p. Through multiple iterations, we obtain the compressed sequence txs'' that still reaches the new state $state'$.

4.6 Implementation

Based on Medusa [10], we implement *State Extractor*, *State Explorer* and *Compressor*, and integrate them as SmartFuzz . In addition, we implement some functionalities as follows: (1) We modify the EVM to dynamically obtain the state from the Ethereum node, based on which our fuzzer can

Algorithm 3: Workflow of Compressor

```
input  : txs', state', statedb_init
output : txs''
txs'' := txs';
for i := len(txs'') − 1; i >= 0; i − − do
    txs_p := concat(txs''[: i], txs''[i + 1 :]);
    EVM := newEVM(statedb_init);
    state'' := EVM.execute(txs_p);
    if IsMatch(state', state'') then
        txs'' = txs_p;
    end
end
```

support on-chain fuzzing, i.e., the generated transaction sequences can be executed in the on-chain environment. In this case, we can test on-chain smart contracts without a complex deployment procedure. (2) We design an invariant checker that allows us to conveniently design bug oracles according to the smart contract state extracted by *State Extractor*. (3) We make a helper contract that supports the launch of the internal transactions to provide more comprehensive testing, including converting external transactions into internal transactions to call and launching internal transactions when the internal transaction calls the helper contract. In total, we implement the entire SmartFuzz with 5,000 extra lines of code in golang [5].

5 Experiments

We conduct experiments to evaluate the performance of SmartFuzz and aim to answer the following research questions:

- RQ1: What is the efficacy of incorporating state-aware feedback in the exploration of smart contracts?
- RQ2: How do the construction of the state space and the identification of interesting states influence the effectiveness of SmartFuzz ?
- RQ3: In what ways does the state-aware seed scheduling strategy improve SmartFuzz 's effectiveness in detecting vulnerabilities?

5.1 Experimental Setup

Dataset. We construct our dataset for evaluation based on a well-known hacking repository, *DeFiHackLabs*, which records the hacking incidents of DApps and their proof of concept (PoC) and has obtained more than 5k stars on GitHub [9]. From the repository, we collect the hacking incidents that occurred from April 2020 to March 2024, and manually extract the related addresses of DApps and the block height before the DApps were hacked. In total, we construct a dataset consisting of 116 DApps that involves 377 on-chain contracts.

Implementation. To conduct experiments, we insert the addresses of DApps and the corresponding block height, and SmartFuzz will automatically crawl the ABIs of the contracts from Etherscan [8] and perform on-chain fuzzing based on our local Ethereum node running with Reth [12]. All experiments are conducted on a Ubuntu machine with an Intel(R) Core(TM) i9-10980XE (3.00GHz) CPU (36 cores) and 256GB of memory.

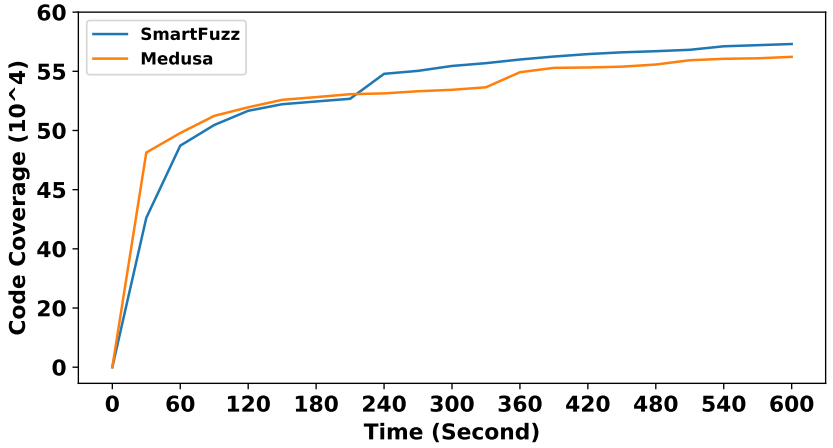

Fig. 6. Code coverage along fuzzing time.

5.2 Efficacy of State Exploration

In this experiment, we evaluate the efficacy of incorporating state-aware feedback with regard to both code and state coverage. Code coverage is measured by the number of executed instructions, while state coverage is assessed by counting the distinct values written to *storage* slots. We apply SmartFuzz to each DApp in our dataset for 10 minutes and record the evaluation metrics.

To benchmark our results, we also apply the code-coverage-guided fuzzer, Medusa, to our dataset, using the same experimental configuration. Since both SmartFuzz and Medusa use the same fuzzing framework, any differences in their performance can be attributed solely to their seed scheduling strategies, ensuring a fair comparison.

Code Coverage. Fig. 6 shows the experiment results regarding code coverage, tracking the total number of executed instructions across all DApps over the fuzzing period. Initially, Medusa achieves a higher code coverage than SmartFuzz , highlighting the effectiveness of its code-coverage-guided seed scheduling in covering shallow code. However, as fuzzing continues, the growth rate of Medusa's code coverage slows down and eventually gets flat, while SmartFuzz 's code coverage continues to increase, ultimately surpassing 2% than that of Medusa. This is because some code branches are only accessible when the smart contract is in specific states (e.g., when state variables meet certain conditions), and state-aware seed scheduling is more efficient at exploring these states than code-coverage-guided seed scheduling. Overall, state-aware seed scheduling enhances the performance of the fuzzer in achieving higher code coverage.

State Coverage. Fig. 7 shows the experiment results regarding state coverage, which measures the total number of distinct values stored in the *storage* slots of all DApps over the fuzzing period. Initially, Medusa also achieves higher state coverage than SmartFuzz . This initial advantage is due to overhead caused by extracting and identifying smart contract states, which lowers the fuzzing test speed early on. However, SmartFuzz has a faster growth rate of state coverage over time, eventually surpassing Medusa by 13% ($\frac{39.2-34.6}{34.6}$). The experimental results demonstrate the efficacy of our state-aware seed scheduling strategy in exploring smart contract states more thoroughly.

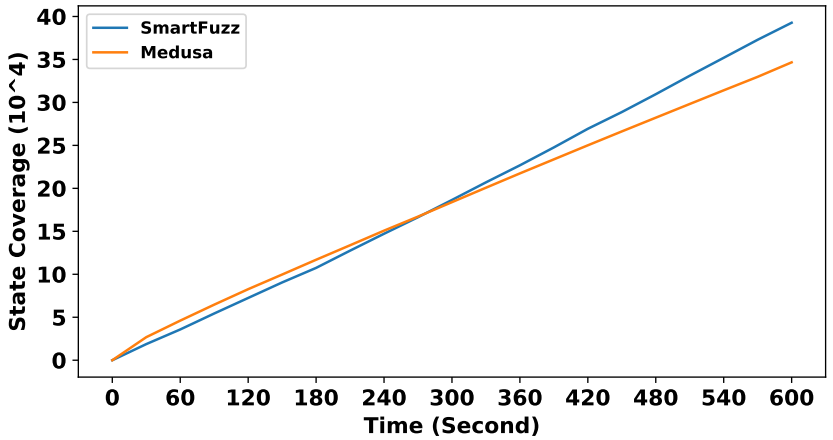

Fig. 7. State coverage along fuzzing period.

5.3 Ablation Study

In this experiment, we aim to evaluate the impact of our designed *state selector* and *state explorer* on state exploration. Specifically, we apply SmartFuzz to each DApp for 10 minutes, as well as its ablation as follows:

- Using *state selector* to construct the state space and *state explorer* to identify interesting states (marked as *SmartFuzz*).
- Only using *state selector* to construct the state space and identifying an interesting state if there exists a variable reaching a new value (marked as *SmartFuzz w/o Explorer*.
- Not using *state selector* to construct the state space and identifying an interesting state if there exists a *storage* slot being written in a new value (marked as *SmartFuzz w/o Selector*).

Consistent with the experiment in Section 5.2, we measure the state coverage as the total number of distinct values written to *storage* slots. The experiment results are shown in Fig. 8. Based on these results, we discuss the impact of the *state selector* and *state explorer* components individually.

Impact of State Selector. We examine the impact of the *state selector* by comparing the results of *SmartFuzz w/o Explorer* and *SmartFuzz w/o Selector*. We observe that *SmartFuzz w/o Selector* outperforms *SmartFuzz w/o Explorer* in exploring smart contract states. This outcome is due to both configurations using the same strategy for identifying interesting states (i.e., directly considering any new state as interesting), while *SmartFuzz w/o Explorer* requires additional time to extract essential variables and recover their values, which reduces its efficiency. Thus, using the *state selector* alone to construct the state space with essential variables does not enhance the fuzzing performance. However, *state selector* plays a crucial role in providing essential information to the *state explorer*, which then effectively identifies interesting states for deeper state exploration.

Impact of State Explorer. As shown in Fig. 8, SmartFuzz achieves the highest state coverage among the three configurations, 37% ($\frac{20.8-15.2}{15.2}$) more than *SmartFuzz w/o Explorer* and 31%($\frac{20.8-15.9}{15.9}$) more than *SmartFuzz w/o Selector*. This improvement is primarily due to the *state explorer*, which identifies new interesting states by dividing the value ranges of variables. This approach reduces the number of homogeneous seeds retained and facilitates the generation of new seeds, allowing the fuzzer to explore more states. In addition, we measure variable coverage as the number of distinct values of variables within the state space and record it during fuzzing. The results are shown in Fig. 9. Consistent with state coverage, SmartFuzz with *state explorer* enables variables

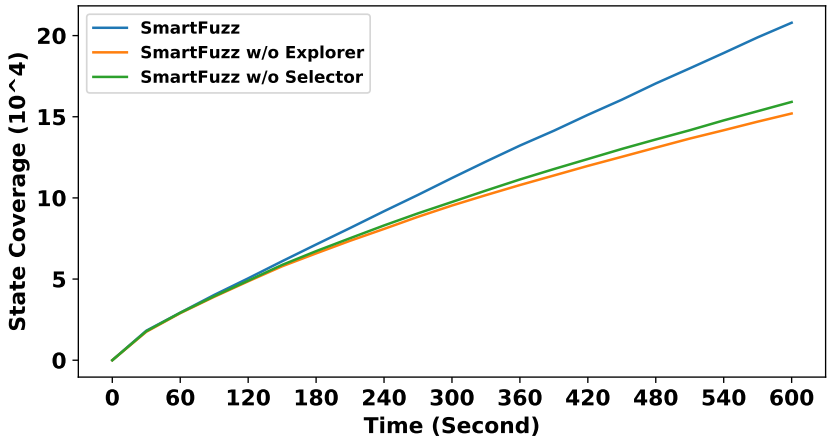

Fig. 8. State coverage along fuzzing period.

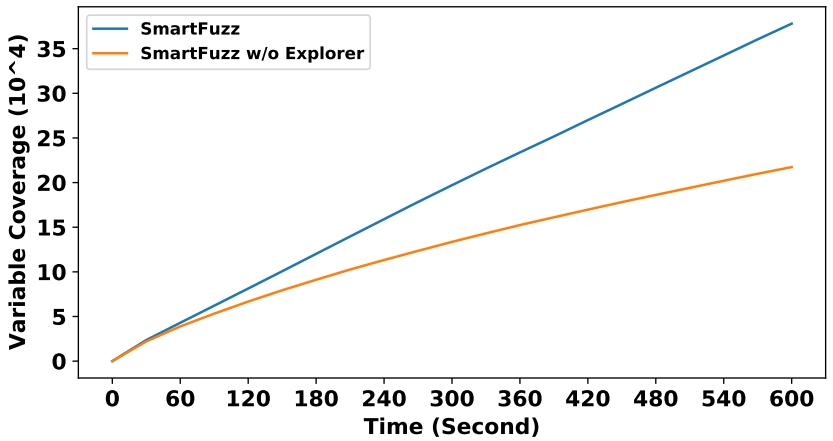

Fig. 9. Variable coverage along fuzzing time.

reach a greater variety values, achieving 74% ($\frac{37.8-21.7}{21.7}$) higher variable coverage. Overall, the *state explorer* significantly enhances SmartFuzz to identify new interesting states and explore a broader state space.

5.4 Effectiveness in Vulnerability Detection

In this experiment, we aim to evaluate the effectiveness of SmartFuzz in detecting vulnerabilities in smart contracts. For each DApp in our dataset, we first construct the bug oracles for DApps according to the PoC of hacking incidents. Specifically, we extract the profitable tokens of the attacker from the PoC and identify an exploit being found by the fuzzer if the transaction sequences successfully make the sender obtain more profitable tokens compared to the initial state. After that, we apply SmartFuzz to the DApp with the bug oracles for 1 hour. Among them, SmartFuzz successfully generates exploits for 11 vulnerabilities in DApps. Next, we use a case study to discuss how the state-aware seed scheduling strategy improves SmartFuzz 's effectiveness in detecting state-relevant vulnerabilities.

```
contract SimplifiedUniswapV2 {
    uint112 _reserve0 public;
    uint112 _reserve1 public;
    IERC20 token0 public;
    IERC20 token1 public;
    function getAmountOut(uint amountIn, uint reserveIn, uint reserveOut) returns (uint) {
        uint amountInWithFee = amountIn.mul(997);
        uint numerator = amountInWithFee.mul(reserveOut);
        uint denominator = reserveIn.mul(1000).add(amountInWithFee);
        return numerator / denominator;
    }
    function swap(uint amount0In, address to, bytes calldata data) external returns (uint) {
        _token0.transferFrom(msg.sender, address(this), amount0In);
        uint amount1Out = getAmountOut(amount0In, _reserve0, _reserve1);
        if (amount1Out > 0) _token1.transfer(to, amount1Out);
        sync();
        return amount1Out;
    }
    function sync() external {
        _reserve0, _reserve1 = token0.balanceOf(address(this)),token1.balanceOf(address(this))
    }
}
```

Fig. 10. A simplified swap contract.

The case is from a vulnerable DApp, namely *PANDORA* [11], which is a token contract (`token0`) and contains a bug that allows anyone to transfer tokens between itself and its corresponding *UniswapV2* contract. As shown in Fig. 11), the `swap` function allows users to swap `token0` to `token1`, the exchanged amount of `token1` (i.e., `amount1Out`) is negatively correlated with `_reserve0`. To achieve profit, the exploit needs to first utilize the bug to alter the `_reserve0` to reach the *vulnerable state* that makes `getAmountOut` function return a sufficiently large value, then call `swap` function to abnormally swap an excess amount of `token1`. In this case, SmartFuzz serves `_reserve0` as a variable of the state space for exploration. Consequently, SmartFuzz reserves the transaction sequence that makes `_reserve0` reach lower values and finally reach the *vulnerable state*.

6 Threats to Validity

In this paper, we aim to guide the fuzzer to explore more smart contract states and efficiently find the transaction sequences that reach *vulnerable state*. Unfortunately, exploring a wider state space may have no effect in detecting the state-irrelevant vulnerabilities (e.g., the vulnerability requires specific function parameters to trigger). However, SmartFuzz still has advantages in detecting state-relevant vulnerabilities (as the example in Section 3), which are widely seen in real-world security incidents.

7 Related Work

In this section, we discuss the fuzzing and other vulnerability detection tools for smart contracts.

7.1 Fuzzing for Smart Contract

fuzzing is one of the most popular techniques among a myriad of software-testing tools, which efficiently and automatically generate seeds for testing programs. In the field of smart contracts, fuzzing also receives extensive attention because of its extensive empirical evidence in discovering real-world vulnerabilities [16, 20, 26, 29, 32, 34]. Note that seed scheduling is essential for smart contract fuzzers; it utilizes fitness metric to assess the quality of seeds and allocate more fuzzing budget to the seeds with higher fitness values.

```
contract ERC404_PANDORA {

    function _preTransferCheck(address from, address to) internal {
        if (_uniswapV3Pool == address(0)) {
            _uniswapV3Pool = to;
            _disableTransferBlock = block.number + 50;
        } else if (_disableTransferBlock < block.number) {
            if (to == _uniswapV3Pool && !whitelist[from]) {
                revert("Transfers are disabled to sell tokens");
            }
        }

        if (initialBuyBlock[to] == 0 && from == _uniswapV3Pool) {
            initialBuyBlock[to] = block.number;
        } else if (to == _uniswapV3Pool && initialBuyBlock[from] != 0) {
            if (block.number - initialBuyBlock[from] > 2) {
                revert("Transfers are disabled after 2 block of initial buy");
            }
        }
    }
    /// @notice Function for mixed transfers
    /// @dev This function assumes id / native if amount less than or equal to current max id
    function transferFrom(
        address from,
        address to,
        uint256 amountOrId
    ) public virtual {
        _preTransferCheck(from, to);

        if (amountOrId <= minted) {
            if (from != _ownerOf[amountOrId]) {revert(" InvalidSender");}

            if (to == address(0)) {revert("InvalidRecipient");}

            if (
                msg.sender != from &&
                !isApprovedForAll[from][msg.sender] &&
                msg.sender != getApproved[amountOrId]
            ) {revert("unauthorized");}

            balanceOf[from] -= _getUnit();
            balanceOf[to] += _getUnit();
            _ownerOf[amountOrId] = to;
            delete getApproved[amountOrId];
            uint256 updatedId = _owned[from][_owned[from].length - 1];
            _owned[from][_ownedIndex[amountOrId]] = updatedId;
            _owned[from].pop();
            _ownedIndex[updatedId] = _ownedIndex[amountOrId];
            _owned[to].push(amountOrId);
            _ownedIndex[amountOrId] = _owned[to].length - 1;

            emit Transfer(from, to, amountOrId);
            emit ERC20Transfer(from, to, _getUnit());
        } else {
            uint256 allowed = allowance[from][msg.sender];
            if (allowed != type(uint256).max)
                allowance[from][msg.sender] = allowed - amountOrId;
            _transfer(from, to, amountOrId);
        }
    }
}
```

Fig. 11. A simplified pandora contract.

Code Coverage fitness metric has been a cornerstone fitness metric of fuzzing, which typically calculates the coverage of executed instructions, basic blocks or branches [16, 26, 32, 34, 38]. For example, Confuzzius [34] calculates the covered branches as the fitness metric for covering more branches. *Distance* fitness metric helps identify seeds that are more likely to reach unexplored areas, which commonly calculate the branch distance or code distance as distance metric [19, 26, 37, 39]. For example, sFuzz [26] selects the seed for each just-missed branch with the closest distance for mutation. In general, the above two fitness metrics mainly focus on reaching more code areas rather than state areas of smart contracts. *Vulnerability* fitness aims to increase the efficiency in detecting vulnerabilities, which always require prior knowledge of the vulnerabilities. For example, RLF [32] uses reinforcement learning to learn from vulnerable function-call sequences and guide the fuzzer to generate vulnerable function-call sequences.

Unlike the fuzzers above, SmartFuzz innovatively guides the fuzzer to explore smart contract states, making testing more comprehensive and helping to reach the *vulnerable state*.

7.2 Vulnerability Detection for Smart Contract

In addition to fuzzing technique, researchers also propose many security tools to detect smart contract vulnerabilities. Among them, we next discuss existing tools with mainstream techniques.

Symbolic execution is one of the most popular techniques being used to detect vulnerabilities, whose core idea is to execute the program with symbolic input and get the concrete input by constraint solving. In 2016, Luu et al. [24] proposed a novel symbolic executor, namely Oyente, which detects four classic vulnerabilities in smart contracts. *Formal Verification* identifies the correctness of *formal specification* in the formal model of the system [21, 28, 31]. For example, Kalra et al.[21] design a translator from Solidity to LLVM bitcode, based on which they leverage symbolic model checking to quickly verify contracts for safety. *Artificial Intelligence (AI)* is commonly used to detect vulnerabilities due to its high efficiency [13, 14, 17, 18]. In addition, AI is also widely used to improve the performance of security tools [30, 32, 38]. For example, Smartest [30] exploits the language model to learn from previous vulnerable transactions, based on which it guides symbolic execution to generate the transaction sequences to trigger vulnerabilities.

In general, the above-mentioned approaches detect smart contract vulnerabilities mainly relying on static properties, such as the code and its Abstract Syntax Tree (AST) and Control Flow Graph (CFG). Considering that smart contracts are *stateful* programs, a lack of information on dynamic states may limit the effectiveness of vulnerability detection, especially in state-relevant cases.

8 Conclusion

This paper proposes a state-aware fuzzer, SmartFuzz , to explore smart contract states and detect vulnerabilities, especially state-relevant cases. SmartFuzz is based on a state-aware seed scheduling that reserves and mutates the seed reaching new states for state exploration. We use two key insights to improve the efficacy of state exploration: (1) construction of the state space that consists of essential variables to compress the exploration state; (2) a novel strategy to identify new and interesting states for improving the efficiency of exploration. In experiments, SmartFuzz outperforms code-coverage-guided fuzzer in the field of both state coverage and code coverage. Meanwhile, the experimental results demonstrate that SmartFuzz can effectively reveal state-relevant vulnerabilities in smart contracts.

9 Data Availability

The tool and experimental data of this paper is released on website https://anonymous.4open.science/r/smartfuzz-5B4E. The artifact will be open-sourced after the paper is accepted.

References

[1] 2017. The Parity Wallet Hack. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[2] 2022. DefiLlama. https://defillama.com/chain/Ethereum

[3] 2022. ERC-20 Token Standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-20/

[4] 2022. Ethereum Yellow Paper: a formal specification of Ethereum, a programmable blockchain. https://ethereum.github.io/yellowpaper/paper.pdf

[5] 2022. Go. https://go.dev/doc/

[6] 2022. Slowmist Hacked Events. https://hacked.slowmist.io/

[7] 2022. Solidity Documentation. https://docs.soliditylang.org/en/v0.8.16/

[8] 2023. Etherscan. https://etherscan.io/

[9] 2024. DeFi Hacks Reproduce. https://github.com/SunWeb3Sec/DeFiHackLabs

[10] 2024. medusa. https://github.com/crytic/medusa

[11] 2024. PANDORA. https://phalcon.blocksec.com/explorer/tx/eth/0x7c5a909b45014e35ddb89697f6be38d08eff30e7c3d3d553033a6efc3b4

[12] 2024. Reth. https://github.com/paradigmxyz/reth

[13] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2023. When chatgpt meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:2309.05520* (2023).

[14] Jiachi Chen. 2020. Finding Ethereum Smart Contracts Security Issues by Comparing History Versions *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1382–1384. https://doi.org/10.1145/3324884.3418923

[15] Zhiyang Chen, Ye Liu, Sidi Mohamed Beillahi, Yi Li, and Fan Long. 2024. Demystifying Invariant Effectiveness for Securing Smart Contracts. *Proc. ACM Softw. Eng.* 1, FSE, Article 79 (jul 2024), 24 pages. https://doi.org/10.1145/3660786

[16] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 227–239. https://doi.org/10.1109/ASE51524.2021.9678888

[17] Zhipeng Gao, Lingxiao Jiang, Xin Xia, David Lo, and John C. Grundy. 2021. Checking Smart Contracts With Structural Code Embedding. *IEEE Transactions on Software Engineering* 47 (2021), 2874–2891.

[18] Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu. 2021. Hunting Vulnerable Smart Contracts via Graph Embedding Based Bytecode Matching. *IEEE Transactions on Information Forensics and Security* 16 (2021), 2144–2156. https://doi.org/10.1109/TIFS.2021.3050051

[19] Songyan Ji, Jian Dong, Junfu Qiu, Bowen Gu, Ye Wang, and Tongqi Wang. 2021. Increasing fuzz testing coverage for smart contracts with dynamic taint analysis. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 243–247.

[20] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[21] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[22] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1144–1156. https://doi.org/10.1145/3597926.3598124

[23] Ye Liu and Yi Li. 2023. InvCon: A Dynamic Invariant Detector for Ethereum Smart Contracts. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 160, 4 pages. https://doi.org/10.1145/3551349.3559539

[24] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[25] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[26] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[27] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. {ParmeSan}: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2289–2306.

[28] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 621–640.

[29] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. Ityfuzz: Snapshot-based fuzzer for smart contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–333.

[30] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. 1361–1378.

[31] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *2021 IEEE Symposium on Security and Privacy (SP)*. 555–571. https://doi.org/10.1109/SP40001.2021.00085

[32] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2023. Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-Guided Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE22)*. Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. https://doi.org/10.1145/3551349.3560429

[33] Jianzhong Su, Xingwei Lin, Zhiyuan Fang, Zhirong Zhu, Jiachi Chen, Zibin Zheng, Wei Lv, and Jiashui Wang. 2023. DeFiWarder: Protecting DeFi Apps from Token Leaking Vulnerabilities. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1664–1675. https://doi.org/10.1109/ASE56229.2023.00110

[34] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 103–119. https://doi.org/10.1109/EuroSP51992.2021.00018

[35] Sam Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William Knottenbelt. 2022. Sok: Decentralized finance (defi). In *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*. 30–46.

[36] Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou. 2024. Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[37] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3368089.3417064

[38] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xfuzz: Machine learning guided cross-contract fuzzing. *IEEE Transactions on Dependable and Secure Computing* 21, 2 (2022), 515–529.

[39] Wei Zhang. 2022. Beak: A directed hybrid fuzzer for smart contracts. *International Core Journal of Engineering* 8, 4 (2022), 480–498.

[40] Yuanhang Zhou, Jingxuan Sun, Fuchen Ma, Yuanliang Chen, Zhen Yan, and Yu Jiang. 2024. Stop Pulling my Rug: Exposing Rug Pull Risks in Crypto Token to Investors *(ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 228–239. https://doi.org/10.1145/3639477.3639722

[41] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009