Received: 18 August 2022 | Revised: 27 December 2022 | Accepted: 18 February 2023

DOI: 10.1002/smr.2557

RESEARCH ARTICLE - TECHNOLOGY

Journal of Software: Evolution and Process WILEY

sFuzz2.0: Storage-access pattern guided smart contract fuzzing

Haoyu Wang[1] | Zan Wang[1] | Shuang Liu[1] | Jun Sun[2] | Yingquan Zhao[1] | Yan Wan[1] | Tai D. Nguyen[2]

[1]College of Intelligence and Computing, Tianjin University, Tianjin, China

[2]School of Information Systems, Singapore Management University, Singapore

Correspondence
Shuang Liu, Tianjin University, Tianjin, China.
Email: shuang.liu@tju.edu.cn

Funding information
National Natural Science Foundation of China, Grant/Award Numbers: 61872263, U1836214, 61802275; Ministry of Industry and Information Technology Open Project, Grant/Award Number: NJ2022027

Abstract

Smart contracts are distributed self-enforcing programs which execute on top of blockchain networks. They have the potential to revolutionize many industries and have already been adopted for applications such as distributed finance and crowdfunding. Because smart contracts are immutable once they are deployed, it is important to identify and eliminate code vulnerabilities in smart contracts systematically. In this work, we propose sFuzz2.0, a storage-access-pattern guided adaptive fuzzer based on sFuzz. sFuzz2.0 is motivated by the fact that certain vulnerabilities only manifest in the presence of certain function call sequences (as well as particular arguments). Given that there are exponentially many function call sequences, sFuzz randomly generates sequences without guidance. As a result, the probability of discovering those vulnerabilities is negligible. sFuzz2.0 tackles the problem with two approaches, that is, by generating function call sequences that trigger different storage-access patterns passively (i.e., by prioritizing seeds which cover new patterns) or actively (i.e., by actively seeking out different patterns). The experiment results suggest that the passive strategy outperforms sFuzz by achieving better code coverage (i.e., 37.53%) and discovering more vulnerabilities (i.e., 20.49%).

KEYWORDS
coverage criteria, fuzzing, smart contract security

1 | INTRODUCTION

In recent years, both academia and industry have ignited a strong interest in smart contracts. A smart contract is a computer program which automatically executes, controls or documents legally relevant events and actions according to the terms of a contract or an agreement.[1] Built on top of a blockchain network, smart contracts allow traceable and irreversible transactions without the need of a trusted third party. They are believed to have the potential to reshape many industries, including legal agreements, finance, and supply chain. In fact, there have already been many real-world applications based on smart contracts, for example, the de facto smart contract platform Ethereum has more than 1.2 million daily transactions for a wide range of applications.

A smart contract is a piece of code written by programmers, and like any programmer-written code, a smart contract may contain code vulnerabilities. Because smart contracts manage valuable digital assets, the vulnerabilities may cause huge amounts of economic losses. Worse yet, because storage variables in smart contracts are stored on the blockchain network, once a vulnerable contract is deployed, it is almost impossible to patch the contract. For instance, a vulnerability in a smart contract called *TheDAO* was exploited by an attacker and, as a result, around four million *ethers* (i.e., Ethereum's digital currency, equivalent to about 50 million dollars at that time) were stolen.[2] After the hack, the Ethereum community conducted a hard fork, which is the only way to roll back the transactions and prevent the same attack from occurring, and caused much controversy.

The importance of systematically testing smart contracts has since been well recognized. Many testing engines have been developed,[3–11] including both fuzzing and symbolic approaches. For instance, Oyente[4] is designed to test each function in a smart contract separately based on symbolic execution,[6] that is, generating test cases by constraint solving each path in one function. ContractFuzzer[5] instead tests a smart contract by generating random inputs for each function in the contract. The symbolic execution tools[4,6,11–14] suffer from high constraint solving cost and potential false positives, while the fuzzing tools have low false positive rate but have limited deep code covering capabilities.

To improve the code covering capabilities and find out more vulnerabilities, we proposed sFuzz,[3] an efficient adaptive efficient fuzzers for Solidity smart contracts. Inspired by AFL,[15,16] sFuzz[3] applies feedback-guided fuzzing algorithm. Though AFL is proven to be efficient, its effectiveness reduces significantly in the presence of strict branching conditions. Thus, in addition to applying the seeds selecting strategy in AFL (i.e., select seeds which cover new branches), we apply an adaptive strategy which prioritizes the seeds according to a quantitative measure (i.e., distance) on how far a seed is from covering any just-missed branch. Specifically, we select seeds triggering smaller branch distance in each iteration until the corresponding branch is covered. Our empirical study[3] shows that the adaptive strategy is useful in increasing the coverage of the generated test suite, and sFuzz is on average more than two orders of magnitude faster than ContractFuzzer, covers more branches, and reveals many more vulnerabilities.

Though sFuzz[3] is effective and efficient. One technical challenge we must address is that sFuzz may be ineffective in generating specific function call sequences because sFuzz[3] randomly generates sequences. It is known that certain code vulnerabilities only manifest in specific function call sequences.[4,8] Consider, for instance, the contract shown in Figure 1, which implements a simple crowdfunding contract. To start a new crowdfunding, the owner, which is set in the base contract *owned*, should first call function *setUp* to open the crowdfunding and specify when the crowdfunding will be closed by calling function *check*(). Users can buy tokens after the crowdfunding is opened by calling function *buyToken*() with the ethers (i.e., msg.value) they want to invest. Users can withdraw their funding through function *refund* only after the owner checks and closes the crowdfunding. The *require* statement at Line 31 must be satisfied for successful refunding; otherwise, the function will be rolled back and the user will not receive their fund. However, a *gasless send* vulnerability[17] can be triggered if Line 34 is executed. That is, function *send*() calls the fallback function of *msg*.*sender* and forwards 2300 units as the gas limit. If the fallback function costs more than the gas limit, it

```
1  contract Crowdsale is owned, safeMath {
2      uint256 public end_date = now + 300 days;
3      uint256 public end = 0;
4      bool isClosed = false;
5      bool isSetup  = false;
6      mapping(address => uint256) fund;
7
8      function setUp(uint256 _end) onlyOwner {
9       if (!isSetup){
10           end    = _end;
11           isSetup   = true;
12           isClosed = false;
13        }
14     }
15
16     function check() onlyOwner {
17       require (isSetup);
18       if (block.number <= end) {
19         isClosed = false;
20       } else {
21         isClosed = true;
22       }
23     }
24
25     function buyToken() payable {
26       require(msg.value > 0 && isSetup);
27       fund[msg.sender] = add(fund[msg.sender], msg.value);
28     }
29
30     function refund(uint count) {
31       require (isClosed && fund[msg.sender] > 100);
32       uint256 ethRefund = fund[msg.sender];
33       fund[msg.sender] = 0;
34       msg.sender.send(ethRefund);
35     }
36
37     function finished() onlyOwner {
38         require(now > end_date);
39         owner.transfer(this.balance);
40     }
41 }
```

FIGURE 1 A motivation example.

rolls back and *send*() returns false. Such a returned value should be properly handled, yet it is not the case in this contract, which constitutes a *gasless send* vulnerability, as the user would not be refunded. The malicious deployer can take advantage of the investigators' misunderstanding (i.e., the statement *send*() transfers ethers correctly while the fallback of the receiver contract is complex) to make the ethers invested into the contract's account cannot be withdrawn, and after *end_date* (at Line 38), the deployer can transfer the balance to the owner to receive the wrongly invested ethers (at Line 39).

To expose this vulnerability, the functions must be called in a particular sequence with proper parameter values, that is, *setUp*(0), *check*(), *buyToken*(), and *refund*() so that the *require* condition at Line 31 are satisfied.

sFuzz generates function call sequences randomly, which means that such vulnerabilities are identified with fairly low probability. For a contract with k functions, the probability of generating a particular sequence of length m is $\frac{1}{k^m}$. The problem is further complicated as, in addition to the call sequence, specific parameters are often required to trigger the vulnerability. So given that there are exponentially many function call sequences with arguments, one problem is how to selectively generate specific function call sequences which are likely to expose vulnerabilities. Existing approaches address this problem in two ways. One is to limit the length of the call sequence to a small number, for example, no more than 2 in the case of Oyente,[4] and 3 in the case of sCompile.[6] The other (e.g., sFuzz[3]) is to leave it to chances,[3,5,7] that is, randomly generating sequences (with a larger bound on the length). While the former surely misses vulnerabilities which manifest only with long sequences (such as the one in Figure 1), the latter identifies those vulnerabilities with fairly low probability. A quick experiment shows ContractFuzzer, Oyente, and sFuzz (with a time budget of 2 min) all fail to identify the vulnerability in the contract shown in Figure 1. To address the above-mentioned issues, we propose a fuzzer for smart contracts called `sFuzz2.0`, which extends sFuzz by selectively generating function call sequences. Our idea is inspired by the recent development on testing concurrent programs.[18] That is, we propose to evaluate the relevance of a particular call sequence in terms of how storage variables are accessed. Note that storage variables in smart contracts are persistent states which are stored in the blockchain network and are always associated with smart contract vulnerabilities (because they are used to store balances). Intuitively, our approach aims to trigger as many different storage-access patterns (SAPs) as possible, which in turn improves `sFuzz2.0`'s chances of discovering vulnerabilities by improving the branch coverage. We study two different strategies to trigger different patterns, either actively (i.e., actively seeking out different patterns) or passively (i.e., randomly mutate call sequences and prioritize sequences which exhibit new patterns).

Our evaluation results on a set of 4603 smart contracts suggest that the passive strategy outperforms sFuzz, that is, covering 37.53% more branches and discovering 20.49% more vulnerabilities. In summary, we make the following contributions.

1. We propose to guide smart contract fuzzing based on a code coverage criteria with SAPs.
2. We develop two strategies, passive fuzzing and active fuzzing, to maximize SAPs.
3. We implement `sFuzz2.0` (available at sFuzz2.0[19]) on top of sFuzz and show its effectiveness with a large set of real-world smart contracts.

The remainder of the paper is organized as follows. We define our problem in Section 2 and illustrate how `sFuzz2.0` works step by step to address the problem. We review algorithms of sFuzz in Section 3 and introduce our extension in Section 4. Our approach is evaluated in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 | PRELIMINARY

In this section, we review relevant background and define our problem.

2.1 | Problem definition

A smart contract $\mathcal{S}$ typically has a number of instance variables, a constructor and multiple functions, some of which are public. It can be equivalently viewed in the form of a control flow graph (CFG) $\mathcal{S} = (N, i, E)$ where N is a finite set of control locations in the program; $i \in N$ is the initial control location, that is, the start of the contract; and $E \subseteq N \times C \times N$ is a set of labeled edges, each of which is of the form (n, c, n') where c is either a condition (for conditional branches like if-then-else or while-loops) or a command (i.e., an assignment).

Test cases. A test case for $\mathcal{S}$ is a pair (σ_0, Σ) where σ_0 is the configuration of the blockchain network and Σ is a sequence of transactions (i.e., function calls). The configuration σ_0 contains all information on the setup of the network which is relevant to the execution of the smart contract. Formally, σ_0 is a tuple (b, ts, SA, SB) where b is the current block number, ts is the current block timestamp, SA is a set of the addresses of the smart contracts (including the smart contract under test as well as other invoked contracts), and SB is a function which assigns an initial balance to each address. $\Sigma = \langle m_0(\overrightarrow{p_0}), m_1(\overrightarrow{p_1}), \ldots \rangle$ is a sequence of public function calls of the smart contract under test, each of which has an optional sequence of concrete input parameters $\overrightarrow{p_i}$. Note that m_0 must be a call of the constructor.

Given any nontrivial smart contract, there could be an enormous number of test cases. One essential problem to be addressed by a fuzzer is thus how to selectively generate test cases. This is often achieved by defining certain equivalent classes of test cases and generating test cases to cover as many classes as possible. For instance, in classic symbolic execution, test cases that cover the same program path are considered to be in the same equivalent class and the goal is to cover as many paths as possible (by generating one test case for each path if possible). In AFL, test cases that cover the same set of edges (assuming there is no hashing collision for simplicity) are considered equivalent. In sFuzz, test cases that cover the same edges are further distinguished by how far they are from covering certain uncovered branches (through a distance function[3]). In this work, we propose to further distinguish test cases based on SAPs, as they often lead to different behaviors.

The task of fuzzing a smart contract is thus to generate a set of test cases (a.k.a. test suite) according to certain testing criteria. The execution of a test case t traverses through a path in the CFG $\mathcal{S}$, which visits a set of nodes and edges. For simplicity, we assume that one test execution covers one unique path (i.e., there is no nondeterminism). Furthermore, a trace generated by t is a sequence of pairs of the form $\langle(\sigma_0, n_0), (\sigma_1, n_1), \ldots\rangle$ where $(n_0, n_1, \ldots)$ is the sequence of nodes visited by t and σ_i is the configuration at the time of visiting node n_i for all i.

Code coverage. Ideally, we aim to generate a test suite which reveals all vulnerabilities in the contract. However, as we do not know where the vulnerabilities are, we must instead aim to achieve something more measurable. In this work, our answer is to focus on code coverage, in particular, branch coverage. We remark that our approach can be extended to support different coverage at the cost of additional code instrumentation. A branch in $\mathcal{S}$ is covered by a test suite if and only if there is a test case t in the suite that visits the edge at least once. The branch coverage of a test suite is calculated as the percentage of the covered branches over the total number of branches. Note that identifying the total number of (feasible) branches statically in a smart contract is often infeasible for two reasons. First, some branches might be infeasible (i.e., there does not exist any test case that visits the branch) and knowing whether a branch is feasible or not is a hard problem. Second, EVM has a stack-based implementation which makes identifying all potentially feasible branches hard. *Our problem is thus reduced to generate a test suite which maximizes the number of covered branches.* To achieve maximum code coverage, one way is to generate a large test suite (e.g., through random test generation). However, in practice, we often have limited resources (in terms of time or the number of computer processes), and thus, our problem is refined as "to generate a test suite which maximizes the number of covered branches as efficiently as possible." Our solution to the problem is feedback-guided adaptive fuzzing.

Fuzzing is one of the most popular methods to create test cases.[20] A feedback-guided fuzzing system (a.k.a. fuzzer) takes a program under test and an initial test suite as input, monitors the execution of the test cases to obtain certain feedback, generates new test cases based on the existing ones in certain ways, and then repeats the process until a stopping criteria is satisfied. We present details of our feedback-guided adaptive fuzzing process in Section 3.

Oracles. The remaining problem is then how to tell whether a test case reveals a vulnerability. In this work, we adopt a set of oracles from previous approaches.[21,22] We remark identifying and defining smart contract vulnerabilities is a research problem orthogonal to ours. In the following, we briefly introduce the vulnerabilities that sFuzz are capable of detecting.

Gasless Send vulnerability. In Solidity, *send*() is a special transfer function that returns whether the transfer is successfully executed. By default, the gas limit of *send*() is 2300 units. The current transaction should stop and rollback immediately if *send*() returns false. The Gasless Send vulnerability is due to incorrect handling of the return value of function *send*(). We check Gasless Send vulnerability by detecting function calls with 2300 units as gas limit.

Exception Disorder vulnerability. In Solidity, the exception will propagate to the caller until low-level calls (e.g., *call*()), after that no side effect will be rolled back, which makes the caller contract be unaware of the errors if the return value is not properly handled. We check whether the exception is propagated to the root transaction to ensure all potential problematic transactions are reverted.

Reentrancy vulnerability. Function could be reentrant through fallback function. However, some developer may unaware of this feature (e.g., updated critical state before low-level calls), which may be exploited by malicious attackers (e.g., the DAO attack[23]). We detect reentrancy vulnerability by checking whether the function is called more than once in the current stack and if this reentrant function call is sending ether.

Timestamp Dependency vulnerability. In Solidity, timestamp can be manipulated by miners, and thus, it is unsafe to use timestamp as part of the condition that determines critical operations. We detect timestamp dependency by checking if the function invokes *TIMESTAMP* opcode and sends ethers at the same time.

Block Number Dependency vulnerability. Similar to timestamp, the block number could also be manipulated by miners. We detect block number dependency by checking if the function invokes *BLOCKNUMBER* and sends ethers at the same time.

Dangerous Delegate Call vulnerability. Function *delegatecall*() is a special call that the code deployed at the target contract is executed on the storage of the caller contract. Allowing arbitrary code executing on the storage of caller contract may be exploited. For example, the first round of parity wallet[24] is attributed to dangerous delegate call. We detect dangerous delegate call vulnerability by checking invocation of the *DELEGATECALL* opcode.

Freezing Ether vulnerability. The ethers in the contract will be frozen when the contract relies merely on delegate call to transfer to the callee and the callee is self-destructed. The second round of parity wallet[24] is attributed to freezing ether. We detect freezing ether vulnerability by checking if the contract can receive ether and invoke delegate call during execution while there is no other ways to transfer ethers.

Integer Overflow/Underflow vulnerability. The arithmetic of operation should be checked to prevent integer overflow and underflow; otherwise, they may produce unexpected result and affect state of the contract. We detect integer bugs by comparing the execution result and the expected result.

2.2 | SAPs

Different from the existing fuzzers, `sFuzz2.0` is designed based on observation that whether a certain branch (and consequently vulnerability) is covered is often correlated with how storage variables (i.e., permanent status stored on the blockchain, such as account balances) are accessed. For instance, consider two call sequences:

$$\begin{aligned} \pi_1 &= \langle setUp(0), check(), refund(), buyToken() \rangle, \\ \pi_2 &= \langle setUp(0), check(), buyToken(), refund() \rangle, \end{aligned}$$

which differ only by the order of the last two function calls. Assume that $msg.value$ for $refund()$ is greater than 0 and $block.number$ is greater than 0. π_1 and π_2 cover the same branches, that is, the branches at Lines 9, 17, 20, and 26. AFL and sFuzz thus consider them the same and randomly drop one of them. A closer look however reveals that fuzzing π_2 (by mutating the value of parameter $msg.value$ for function $buyToken$) is more likely to cover the branch at Line 31 and trigger the vulnerability at Line 34. Intuitively, this is because π_2 differs from π_1 on how $fund[msg.sender]$ is accessed, that is, $buyToken$ writes $fund[msg.sender]$ at Line 27 and then $refund$ reads $fund[msg.sender]$ at Line 31, and as a result, Line 31 reads a value greater than 100, and thus, the require condition is satisfied. That is, π_2 and π_1 are different in terms of storage variables access patterns.

We argue that the order of storage access matters. In this example, function $buyToken$ writes $fund[msg.sender]$ (at Line 27) and function $refund$ reads $fund[msg.sender]$ as part of the condition (at Line 31), the order of the two storage variable access events is critical to trigger the bug. If the write access (at Line 27) is executed before read (at Line 31), the read access can obtain different values through changing the $msg.value$ and there is a higher chance to cover the branch (at Line 31) if we perform further value mutation.

We thus introduce the concept of SAPs and use it to optimize test case generating through fuzzing. We remark that SAP is inspired by the work in Wang et al[18] and Park et al,[25] which shows memory-access patterns are often associated with concurrent bugs. Intuitively, an SAP captures how storage variables are accessed by multiple functions in a trace. An SAP consists of a sequence of storage-access events. A storage-access event is a record of storage access by functions, which is defined as follows:

Definition 1 Storage-access event. A storage-access event is a tuple of the form (f,x,R) or (f,x,W) where f is a function that reads/writes the storage variable, x is a storage variable, and R/W represents reading/writing access, respectively.

Intuitively, (f,x,R) is the event of function f reads variable x and (f,x,W) is the event of function f writes variable x. With the definition of storage-access event, the SAP is defined as:

Definition 2 SAP. A SAP is a consecutive sequence of two to four storage-access events, as shown in Table 1.

The 17 patterns intuitively enumerates all possible ways of accessing one or two storage variables by different functions. We remark that more complicated patterns (which involve more than four functions or two storage variables) can be separated into a combination of multiple patterns in this table. For example, a len-3 P4 pattern is formed by one len-2 P1 Pattern and another len-2 P2 pattern by merging the two identical events (f,x,R) into one. For the detail of how len-2 patterns form len-3 and len-4 patterns, please refer to Algorithm 5. Given a test case π, we can systematically obtain the set of SAPs covered by t, denoted as $patterns(t)$. Note that the same read/write sequences on a contract variable x by different functions are considered different patterns.

Our goal is to cover a variety of different SAPs during fuzzing process, which we believe would contribute to improving branch coverage during fuzzing. This is evidenced by the example above as well as the empirical study results in Section 5. In other words, we propose an SAP-Coverage to complement existing coverage such as branch coverage. Given a test suite, computing its SAP-Coverage is highly nontrivial as we must know how many SAPs are feasible in general, which in turn requires us to identify infeasible program paths. Fortunately, for the purpose of fuzzing, all we need is to try generating new SAPs based on a given set of seed test cases.

TABLE 1 Seventeen storage-access patterns.

ID	Storage-access pattern
P1	$(f_1,x,R),(f_2,x,W)$
P2	$(f_1,x,W),(f_2,x,R)$
P3	$(f_1,x,W),(f_2,x,W)$
P4	$(f_1,x,R),(f_2,x,W),(f_3,x,R)$
P5	$(f_1,x,W),(f_2,x,W),(f_3,x,R)$
P6	$(f_1,x,W),(f_2,x,R),(f_3,x,W)$
P7	$(f_1,x,R),(f_2,x,W),(f_3,x,W)$
P8	$(f_1,x,W),(f_2,x,W),(f_3,x,W)$
P9	$(f_1,x,W),(f_2,x,W),(f_3,y,W),(f_4,y,W)$
P10	$(f_1,x,W),(f_2,y,W),(f_3,x,W),(f_4,y,W)$
P11	$(f_1,x,W),(f_2,y,W),(f_3,y,W),(f_4,x,W)$
P12	$(f_1,x,W),(f_2,x,R),(f_3,y,R),(f_4,y,W)$
P13	$(f_1,x,W),(f_2,y,R),(f_3,x,R),(f_4,y,W)$
P14	$(f_1,x,R),(f_2,x,W),(f_3,y,W),(f_4,y,R)$
P15	$(f_1,x,R),(f_2,y,W),(f_3,x,W),(f_4,y,R)$
P16	$(f_1,x,R),(f_2,y,W),(f_3,y,R),(f_4,x,W)$
P17	$(f_1,x,W),(f_2,y,R),(f_3,y,W),(f_4,x,R)$

2.3 | An illustrative example

In the following, we illustrate how our approach works through the example shown in Figure 1.

Let us first examine how sFuzz[3] works. sFuzz's overall design follows that of AFL, that is, it keeps mutating the test inputs (i.e., parameter values and system configuration values), and selects those test cases which cover new branches or obtain a closer distance to those just-missed branches as seeds for further mutation. The strategy of sFuzz is shown to be effective in obtaining high branch coverage and revealing vulnerabilities, as shown in Nguyen et al.[3] However, sFuzz fails to satisfy the branch conditions at Line 31 after fuzzing for 2 min. We observe that it is because sFuzz initially generates the call sequence $\langle refund, buyToken, check, setUp\rangle$ (according to the order of the functions in ABI) and fails to generate the call sequence $\langle\ setUp, check, buyToken, refund\ \rangle$. Note that the *require* condition $fund[msg.sender] > 100$ (at Line 12) is satisfied if and only if functions are invoked in the above order and $msg.value$ for function *buyToken* is set to be greater than 100.

`sFuzz2.0` extends sFuzz by using SAP-Coverage to guide the fuzzing process. Given the contract in Figure 1, `sFuzz2.0` generates the same call sequence $\langle check, setUp, refund, buyToken\rangle$ initially. After the test case is executed, `sFuzz2.0` collects branch information related to branch coverage (i.e., for each branch, whether it is covered and if not, how far is the branch from being covered) and extracts SAPs from the test case. Next, `sFuzz2.0` evolves the test suite, by not only applying mutations on the function parameters but also mutations that aim to cover new SAP, that is, one that writes $fund[msg.sender]$ before reading it. Note that there may be many storage-access events in a single function. Because a function is executed without interruption in Ethereum, the order of the event inside a function is fixed. Thus, we mutate the function call sequence to cover new SAPs. `sFuzz2.0` supports two strategies for triggering new SAP. The passive strategy randomly adds/removes/swaps function calls, for example, functions *buyToken* and *refund* are swapped to generate a new seed $\langle check, setUp, buyToken, refund\rangle$, which covers new SAPs and is kept for further mutation. Subsequently, function *check* and *setUp* are swapped, resulting in the vulnerability-triggering function sequence $\langle setUp, check, buyToken, refund\rangle$ that cover the key SAP (i.e., $E_{10}E_{12}$ in Figure 2B), and the seed is kept for further mutation. Finally, when $msg.value$ is mutated to be greater than 100, the condition at Line 31 is satisfied and the vulnerability is revealed. Note that keeping the test case which triggered the new SAP is the key to covering the new branch and triggering the corresponding vulnerability. sFuzz may also generate the above sequence, but sFuzz does not do further argument mutation on this sequence, and thus misses the chance to reveal the vulnerability. The active strategy first identifies potential storage-access events through dynamic analysis, for example, recording the arguments and related configuration values. For instance, when *refund* reads (at Line 31) and *buyToken* writes (at Line 27), the value of *end* (for *setup*) and $msg.value$ (for *buyToken*) are recorded. Afterwards, `sFuzz2.0` selectively introduces/removes a function call in the sequence in order to trigger new patterns. In this example, `sFuzz2.0` generates the call sequence $\langle check, setUp, buyToken, refund\rangle$ and detects the vulnerability in 7 s.

GID	Event
E_1	$(setUp, isSetup, R)$
E_2	$(setUp, end, W)$
E_3	$(setUp, isSetup, W)$
E_4	$(setUp, isClosed, W)$
E_5	$(check, isSetup, R)$
E_6	$(check, end, R)$
E_7	$(check, isClosed, W)$
E_8	$(buyToken, isSetUp, R)$
E_9	$(buyToken, fund, R)$
E_{10}	$(buyToken, fund, W)$
E_{11}	$(refund, isClosed, R)$
E_{12}	$(refund, fund, R)$
E_{13}	$(refund, fund, W)$

(A) storage access event sequence

Events	Pattern
E_2E_6	$(setUp, end, W), (check, end, R)$
E_3E_5	$(setUp, isSetup, W), (check, isSetup, R)$
E_3E_8	$(setUp, isSetup, W), (buyToken, isSetUp, R)$
E_4E_7	$(setUp, isClosed, W), (check, isClosed, W)$
E_7E_{11}	$(check, isClosed, W), (refund, isClosed, R)$
E_9E_{13}	$(buyToken, fund, R), (refund, fund, W)$
$E_{10}E_{12}$	$(buyToken, fund, W), (refund, fund, R))$
$E_{10}E_{13}$	$(buyToken, fund, W), (refund, fund, W)$
$E_4E_7E_{11}$	$(setUp, isClosed, W), (check, isClosed, W), (refund, isClosed, R)$

(B) patterns

FIGURE 2 Examples of storage-access patterns extracted from test case ⟨*setUp*,*check*,*buyToken*,*refund*⟩.

3 | sFUZZ FUZZING ALGORITHM

Algorithm 1 Fuzzing algorithm

```
1: let suite be an empty test suite
2: let seeds:= initPopulation()
3: for each branch br do
4:     let min(br) be $+\infty$
5: end for
6: while not time out do
7:     let seeds := fitToSurvive(seeds, suite)
8:     let seeds := crossoverMutation(seeds)
9: end while
10: return suite
```

In this section, we briefly review the algorithm in sFuzz.[3] Note that sFuzz and `sFuzz2.0` are both grey-box feedback-guided fuzzing, they share the same main Algorithm 1, and `sFuzz2.0` differs sFuzz on *fitness criteria and mutation operators*.

Grey-box feedback-guided fuzzing. As shown in Algorithm 1, sFuzz[3] employs a genetic algorithm to evolve the test suite in order to cover as many branches as possible. Variable *suite* is the test suite to be generated and is initially set to be empty. Variable *seeds* is a set of seed test cases, based on which new test cases are generated and is initiated with function *initPopulation*(). For each branch *br* in the contract, *min*(*br*) maintains the smallest distance from covering this branch, which is initialized to be $+\infty$ for each branch.

The loop from Lines 6–9 iteratively evolves the test suite and stops only when it times out. Function *fitToSurvive*(*seeds*,*suite*) executes each seed (i.e., test case) in *seeds*, collects branch information, and returns a set of selected seeds which are likely to generate new test cases that cover new branches (Line 7). Function *crossoverMutation*() generates new test cases based on the selected seeds through crossover and mutation (Line 8).

Generating initial population. Function *initPopulation*() generates test cases (in the form of a bit vector) as the initial seeds. A test case consists of both the blockchain configuration and a sequence of function calls. Blockchain configuration is a set of unsigned integers and is initialized with random values. A function contains two parts, that is, the function selector and a set of parameters. Recall that the function selector is the first 4 bytes of the function signature hash value. The parameter contains fixed-length type or dynamic-length type values. For dynamic-length type values, a length in the range of [0,255] is generated first and then a random value of the given length is generated. For fixed-length type values, a random value of the given length is generated.

The fitness function. A fitness function is used to select seeds which are likely to cover new branches through crossover or mutation. It selects those promising seeds from the input *seeds*. Variable *newSeeds* is initially an empty set of test cases. Seeds which are "fit to survive" are inserted into this set.

sFuzz applies two fitness criteria, one is adopted from AFL[15,16] which considers *seeds* covers new branches as *newSeeds*. Another is a lightweight objective function where the objective is shrinking the distance from covering just-missed branches. For a branch *br* labeled with condition *c*, which can be any of $false, a == b, a! = b, a> = b, a>b, a< = b, a<b$. The $distance(br)$ is defined as follows.

$$dist(seed, br) = \begin{cases} K & \text{if } c \text{ is false or } a! = b, \\ |a-b| + K & Otherwise, \end{cases}$$

where *K* is a constant which represents the minimum distance. Intuitively, $distance(t, br_n)$ is defined such that the closer the branch is from being covered, the smaller the resultant value is. In `sFuzz2.0`, the *K* is set to be 1 as the same in sFuzz.

Algorithm 2 fitToSurvive(*seeds*, *suite*)

```
1: let newSeeds be an empty set of test cases
2: for each seed in seeds do
3:     execute the seed to collect information
4:     if seed covers a new branch then
5:         add seed into newSeeds and suite
6:     end if
7:     for each uncovered branches br do
8:         if $dist(seed, br) < min(br)$ then
9:             let $min(br)$ be $dist(seed, br)$
10:            add seed into newSeeds
11:        end if
12:    end for
13:    if seed covers a new pattern then
14:        add seed into newSeeds
15:        updatePatterns(seed)
16:    end if
17:    updateRW(seed); //only needed for active fuzzing
18: end for
19: return newSeeds
```

With the above, Algorithm 2 shows the details on how sFuzz selects seeds. It examines every test case in *seeds*. At Line 3, the selected *seed* is executed to identify the covered branches and SAPs. If the *seed* covers a new branch, it is added into *newSeeds* and *suite* (Line 5). Then, we update the minimum distance of each seed with just-missed branches according to sFuzz[3] (Lines 7–12). For each just-missed branch *br*, we retain the test case which achieves the minimal distance as a seed. Note that the third fitness criteria (Lines 13–17) is only used in `sFuzz2.0`, which we will introduce later in Section 4.

Crossover and mutation. Recall that a seed consists of a network configuration and a sequence of function calls with concrete arguments. $crossoverMutation()$ generates new test cases through crossover and mutation. Crossover works by "mixing" two seeds to generate new test cases. Mutation works by modifying either the network configuration, the function parameters, or the function call sequence.

Algorithm 3 mutation (*seeds*)

```
1: let new be an empty set of test cases
2: for each seed in seeds do
3:     add $AFLMutate(seed)$ into new
4:     add into new a seed by removing a function f from seed
5:     add into new a seed by swapping two functions in seed
6:     add into new a seed by inserting a function f in seed
7: end for
8: return new
```

Algorithm 3 shows the details of mutation. For each given seed, sFuzz first applies the AFL mutation operators one by one, which mutate on the bit/byte level. For those values which have the fixed-length type (e.g., *uint*32), the mutation operators are systematically applied to generate new values. Note that the value of type *address* is handled separately by a special operator as there are special format requirements. Each address has 32 bytes, in which the first 12 bytes contain the balance of the address and the last 20 bytes contain the address value. Those values that have dynamic-sized types (e.g., *array*) consist of length and content. To mutate the value of those types, sFuzz generates a length between 0 and 255 first and then pads (prunes) the missing (extra) bits if the new length is more (less) than the current one. To mutate the call sequence, sFuzz randomly adds/removes a function call or swap two randomly selected function calls to change the call sequence.

4 | SAP-COVERAGE-GUIDED FUZZING

In this section, we introduce our extension to sFuzz. At top level, `sFuzz2.0` and sFuzz share Algorithm 1, and `sFuzz2.0` complements sFuzz with SAPs. In particular, we propose two algorithms to guide the fuzzing process. The first algorithm is the passive strategy, that is, we extend the fuzzing algorithm in `sFuzz` with a seed selection procedure and select test cases which exhibit new SAP as seeds. We also propose an active strategy, which actively mutates a seed test case in a way that maximizes the likelihood of generating a new SAP.

4.1 | Passive fuzzing

Passive fuzzing is a simple approach which aims to minimize the computational overhead.

To effectively generate coverage-improving function sequence orders, we selects those seeds which cover any new SAP (i.e., one that is not in visited patterns). Given a seed that contains a sequence of storage-access events, we can systematically extract the SAPs as follows. `sFuzz2.0` instruments two instructions, that is, *SLOAD* and *SSTORE*, in EVM for collecting storage-access events. *SLOAD* reads a value v at a position p and *SSTORE* write a value v to a position p in storage. Given the test case, for each execution of *SLOAD* or *SSTORE*, we record the relevant information in the form of

$$E_{ID} : (selector, p, A),$$

where E_{ID} is a global storage-access event id; *selector* is the first four bytes of the Keccack-256 hash of the function signature, representing the function to be called; p is the current storage-access position which acts as a variable identifier; and A is R (and W) if the opcode is *SLOAD* (and *SSTORE*). As a result, for each test case, we obtain a sequence of indexed storage-access events.

For the contract shown in Figure 1, given the test case that consists of the call sequence

$$\langle setUp(0), check(), buyToken(), refund() \rangle,$$

the sequence of storage-access events are shown in Figure 2A on the left. E_1 is the event of the first function *setUp* reads the value of contract variable *isSetup* at Line 9, which contains a *SLOAD* opcode. Before the opcode is executed, the position of variable *openDate* is pushed onto the call stack, and the selector of the function is *keccak("setUp(uint256)")*. Hereafter, for simplicity, we use the function name as the selector (when there is no ambiguity).

For every $E_s : (f, x, A)$ in the sequence, `sFuzz2.0` aims to identify a Length 2 pattern starting from E_{id} and search through the subsequent events until a matching event according to the pattern is identified (if there is any). For instance, if the event E_s is a read access, a P1 pattern is matched; otherwise, we aim to match one P2 pattern. Note that the first event and the second event must access the same variable x. For example, given the trace shown in Figure 2A, for event $E_9 : (buyToken, fund, R)$, event $E_{13} : (refund, fund, W)$ is the first event that writes to the variable *balance*. These two events thus form a P1 pattern.

For each pair of patterns of Length 2, we check whether they can form a new pattern (of Length 3 or 4). If the second event in the first pattern is the same as the first event in the second pattern, a Length 3 pattern is formed; otherwise, a Length 4 pattern is formed. For the example in Figure 2B, the Length 2 pattern composed of E_4 and E_7 and that composed of E_7 and E_{11} form a Length 3 pattern. Two Length 2 patterns (i.e., one composed of E_2 and E_6 and the other composed of E_9 and E_{13}) form a Length 4 pattern. The Length 4 patterns are omitted in the table for the sake of space.

With the above, Algorithm 2 shows the details on how `sFuzz2.0` selects seeds. For each *seed* in *seeds*, in addition to the branch fitness and distance fitness (Lines 4–12), if the *seed* covers a new pattern, it is also added into *newSeeds* (Lines 13–17). Furthermore, we call *updatePatterns*(*seed*) (Line 18) to update the visited patterns. Note that function *updateRW*(*seed*) is called only when active fuzzing is chosen, as we explain later.

Mutation operators. In the passive strategy, we adopt all mutation operators from sFuzz as previously shown in Algorithm 3, with the hope that altering the call sequence (i.e., randomly swap, add, and remove functions) would alter the order of storage-access events and thus generate SAPs. Note that there is no guarantee that the altered call sequence will generate new SAPs. However, thanks to the seed selection algorithm (i.e., Algorithm 2), whenever a mutated call sequence covers a new SAP, it will be selected to generate new test cases. If it happens that the new test cases are able to obtain smaller distances, they will be further mutated to cover the branches gradually. Although this strategy is simple in nature, it has the advantage of introducing little overhead, which is fairly important for fuzzing.

4.2 | Active fuzzing

To complement the passive strategy, we further propose an active fuzzing strategy, which mutates a call sequence in specific ways so that it is likely to cover new patterns. The active fuzzing strategy is shown in Algorithm 4. Different from the passive fuzzing algorithm, the active fuzzing algorithm actively attempts to increase SAP-Coverage by mutating the seeds in specific ways.

Identifying potential patterns. We start with identifying potential SAP at Line 2. That is, by knowing how storage variables are accessed by each function, we can avoid those call sequences which are unlikely to generate uncovered SAPs. Recall that a function *updateRW* is called in Algorithm 2 when active fuzzing is applied. Its goal is to maintain two mappings *read* and *write*, from storage variables to functions. Intuitively, given a storage variable x, *read*[x] (and *write*[x]) is the set of functions which read (and write) x. We remark the two mappings could be identified through static analysis, although we would suffer from false positives. In this work, we instead do it through dynamic analysis. Based on the information, we can then determine where to insert or remove a function call in the given test case so that the resultant call sequence is likely to cover a new SAP.

Algorithm 5 shows the details on how potential patterns are identified. The idea is to enumerate all possible combinations of one or two storage variables, and functions based on the *read* and *write* mapping, according to the pattern template in Table 1. Note that Algorithm 5 can be further optimized so that all combinations are enumerated only updated for those storage variables and functions that are updated in the latest call of *updateRW*, that is, the potential patterns are identified incrementally.

Algorithm 4 activeMutation(*seeds*, *visitedPatterns*)

```
1:  let new be an empty set of test cases
2:  let possiblePatterns := identifyPotentialPatterns()
3:  let uncoveredPatterns := possiblePatterns − visitedPatterns
4:  for each $seed = \langle f_1(\bar{x}_1), f_2(\bar{x}_2), \cdots f_n(\bar{x}_n) \rangle$ in seeds do
5:      add AFLMutate(seed) into new
6:      for each function call $f_i(\bar{x}_i)$ in seed do
7:          let pe be $\langle f_1(\bar{x}_1), f_2(\bar{x}_2), \cdots f_i(\bar{x}_i) \rangle$
8:          for each pt in uncoveredPatterns do
9:              let $(f, x, *) = next(pe, pt)$ be the next event in pt after all events in pe
10:             add into new a seed by removing $f_{i+1}$ if $f_{i+2}$ is $f$
11:             add into new a seed by swapping $f_{i+1}$ with any $f$ in seed
12:             add into new a seed by inserting a function call $f$ in seed
13:         end for
14:     end for
15: end for
16: return new
```

Once we identify the potential patterns (Line 2), we calculate *uncoveredPatterns* (Line 3) by removing those SAPs which have already been covered. Afterwards, for each seed, we first apply AFL mutation operators to generate new seeds. For each function call $f_i(\overline{x_i})$ in the given sequence, let *pe* be a prefix of the function sequence ending with $f_i(\overline{x_i})$. The active mutation process (Lines 8 to 13) iterates on each pattern *pt* in *uncoveredPatterns* to actively mutate *seed* to generate new test cases which are likely to cover the target pattern *pt*. Function *next*(*pe*,*pt*) returns an event (f,x,A) in *pt*, that is, given a function sequence *pe*, we match the event sequence prefix of *pt* and return the event after the prefix in *pt* if it exists. Lines 10 to 12 add/remove/swap a function call to generate new seeds which attempt to cover a longer prefix of *pt*.

Algorithm 5 identifyPotentialPatterns()

```
1:  let $patterns$ be an empty set of patterns
2:  for each $x$ in contract do
3:      for each $f \in read[x]$ and each $f' \in write[x]$ do
4:          add $(f,x,R),(f',x,W)$ into $patterns$
5:          add $(f',x,W),(f,x,R)$ into $patterns$
6:      end for
7:      for each $f \in write[x]$ and each $f' \in write[x]$ do
8:          add $(f,x,W),(f',x,W)$ into $patterns$
9:      end for
10:     for each $f' \in read[x]$ and $f',f'' \in write[x]$ do
11:         add $(f',x,W),(f,x,R),(f'',x,W)$ into $patterns$
12:         add $(f,x,R)(f',x,W)(f'',x,W)$ into $patterns$
13:     end for
14:     for each $f,f'' \in read[x]$ and $f' \in write[x]$ do
15:         add $(f,x,R),(f',x,W),(f'',x,R)$ into $patterns$
16:     end for
17:     for each $f,f' \in write[x]$ and $f'' \in read[x]$ do
18:         add $(f,x,W)(f',x,W)(f'',x,R)$ into $patterns$
19:     end for
20:     for each $f,f',f'' \in write[x]$ do
21:         add $(f,x,W),(f',x,W)(f'',x,W)$ into $patterns$
22:     end for
23: end for
24: for each $x$ in contract and $y$ in contract do
25:     for each $f,f' \in write[x]$ and $f'',f''' \in write[y]$ do
26:         add $(f,x,W),(f',x,W),(f'',y,W),(f''',y,W)$ into $patterns$
27:         add $(f,x,W),(f'',y,W),(f',x,W),(f''',y,W)$ into $patterns$
28:         add $(f,x,W),(f'',y,W),(f''',y,W),(f',x,W)$ into $patterns$
29:     end for
30:     for each $f \in (write[x])$ and $f' \in (read[x])$ and $f'' \in read[y])$ and $f''' \in write[y]$ do
31:         add $(f,x,W),(f',x,R),(f'',y,R),(f''',y,W)$ into $patterns$
32:         add $(f,x,W),(f'',y,R),(f',x,R),(f''',y,W)$ into $patterns$
33:         add $(f,x,W),(f'',y,R),(f''',y,W),(f',x,R)$ into $patterns$
34:         add $(f',x,R),(f,x,W),(f''',y,W),(f'',y,R)$ into $patterns$
35:         add $(f',x,R),(f''',y,W),(f,x,W),(f'',y,R)$ into $patterns$
36:         add $(f',x,R),(f''',y,W),(f'',y,R),(f,x,W)$ into $patterns$
37:     end for
38:     return $patterns$
```

For instance, assume pattern *pt* is

$$(f1,x,W),(f2,x,R),(f3,x,W).$$

Given $pe=(f1,x,R),(f2,x,W)$, the prefix of *pt* that is matched is $(f1,x,W)$, $next(pe,pt)$ is the event after the pattern prefix, which is $(f2,x,R)$. As a result, function *f*2 with concrete parameters is inserted after the prefix at Line 10, aiming to cover the longer pattern prefix $(f1,x,W),(f2,x,R)$. Similarly, we remove a function call or swap two function calls at Lines 11 and 12 if it is likely to extend the matched prefix of the pattern. Note that there is no guarantee that *f*2 writes variable *x* even if the parameter is set to be the same with which *f*2 wrote to *x* in a previous test case, because the context might have changed. One complication that is worth discussing here is regarding those dynamic-sized storage variables, such as *arrays* and *mappings*. In Ethereum, these variables are "flattened" and stored in a huge map which maps positions on the global storage to values. Given a position, it is in general difficult to determine which source-level variable it belongs to. In our approach, we determine whether a position *p* stores the value of a dynamic or fixed-sized variable by the magnitude of *p*. Contract variables are stored continuously starting from Position 0 and the fixed-sized variables occupy fixed numbers of slots. For instance, one slot (i.e., 32 bytes) is allocated for a variable of type *uint*. For dynamic-sized variables like mappings and dynamic arrays, due to their unpredictable size, they take 32 bytes (one slot) to store meta-information (e.g., length of an array), and the elements they contain are stored starting at a different storage slot which is computed using a Keccak-256 hash. Thus, by calculating the number of total slots occupied by fixed-sized variables and the meta-information of dynamic-sized variables, we are able to identify all those fixed-sized variable values systematically. For the remaining positions on the global storage which are accessed by the contract, we are aware that they are associated with certain dynamic-sized variables, although it is highly nontrivial to know exactly which one. To reduce the number of patterns and speed up the fuzzing process, for patterns regarding those dynamic-sized variables, we only sample a threshold number of positions on the accessed global storage (i.e., 1/10 or an upper bound of 200 which is smaller). This is justified as covering a pattern that accesses the same array at a different index from a covered pattern is perhaps not as valuable as covering a pattern on different variables.

5 | IMPLEMENTATION AND EVALUATION

`sFuzz2.0` is implemented in C++ and has been integrated into sFuzz and is publicly available.[19] It is built on the Ethereum client aleth, which is used to simulate the Solidity execution environment and record storage-access events at the bytecode level. `sFuzz2.0` extends the sFuzz framework[3] with both the passive and the active strategy. `sFuzz2.0` supports all types of vulnerabilities defined in Nguyen et al.[3]

To evaluate the effectiveness of `sFuzz2.0`, we designed a set of experiments to answer the following research questions (RQ):

- RQ1: Does `sFuzz2.0` achieve higher code coverage than the state-of-the-art approach?
- RQ2: Is `sFuzz2.0` effective in achieving high SAP-Coverage?
- RQ3: Is the overhead of the active strategy in `sFuzz2.0` justified and is the passive and active strategy complementary?
- RQ4: Is `sFuzz2.0` able to find new vulnerabilities?

Experiment setup. To evaluate the smart contract testing tools, several works have proposed various benchmark datasets. SmartBugs is the first smart contract benchmark. It contains hundreds of vulnerable contracts and collects de-duplicated contracts from Etherscan. SolidiFI[26] constructs a buggy dataset by injecting thousands of distinct bugs in 50 contracts. Ren et al[27] complement the above benchmarks with vulnerable contracts crawled from the CVE dataset. In the following, we focus on a selected subset of contracts from the unlabeled real-world contracts (UR) dataset since it is more comprehensive. Our experiment subjects are gathered from the existing study. Ren et al[27] collect 45,622 contracts from the Ethereum blockchain that have Solidity source code available in Etherscan. Due to the limited computing resources that we have, we randomly select 4603 (around 10% of the dataset) of them as our test subjects. The LoC of these contracts ranges from hundreds to thousands, we illustrate the distribution of LoC of these contracts in Figure 3. As can be seen, most of the contracts (about 90%) are relatively complicated (i.e., with LoC greater than 100).

For a baseline comparison, we compare the two strategies of `sFuzz2.0`, that is, $\texttt{sFuzz2.0}_{\textit{passive}}$ and $\texttt{sFuzz2.0}_{\textit{active}}$, with sFuzz and ContractFuzzer.[5] Note that there are additional recent testing engines for smart contracts, such as Clairvoyance[12] and Echidna.[7,28] They cannot be compared as Clairvoyance is not available and Echidna requires user-defined assertions, which are nontrivial to generate automatically or even manually (e.g., it is not clear what the assertion should be for the TheDAO vulnerability[2]). All experiment results reported below are obtained on a machine with two octa-core CPUs Intel(R) Xeon(R) E5-2620 v4 @ 2.10 GHz and 94 GB memory, running Ubuntu 18.04.2 LTS. The timeout is set to be 10 minutes for fuzzing each contract. To minimize the impact of randomness, each run is repeated three times independently, and we report the average result.

RQ1: Does `sFuzz2.0` *achieve higher branch coverage than the state-of-the-art approach?*

To answer this question, we systematically apply `sFuzz2.0`, ContractFuzzer, and sFuzz to every contract for 10 min and measure the number of distinct branches covered. Figure 4 illustrates the distribution of branch number covered by ContractFuzzer, sFuzz, and two strategy of `sFuzz2.0`. The vertical axis is the number of branches covered. It can be seen the two strategies of `sFuzz2.0` outperform ContractFuzzer (about 2× branches covered) and sFuzz (about 1.3× branches covered) in terms of branch coverage. The passive strategy performs slightly better than the active strategy.

We summarizes the comparison between $\texttt{sFuzz2.0}_{\textit{passive}}$ versus sFuzz, $\texttt{sFuzz2.0}_{\textit{active}}$ versus sFuzz, and $\texttt{sFuzz2.0}_{\textit{passive}}$ versus $\texttt{sFuzz2.0}_{\textit{active}}$ on every single contact in Figure 5. The vertical axis is the difference between the number of branches covered by the two approaches, that is, the number of branches covered by the first approach (e.g., $\texttt{sFuzz2.0}_{\textit{passive}}$ for Figure 5A) minus that of the second approach

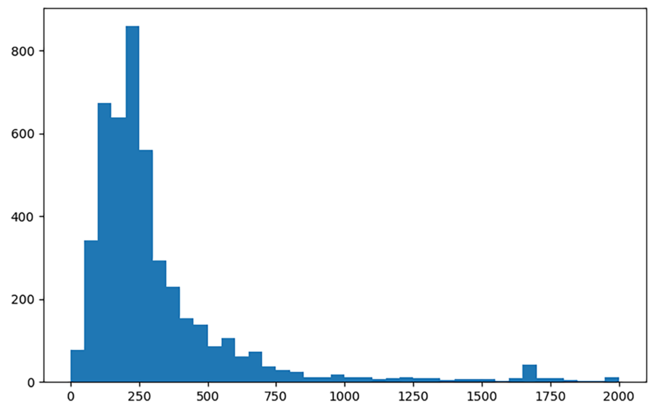


FIGURE 3 LoC distribution of benchmark contracts.

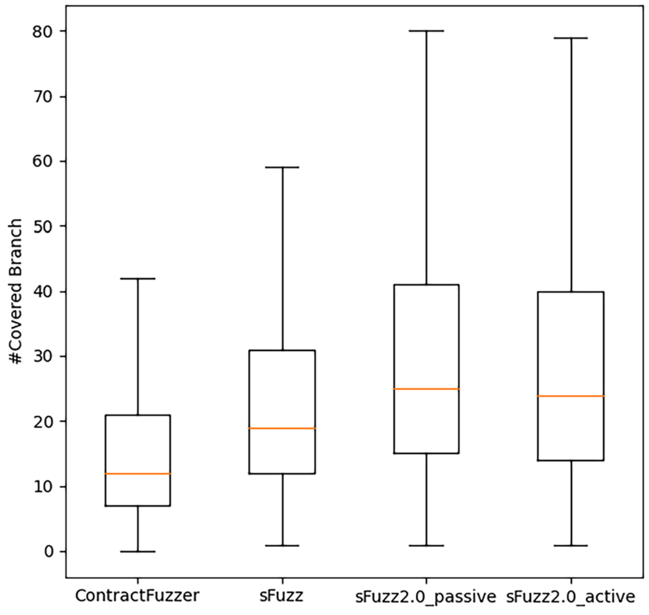
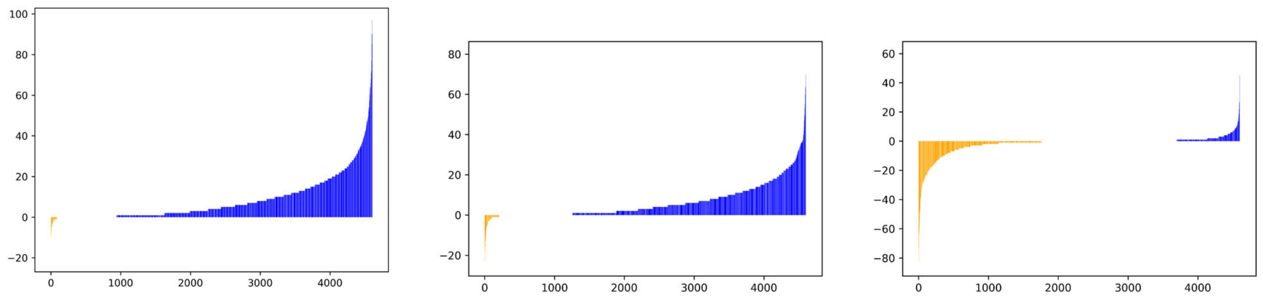


FIGURE 4 Branch coverage of ContractFuzzer, sFuzz, and `sFuzz2.0` in 10 min.



(A) `sFuzz2.0`$_{passive}$ vs.sFuzz (B) `sFuzz2.0`$_{active}$ vs. sFuzz (C) `sFuzz2.0`$_{active}$ vs. `sFuzz2.0`$_{passive}$

FIGURE 5 Comparison on the number of covered branches.

(e.g., sFuzz for Figure 5A). Each point on the horizontal axis represents a smart contract. The contracts are sorted by the difference. Thus, the more points above the 0 line, the better the first approach is.

First, both `sFuzz2.0`$_{passive}$ and `sFuzz2.0`$_{active}$ outperform sFuzz, that is, `sFuzz2.0`$_{passive}$ covers more branches than sFuzz on more than four fifths of the contracts (i.e., 3662 out of 4603) and `sFuzz2.0`$_{active}$ covers more branches than sFuzz on 3348 out of the 4603 contracts. In contrast, sFuzz covers more branches on 87 contracts than `sFuzz2.0`$_{passive}$ and 208 contracts than `sFuzz2.0`$_{active}$. Our conjecture on why `sFuzz2.0` is not universally better than sFuzz is that (1) while covering more SAPs improves branch coverage in general, it may not always be the case; (2) `sFuzz2.0` spends time on code instrumentation (for collecting storage-access events and identifying SAPs), which makes `sFuzz2.0` slightly slower than sFuzz.

Second, comparing `sFuzz2.0`$_{passive}$ and `sFuzz2.0`$_{active}$, as shown in Figure 5C, we observe that `sFuzz2.0`$_{passive}$ and `sFuzz2.0`$_{active}$ complement each other. That is, `sFuzz2.0`$_{active}$ covers more branches in 907 contracts whereas `sFuzz2.0`$_{passive}$ covers more branches in 1763 contracts. We conjecture that the reason is that while active fuzzing is more active in triggering new SAPs, it suffers from some significant overhead as well. Note that we put this conjecture under further examination in RQ3.

We further conduct an experiment to measure the throughput, that is, the number of test cases generated and executed per second, of each method. Please note that we use the unified experiment setup in RQ1. The average throughput of sFuzz, `sFuzz2.0`$_{passive}$, `sFuzz2.0`$_{active}$ are 786.95, 778.74, and 726.48, respectively. Overall, the efficiency of `sFuzz2.0` is around 7.7% lower than sFuzz. `sFuzz2.0` $_{passive}$, which is only 0.99% less efficient than sFuzz, is more efficient than `sFuzz2.0`$_{active}$. The overhead of sFuzz 2.0 is mainly caused by the calculation of SAP. Considering the branch coverage increment that `sFuzz2.0` achieves, we believe this overhead is worthwhile and acceptable.

RQ2: Is `sFuzz2.0` effective in obtaining high SAP-Coverage?

Next, we evaluate whether `sFuzz2.0` achieves its goal of covering more SAPs. Because it is infeasible to obtain the total number of patterns, we report the unique patterns covered during the fuzzing process. That is, we systematically measure the number of unique SAPs covered on each contract by sFuzz, $\texttt{sFuzz2.0}_{active}$, and $\texttt{sFuzz2.0}_{passive}$. As discussed above, given the difficulty of distinguishing patterns regarding dynamic-sized variables, we focus on patterns on fixed-sized variables only.

The result is summarized in Figure 7. Similarly, for each pair of approaches, the vertical axis is the number of patterns covered by the first approach minus that of the second approach. The comparison between sFuzz and `sFuzz2.0` is as expected, that is, both $\texttt{sFuzz2.0}_{passive}$ and $\texttt{sFuzz2.0}_{active}$ cover more SAPs than sFuzz on almost all the contracts. The comparison between $\texttt{sFuzz2.0}_{passive}$ and $\texttt{sFuzz2.0}_{active}$, as shown in Figure 6C, again reveals a mixed story, that is, $\texttt{sFuzz2.0}_{passive}$ and $\texttt{sFuzz2.0}_{active}$ both outperform each other on a large number of contracts. We conjecture that this is because although $\texttt{sFuzz2.0}_{active}$ is likely to achieve a higher SAP-Coverage if the same number of test cases are executed, $\texttt{sFuzz2.0}_{passive}$ may still be superior in some contracts as it suffers from less overhead, which is critical for fuzzing algorithms.[3,15,16] In the next part, we put this conjecture under test in RQ3 and then propose a way of combining $\texttt{sFuzz2.0}_{passive}$ and $\texttt{sFuzz2.0}_{active}$.

RQ3: Is the overhead of $\texttt{sFuzz2.0}_{active}$ *justified and is the passive and active strategy complementary?*

To answer RQ3, we show how the number of covered branches and covered patterns changes with time. If our conjecture that $\texttt{sFuzz2.0}_{active}$ is slower but more effective in covering SAPs is correct, we expect $\texttt{sFuzz2.0}_{active}$ to catch up $\texttt{sFuzz2.0}_{passive}$ given sufficient time. In other words, though suffered from more overhead, compared to passive, $\texttt{sFuzz2.0}_{active}$ is supposed to cover more SAPs in the long run if enough potential patterns are identified. In this experiment, we randomly sample 1000 contracts and set the fuzzing timeout to be 100 min. The trend over time in terms of branch coverage is shown in Figure 7A, where the *y* axis is the average number of covered branches of all contracts. Correspondingly, Figure 7B shows the trend of SAP-Coverage. Note that we similarly only count patterns regarding static-sized variables.

Considering the branch coverage the results show that $\texttt{sFuzz2.0}_{passive}$ consistently outperforms $\texttt{sFuzz2.0}_{active}$, and both $\texttt{sFuzz2.0}_{passive}$ and $\texttt{sFuzz2.0}_{active}$ consistently and significantly outperform sFuzz. Furthermore, with a time limit of 1 h, the gap between $\texttt{sFuzz2.0}_{passive}$ and $\texttt{sFuzz2.0}_{active}$ shows no sign of shrinking. Considering the SAP-Coverage, the passive strategy and the active strategy lead alternatively. Finally, after around 2700 s, the active strategy is overtaken by the passive strategy. We further evaluate the success rate of the active strategy, that is, the percentage of times that the to-be-covered SAP is actually covered after the mutation. The result is that the success rate is on average around 50%, which we believe is reasonably good considering complications such as infeasible program paths. A further investigation shows that for over

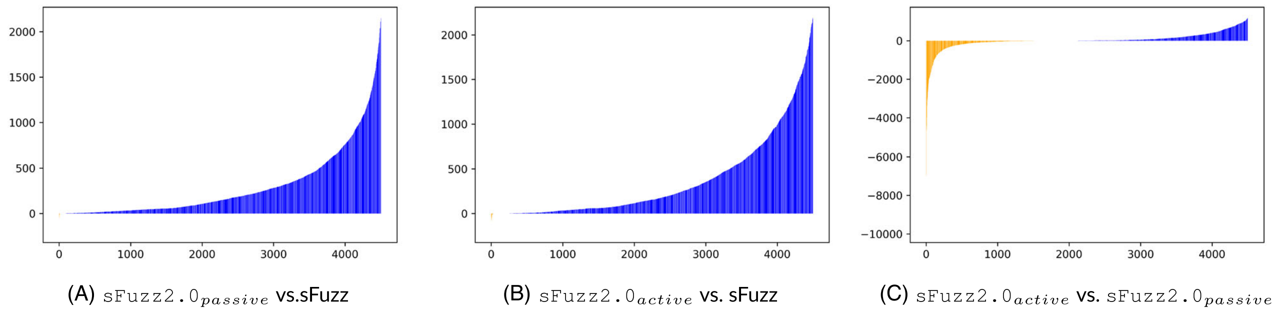
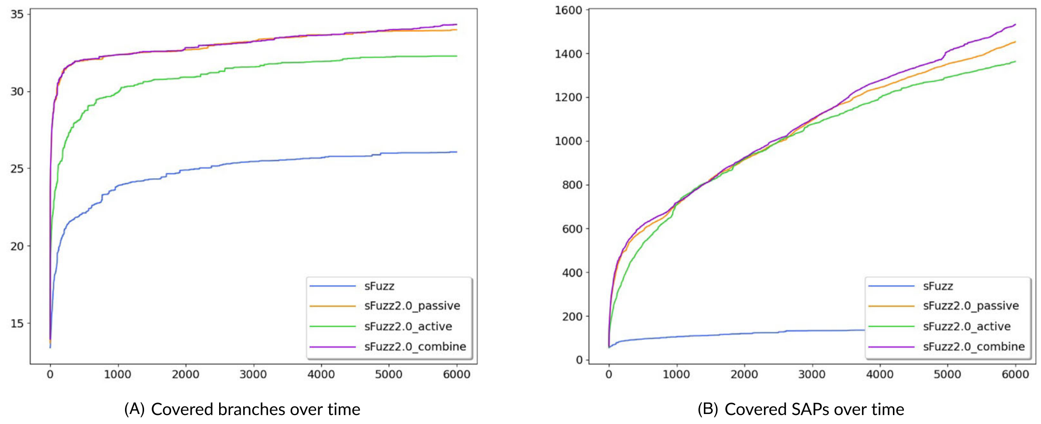

(A) $\texttt{sFuzz2.0}_{passive}$ vs.sFuzz (B) $\texttt{sFuzz2.0}_{active}$ vs. sFuzz (C) $\texttt{sFuzz2.0}_{active}$ vs. $\texttt{sFuzz2.0}_{passive}$

FIGURE 6 Comparison on the number of covered SAPs.



(A) Covered branches over time (B) Covered SAPs over time

FIGURE 7 Branch/pattern coverage over time.

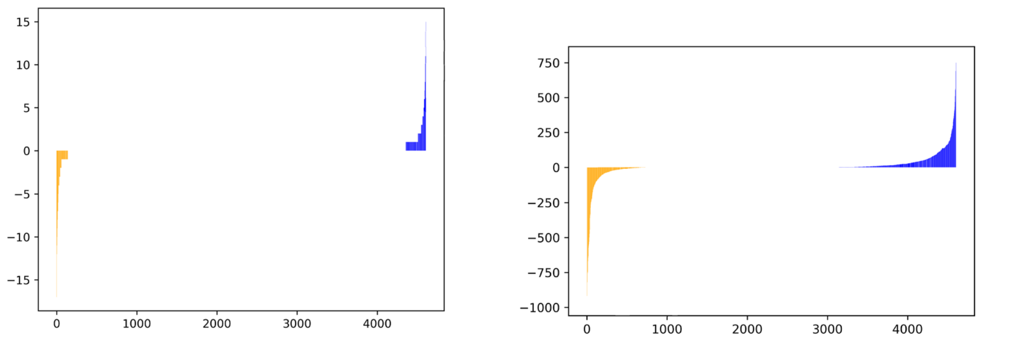
half of the contracts, the total number of potential patterns identified in Algorithm 5 in `sFuzz2.0`$_{active}$ is less than or equal to the total number of covered patterns in `sFuzz2.0`$_{active}$. Our conclusion is thus that `sFuzz2.0`$_{active}$ fails to cover more SAPs for many contracts because it is limited by the already-seen storage-access events collected by *updateRW*. In other words, if some storage-access events have never been triggered, `sFuzz2.0`$_{active}$ has no idea that they exist and thus would not be able to cover the corresponding SAPs. On the other hand, because `sFuzz2.0`$_{passive}$ mutates randomly, it is more likely to trigger new storage-access events.

The result suggests `sFuzz2.0`$_{passive}$ and `sFuzz2.0`$_{active}$ naturally complement each other. That is, `sFuzz2.0`$_{passive}$ can be applied initially to trigger many storage-access events (and cover those easy to cover SAPs), whereas `sFuzz2.0`$_{active}$ can be applied to cover hard-to-cover SAPs based on those storage-access events identified by `sFuzz2.0`$_{passive}$. To evaluate our hypothesis, we proposed a strategy which combines `sFuzz2.0`$_{passive}$ and `sFuzz2.0`$_{active}$ sequentially, that is, for each contract, given a timeout of 10 min, we first apply `sFuzz2.0`$_{passive}$ for 8 min and then switch to `sFuzz2.0`$_{active}$ for 2 min. The results are summarized in Figure 8A,B. Compared with `sFuzz2.0`$_{passive}$, the combined version covers 65 more branches in total. We believe that the combined strategy can potentially perform better if more time is available. As shown in the Figure 7A,B, after 4800 s (i.e., the point when switching the strategy from passive to active), the combine strategy starts to cover more branches and more patterns than the passive strategy, and the gap between the two strategies shows an increasing trend. The results suggest that the active strategy complements the passive strategy and the combined strategy shows the best performance given a longer fuzzing time budget.

RQ4: Is `sFuzz2.0` able to find new vulnerabilities?

To answer RQ4, we adopt the nine kinds of oracles used in sFuzz and run `sFuzz2.0` (with the passive, active, and combined strategy as discussed above) on all contracts to detect vulnerabilities. Note that the exact same vulnerability detectors are adopted from sFuzz.

The result is shown in Table 2, where the first column is the type of vulnerability, and the following column shows the total number of vulnerabilities found by five approaches in the five subcolumns. `sFuzz2.0` finds more vulnerabilities than sFuzz in all categories. For all categories,



(A) $\texttt{sFuzz2.0}_{combine}$ vs. $\texttt{sFuzz2.0}_{passive}$: Branches
(B) $\texttt{sFuzz2.0}_{combine}$ vs. $\texttt{sFuzz2.0}_{passive}$: Patterns

FIGURE 8 Comparison between combine and passive strategy.

TABLE 2 Vulnerabilities: `sFuzz2.0` versus sFuzz and ContractFuzzer.

	#Vulnerabilities				
Vulnerability Type	**ContractFuzzer**	**sFuzz**	`sFuzz2.0`$_{passive}$	`sFuzz2.0`$_{active}$	`sFuzz2.0`$_{combine}$
Gasless Send	1066	1323	1556	1468	1555
Dangerous Delegate Call	2	6	7	7	7
Freezing Ether	2	6	7	7	7
Reentrancy	0	31	37	28	37
Exception Disorder	0	73	75	63	75
Integer Overflow	8	10	17	14	17
Integer Underflow	1	3	5	6	6
Timestamp Dependency	454	483	622	584	620
Block Number Dependency	73	90	114	105	114
Total	1606	2025	2440	2282	2438

passive, active, and combined strategies of `sFuzz2.0` find more or equal number of vulnerable contracts compared to sFuzz and ContractFuzzer. The passive and the combined strategies obtain the highest pattern coverage and branch coverage; thus, they reveal the maximum number of vulnerabilities (i.e., 315 new vulnerabilities reported comparing to sFuzz) as expected. Note that the active strategy also outperforms sFuzz. Detailed investigations show that certain function sequences and argument values are needed to trigger those new vulnerabilities, and the SAP-Coverage criteria is critical for fuzzing algorithms to detect such vulnerable contracts.

To illustrate how `sFuzz2.0`'s approach helps to discover vulnerabilities, we present the following vulnerable contract which is detected by `sFuzz2.0` but not sFuzz and ContractFuzzer. The simplified example in Figure 9 contains a dangerous delegate call at Line 18. The statement *delegateCall* passes the context of the *msg.sender* and storage variables to the *addr* contract. An attacker can take advantage of this by first calling *update*() to set *deprecated* to be true and pointing *addr* to the address of the attacker contract in Figure 10; then, the attacker can call *check*() to invoke a delegate call to the *end*() function in Attacker contract. Because the *delegateCall* will execute the code on the caller contract, the owner of *Ohni* is set to the address of the attacker contract. Note that this vulnerability in Figure 9 is only found by `sFuzz2.0` because it requires a specific sequence of function calls with specific parameters. To trigger the vulnerability, the *check*() function should be called when the state variable *deprecated* is true. To satisfy this condition, the *update* function should be called previously where the *msg.sender* should be equal to *owner* and the parameter *depr* should be true. sFuzz and ContractFuzzer failed to generate the required function sequence; therefore, they failed to detect the vulnerability.

Another vulnerable contract exposed by `sFuzz2.0` but not the other tools is simplified and shown in Figure 11. This contract contains a *Timestamp Dependency vulnerability* from Lines 20 to 21. Function *purchase* will transfer to the caller (at Line 21) if a day has passed since the *lastPurchase*. Because the timestamp can be manipulated by miners, it is unsafe to use the timestamp as part of the condition that determines transfer operations. As a result, in this vulnerable contract, the transfer may happen at the incorrect time (e.g., before *lastPurchase* + 1*days*) and cause losses. To trigger the vulnerability, the *require* condition of function *purchase* at Line 19 must be satisfied, and consequently, function *createListing* must be called before *purchase* and set *burrito.price* (at Line 12) to be the proper value (i.e., with range [0,*msg.value*]). sFuzz and ContractFuzzer either fail to generate the function sequence or the proper value for argument *_startingPrice* (which will affect the value of *burrito.price*) and thus did not expose the vulnerable contract.

Threats to validity. First, due to limited resources, we are unable to run all contracts from SmartBugs. Despite our sample size being reasonable, experimenting with all contracts would provide better assurance of the results. Second, identifying dynamic-sized variables at the bytecode level is a challenge to be addressed in the future. It leads to inaccuracy in counting the number of covered SAPs, and thus, in this work, we only count those SAPs with respect to static-sized variables. Third, though `sFuzz2.0` can improve existing fuzzers by achieving higher code coverage, the detecting ability may still be limited by the oracles we adopt, this can be further improved by adopting advanced oracles.

```
 1  contract Ohni {
 2    address owner = msg.sender;
 3    address addr;
 4    bool deprecated = false;
 5
 6    function set_owner(address addr) public {
 7        require(owner == msg.sender);
 8        owner = addr;
 9    }
10
11    function update(address newAddress, bool depr) {
12      addr = newAddress;
13      deprecated = depr;
14    }
15
16    function check() {
17      if (deprecated) {
18        if (!addr.delegatecall(bytes4(keccak256("end()"))))    throw;
19      }
20    }
21  }
```

FIGURE 9 Code snippet of *Dangerous Delegate Call* vulnerability found by `sFuzz2.0` but not sFuzz or ContractFuzzer.

```
contract Attacker {
    address owner;
    address addr;

    function end() {
        owner = address(this);
    }
}
```

FIGURE 10 Attacker contract to exploit the Dangerous Delegate Call vulnerability.

```
contract BurritoToken is ERC721, Ownable {
  struct Burrito {
      uint256 price;
      //...
  }
  mapping (uint256 => Burrito) public burritoData;

  function createListing(uint256 _tokenId, uint256 _startingPrice, uint256 _payoutPercentage,
       address _owner) onlyOwner() public {
    require(_startingPrice > 0);
    require(burritoData[_tokenId].price == 0);
    Burrito storage newBurrito = burritoData[_tokenId];
    newBurrito.price = getNextPrice(_startingPrice);
    //...
  }

  function purchase(uint256 _tokenId) public payable {
    Burrito storage burrito = burritoData[_tokenId];
    uint256 price = burrito.price;
    require(price > 0 &&msg.value > price );
    require(now > lastPurchase + 1 days);
    msg.sender.transfer(excess);
    lastPurchase = now;
  }
}
```

FIGURE 11 Code snippet of *Timestamp Dependency* vulnerability found by `sFuzz2.0` but not sFuzz or ContractFuzzer.

6 | RELATED WORK

`sFuzz2.0` is directly related to existing fuzzing approaches to test smart contracts. ContractFuzzer[5] applies black-box fuzzing on smart contracts and defines seven types of vulnerabilities. Echidna[7] supports user-defined properties. Harvey[8] extends standard grey-box fuzzer with input prediction and demand-driven transaction sequence fuzzing. sFuzz[3] is another grey-box fuzzer which applies a lightweight multiobjective adaptive strategy towards covering strict conditions. Smartian[11] leverage data-flow analysis and taint analysis to effectively and precisely detect the bugs. ILF[29] learns fuzzing policy (i.e., how to append transaction to current sequence). ILF first leverages symbolic execution to get sequences that maximize the coverage, encodes the transactions into feature vectors, and then feeds the feature vectors to RNN to learn the fuzzing policies. The effectiveness of ILF relies heavily on the training set. Recent study[26,27] shows that the sFuzz is complementary to existing fuzzers. Confuzzius[30] is a hybrid fuzzer that combines fuzzing and symbolic execution. Confuzzius tracks the invocation of *SSTORE* and *SLOAD* and constructs Read-After-Write dependency to generate a meaningful transaction sequence. However, focusing only on the Read-After-Write constraint is not complete. For example, the order of two consecutive writes is important since the final state depends on the value of the last write. Compared with the existing fuzzing approaches, `sFuzz2.0` takes function order into consideration and achieves better performance.

Symbolic execution is another approach to detect bugs, these tools first construct CFGs through static analysis, then check whether a path is feasible through constraint solving. Oyente[4] checks four kinds of vulnerabilities. TeEther[9] explores critical paths that reach instructions which can be manipulated by attackers and then generate a sequence of transactions to create an exploit. MAIAN[10] is developed to detect three types of *trace* vulnerabilities. Osiris[31] leverages taint analysis and symbolic execution to detect integer bugs. Mythril[13] is a security analysis tool for EVM-compatible blockchains. Manticore[14] is a symbolic execution framework used in Trail of Bits. Solar[32] is an adversarial attack synthesizer, which uses a summary-based symbolic evaluation to reduce the number of instructions to be evaluated. sCompile[6] constructs a CFG including inter-contract function calls, prioritized path by computing critical scores to address path explosion. Smartian[11] leverages on grey-box concolic, which approximate branch constraints with the linear and monotonic relationships. The general problem of symbolic execution is constraint solving is computationally expensive, especially for smart contracts because the hash function is commonly used and is difficult for SMT solvers like Z3.[33] It has shown that fuzzing and symbolic execution are complementary[34,35]; thus, `sFuzz2.0` can combine with those symbolic execution tools to improve efficiency.

Our idea is also inspired by concurrent program testing. `sFuzz2.0` takes parallel execution of multiple functions into consideration and is related to coverage-guided concurrency program testing tools. Choudhary et al[36] present a tool that generates test cases that are likely to cover new method pairs. Wang et al[18] adopt the memory-access patterns defined in Unicorn[25] and propose a novel MAP-Coverage to guide the searching process. We adopt the idea of coverage-guided fuzzing and propose SAP-Coverage to guide the fuzzing process.

`sFuzz2.0` is also related to the formal verification and analysis of smart contracts. Bhargavan et al[37] transform contracts into formal languages F*. Zeus[38] first translates smart contracts into LLVM bit code and then uses abstract interpretation and symbolic model checking to analyze contracts. SmartCheck[39] translates Solidity source code into XML and checks it against XPath patterns. Ethor[40] proposes a static Smart Contract analyzer based on an abstraction of bytecode semantics based on Horn clauses. FairCon[41] is designed to verify smart contract fairness. Clairvoyance[12] is a cross-contract and cross-function static analyzer focused on reentrancy bugs. Fröwis and Böhme[42] extract call graph to discover mutable control flows which make the contract less trustworthy and find out that 40% contracts need a third party to trust. Pluto[43] is the first to build an inter-contract control flow graph (ICFG) to extract semantic information among contract calls, enumerate reachable paths, and then detect whether there is a vulnerability based on some predefined rules. Chen et al[44] identify seven gas-costly patterns and analyze bytecode to automatically localize these patterns. Securify[45] extracts precise semantic information from the code and checks patterns to prove whether the behavior satisfied the given property. MadMax[46] is a static program analysis technique focused on gas-related vulnerabilities. Vandal[47] converts bytecode to semantic logic relations. Mariano et al[48] summarize how loops are used using domain-specific language. Grossman et al[49] present a general correctness condition for callback. Differing from the above approaches implemented through static analysis, `sFuzz2.0` detects vulnerabilities dynamically.

7 | CONCLUSION

To conclude, we propose `sFuzz2.0` in this work, which extends sFuzz by selectively generating function call sequences guided by patterns defined in SAP-Coverage. We design two different strategies, that is, the active strategy and the passive strategy, to trigger patterns. The experiment results show that `sFuzz2.0` outperforms state-of-the-art fuzzers by achieving higher code coverage and finding more vulnerabilities. We further combine the passive and the active strategies sequentially, which achieves the best overall results.

ACKNOWLEDGMENTS

This work is partially funded by projects 61872263, U1836214, and 61802275 from the National Natural Science Foundation of China, and No. NJ2022027 from the Ministry of Industry and Information Technology Open Project.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in sFuzz2.0 at https://github.com/duytai/sFuzz/tree/v2.0.

ORCID

Haoyu Wang https://orcid.org/0000-0001-9377-7246

REFERENCES

1. Smart contract. https://en.wikipedia.org/wiki/Smart_contract
2. Understanding the DAO attack. https://www.coindesk.com/understanding-dao-hack-journalists
3. Nguyen TD, Pham LH, Sun J, Lin Y, Minh QT. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In: ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June–19 July 2020 Rothermel G, Bae D-H, eds. ACM; 2020:778-788.
4. Luu L, Chu D-H, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. Cryptology ePrint Archive, Report 2016/633; 2016.
5. Jiang B, Liu Y, Chan WK. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018 Huchard M, Kästner C, Fraser G, eds. ACM; 2018:259-269.
6. Chang J, Gao B, Xiao H, Sun J, Cai Y, Yang Z. sCompile: critical path identification and analysis for smart contracts. In: Formal Methods and Software Engineering: 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5–9, 2019. Springer International Publishing; 2019.
7. Grieco G, Song W, Cygan A, Feist J, Groce A. Echidna: effective, usable, and fast fuzzing for smart contracts. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2020. Association for Computing Machinery; 2020:557-560.
8. Wüstholz V, Christakis M. 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. ESEC/FSE 2020. Association for Computing Machinery; 2020: 1398-1409.
9. Krupp J, Rossow C. teEther: gnawing at Ethereum to automatically exploit smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association; 2018:1317-1333. https://www.usenix.org/conference/usenixsecurity18/presentation/krupp
10. Nikolić I, Kolluri A, Sergey I, Saxena P, Hobor A. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference. ACSAC '18. Association for Computing Machinery; 2018:653-663.
11. Choi J, Kim D, Kim S, Grieco G, Groce A, Cha SK. SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE; 2021:227-239.
12. Xue Y, Ma M, Lin Y, Sui Y, Ye J, Peng T. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ASE '20. Association for Computing Machinery; 2020: 1029-1040.
13. Mythril Classic; 2018. https://github.com/ConsenSys/mythril
14. Mossberg M, Manzano F, Hennenfent E, et al. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE; 2019.
15. Technical whitepaper for afl-fuzz. https://lcamtuf.coredump.cx/afl/techical_details.txt
16. Böhme M, Pham V-T, Roychoudhury A. Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16. Association for Computing Machinery; 2016:1032-1043.

17. Atzei N, Bartoletti M, Cimoli T. A survey of attacks on Ethereum smart contracts; 2016. https://eprint.iacr.org/2016/1007
18. Wang Z, Zhao Y, Liu S, Sun J, Chen X, Lin H. MAP-Coverage: a novel coverage criterion for testing thread-safe classes. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019. IEEE; 2019:722-734.
19. sFuzz2.0. https://github.com/duytai/sFuzz/tree/v2.0
20. Klees G, Ruef A, Cooper B, Wei S, Hicks M. Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2018:2123-2138.
21. Jiang B, Liu Y, Chan WK. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ASE 2018. ACM; 2018:259-269.
22. Luu L, Chu D-H, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2016:254-269.
23. Daian P. Analysis of the DAO exploit. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/
24. The Parity Wallet hack explained. https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/
25. Park S, Vuduc R, Harrold MJ. A unified approach for localizing non-deadlock concurrency bugs. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE; 2012:51-60.
26. Ghaleb A, Pattabiraman K. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020 Khurshid S, Pasareanu CS, eds. ACM; 2020:415-427.
27. Ren M, Yin Z, Ma F, et al. Empirical evaluation of smart contract testing: what is the best choice?. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2021. Association for Computing Machinery; 2021:566-579.
28. Groce A, Grieco G. echidna-parade: a tool for diverse multicore smart contract fuzzing. In: ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021 Cadar C, Zhang X, eds. ACM; 2021:658-661.
29. He J, Balunović M, Ambroladze N, Tsankov P, Vechev M. Learning to fuzz from symbolic execution with application to smart contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. CCS '19. Association for Computing Machinery; 2019:531-548.
30. Torres CF, Iannillo AK, Gervais A, State R. ConFuzzius: a data dependency-aware hybrid fuzzer for smart contracts. In: 2021 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE; 2021:103-119.
31. Torres CF, Schütte J, State R. Osiris: hunting for integer bugs in Ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. ACM; 2018:664-676.
32. Feng Y, Torlak E, Bodik R. Summary-based symbolic evaluation for smart contracts. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). Association for Computing Machinery; 2020:1141-1152.
33. z3. https://github.com/Z3Prover/z3
34. Wang X, Sun J, Chen Z, Zhang P, Wang J, Lin Y. Towards optimal concolic testing. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18. Association for Computing Machinery; 2018:291-302.
35. Yun I, Lee S, Xu M, Jang Y, Kim T. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association; 2018:745-761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun
36. Choudhary A, Lu S, Pradel M. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE; 2017:266-277.
37. Bhargavan K, Delignat-Lavaud A, Fournet C, et al. Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016 Murray TC, Stefan D, eds. ACM; 2016:91-96.
38. Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society; 2018. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf
39. Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. SmartCheck: static analysis of Ethereum smart contracts. In: 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). Association for Computing Machinery; 2018:9-16.
40. Schneidewind C, Grishchenko I, Scherer M, Maffei M. eThor: practical and provably sound static analysis of Ethereum smart contracts. In: CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020 Ligatti J, Ou X, Katz J, Vigna G, eds. ACM; 2020:621-640.
41. Liu Y, Li Y, Lin S-W, Zhao R. Towards automated verification of smart contract fairness. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2020. Association for Computing Machinery; 2020:666-677.
42. Fröwis M, Böhme R. In code we trust?—measuring the control flow immutability of all smart contracts deployed on Ethereum. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology—ESORICS 2017 International Workshops, DPM 2017 and CBT 2017, Oslo, Norway, September 14-15, 2017, Proceedings García-Alfaro J, Navarro-Arribas G, Hartenstein H, Herrera-Joancomartí J, eds., Lecture Notes in Computer Science, vol. 10436. Springer; 2017:357-372.
43. Ma F, Xu Z, Ren M, et al. Pluto: exposing vulnerabilities in inter-contract scenarios. *IEEE Trans Softw Eng*. 2021;48(11):4380-4396.
44. Chen T, Li X, Luo X, Zhang X. Under-optimized smart contracts devour your money. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017 Pinzger M, Bavota G, Marcus A, eds. IEEE Computer Society; 2017:442-446.
45. Tsankov P, Dan A, Cohen DD, Gervais A, Buenzli F, Vechev M. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery; 2018.
46. Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proc ACM Program Lang*. 2018;2(OOPSLA):116:1-116:27. https://doi.org/10.1145/3276486
47. Brent L, Jurisevic A, Kong M, et al. Vandal: a scalable security analysis framework for smart contracts; 2018.

48. Mariano B, Chen Y, Feng Y, Lahiri SK, Dillig I. Demystifying loops in smart contracts. In: 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. IEEE; 2020:262-274.
49. Grossman S, Abraham I, Golan-Gueta G, et al. Online detection of effectively callback free objects with applications to smart contracts. *Proc ACM Program Lang*. 2018;2(POPL):48:1-48:28. https://doi.org/10.1145/3158136

How to cite this article: Wang H, Wang Z, Liu S, et al. `sFuzz2.0`: Storage-access pattern guided smart contract fuzzing. *J Softw Evol Proc*. 2023;e2557. doi:10.1002/smr.2557