sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts

Tai D. Nguyen
Singapore Management University, Singapore
dtnguyen.2019@smu.edu.sg

Long H. Pham
Singapore Management University, Singapore
longph1989@gmail.com

Jun Sun
Singapore Management University, Singapore
sunjunhqq@gmail.com

Yun Lin
National University of Singapore, Singapore
llmhyy@gmail.com

Quang Tran Minh
Ho Chi Minh City University of Technology, Vietnam
quangtran@hcmut.edu.vn

ABSTRACT

Smart contracts are Turing-complete programs that execute on the infrastructure of the blockchain, which often manage valuable digital assets. Solidity is one of the most popular programming languages for writing smart contracts on the Ethereum platform. Like traditional programs, smart contracts may contain vulnerabilities. Unlike traditional programs, smart contracts cannot be easily patched once they are deployed. It is thus important that smart contracts are tested thoroughly before deployment. In this work, we present an adaptive fuzzer for smart contracts on the Ethereum platform called sFuzz. Compared to existing Solidity fuzzers, sFuzz combines the strategy in the AFL fuzzer and an efficient lightweight multi-objective adaptive strategy targeting those hard-to-cover branches. sFuzz has been applied to more than 4 thousand smart contracts and the experimental results show that (1) sFuzz is efficient, e.g., two orders of magnitude faster than state-of-the-art tools; (2) sFuzz is effective in achieving high code coverage and discovering vulnerabilities; and (3) the different fuzzing strategies in sFuzz complement each other.

ACM Reference Format:
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 *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3377811.3380334

1 INTRODUCTION

Nowadays, smart contracts [11, 28] are implemented as Turing-complete programs that execute on the infrastructure of the blockchain [33]. It provides a framework that potentially allows any program (equivalently, contract) to be executed in an autonomous, distributed, and trusted way. Smart contracts thus have the potential to revolutionize many industries. Popular applications of smart contracts include crowd fundraising, online gambling and so on. Ethereum [1, 31] is the first to introduce the functionality of smart contracts. Based on the Ethereum platform, Solidity is the most popular programming language for smart contracts [6].

Like traditional C or Java programs, smart contracts may contain vulnerabilities. Unlike traditional programs, smart contracts cannot be modified easily once they are deployed on the blockchain [23]. As a result, a vulnerability renders the smart contract forever vulnerable, which significantly magnifies the problem. In recent years, there has been an increasing number of news reports on attacks which exploit security vulnerabilities in Ethereum smart contracts. One particularly noticeable example is the DAO attack [12], i.e., an attacker stole more than 3.5 million Ether (which is equivalent to about $45 million USD at the time) exploiting a vulnerability in the DAO contract. To fix the vulnerability, a hard fork was launched which was not only expensive but also caused much controversy [12].

It is thus desirable to develop tools for validating smart contracts to identify vulnerabilities, ideally before they are deployed. Among the range of complementary techniques for validating smart contracts, we focus on automatic testing of smart contracts in this work as testing is often the least expensive and thus the most applicable. To automatically test smart contracts, we must solve the following three problems:

- the test automation problem (i.e., how to run test cases),
- the test generation problem (i.e., what to test),
- and the oracle problem (i.e., what are vulnerabilities).

In the literature, several approaches have been developed for automatic testing smart contracts, each of which answers these three problems in slightly different ways. For instance, ContractFuzzer [18] builds a network with pre-deployed contracts and generates transactions to run smart contracts, generates test cases based on a set of predefined parameter values and targets a set of oracles specific for smart contracts. Oyente [22] runs smart contracts symbolically through symbolic execution, generates test cases for covering different program paths in single functions through constraint solving, and supports multiple oracles to identify 4 kinds of vulnerabilities. teEther [21] similarly applies symbolic execution to generate test cases covering program paths, and focuses on oracles which are related to financial transactions.

In this work, we propose a fully automatic testing engine for smart contracts running on Ethereum called sFuzz. sFuzz is inspired by AFL [7], a well-known fuzzer for C programs, i.e., sFuzz is a feedback-guided fuzzing engine and is inexpensive to apply. sFuzz

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380334

complements existing testing engines based on symbolic execution like Oyente and teEther, as it is known that fuzzing and symbolic execution are complementary [30, 32]. While AFL-based fuzzing is often effective, it has its limitation as well, i.e., it is often expensive in covering branches guarded with strict conditions. To tackle the problem, sFuzz integrates AFL-based fuzzing with an efficient lightweight adaptive strategy for selecting seeds. Although inspired by search-based software testing [16, 24], the latter distinguishes itself by having a lightweight objective function (designed considering characteristics of Solidity programs) as well as a novel multi-objective optimization strategy.

sFuzz is built based on Aleth [2] (i.e., an Ethereum VM written in C++), has a system architecture similar to AFL, and is extensible to different Ethereum VMs and oracles as well as fuzzing strategies. sFuzz has been systematically applied to a set of more than 4 thousand smart contracts. The experimental results show that sFuzz is on average more than two orders of magnitudes faster than ContractFuzzer, covers more branches and reveals many more vulnerabilities. A comparison between sFuzz and Oyente shows that they are complementary. Furthermore, experiments with prolonged fuzzing time show that the adaptive strategy improves code coverage. sFuzz is available online and has been adopted by multiple companies.

The remainder of the paper is organized as follows. Section 2 illustrates how sFuzz works through examples. Section 3 presents the details of the approach. Section 4 shows implementation details of sFuzz. Section 5 reports evaluation results. Section 6 reviews related work and concludes.

2 ILLUSTRATIVE EXAMPLES

In this section, we show how sFuzz works step-by-step through two illustrative examples. Note that Solidity source codes for both examples are shown for simplicity. sFuzz requires only the EVM (i.e., Ethereum Virtual Machine) bytecode [1, 31] to fuzz smart contracts.

Given a smart contract, sFuzz automatically configures a blockchain network, deploys the smart contract, and generates multiple transactions each of which calls a function in the contract. The transactions are then executed with an EVM enriched with a set of oracles for identifying vulnerabilities. sFuzz monitors the execution of the transactions to collect certain feedback, e.g., whether a certain branch has been covered and how far the branch is covered. Whenever a vulnerability is revealed, the transactions and the network configuration (i.e., a test case) are saved and reported to the user later on. Otherwise, some of the test cases are selected as *seeds* based on feedback collected during the transaction execution according to certain seed selection criteria. Afterwards, the *seeds* are mutated to generate the next generation of test cases. This process repeats until a time out occurs.

In the following, we describe how sFuzz works using the contract shown in Figure 1. The contract implements a simple quiz game. The contract is based on contract *opposite_game*[1] with minor modification for simplicity. A quiz can be created by calling function *start_quiz_game*. The response is hashed and then saved in the *responseHash* variable. The user then calls the *try* function with their answer as the argument and pays a fee of 100 *finney* (which

[1]address: 0x467532e79222670a2044c9b168bcbaa33b390ef5

```
1  pragma solidity ^0.4.20;
2  contract opposite_game {
3    string public question;
4    address questionSender;
5    bytes32 responseHash;
6
7    function Try(
8      string _response ) external payable {
9      if(responseHash == keccak256(_response) &&
10         msg.value == 100 finney) {
11       msg.sender.send(this.balance); } }
12
13   function start_quiz_game(
14     string _question, string _answer) public payable {
15     if(responseHash==0x0) {
16       responseHash = keccak256(_answer);
17       question = _question;
18       questionSender = msg.sender; } }
19
20   function() public payable {} }
```

Figure 1: An example with single objective function

is a unit of the token) for each try. If the answer is correct, a reward is sent to the user.

This contract suffers from a vulnerability known as *Gasless Send* when line 11 is executed and a costly *fallback function* is called. That is, when function *send*() at line 11 is executed, if the receiver is a contract, its fallback function is executed automatically. Because function *send*() only forwards 2300 units of gas (i.e., price to pay for executing the function), an *out-of-gas* exception is thrown if the fallback function is costly (e.g., costs more than 2300 units of gas). In this case, the *send*() function simply returns $false$ and because the returned value is not checked and handled accordingly, the owners of the contract can keep the reward for themselves.

To expose this vulnerability, first a network is configured with several addresses and associated balances. This contract is then deployed at one of the addresses. In addition, an attacker contract with a costly fallback function is deployed automatically. To expose the vulnerability, a test case (i.e., a sequence of transactions) with such a network configuration must first call function *start_quiz_game* and then function *Try* with parameters such that all 2 conditions in function *Try* at line 9 and 10 are satisfied. The condition at line 9 is satisfied with a test case that sets all the parameters and contract variables to the default value of 0. Note that *responseHash* is set to *keccak*256(*_answer*) at line 16 and is compared to *keccak*256(*_response*) at line 9. However, generating a test case which satisfies the second condition by randomly generated test values is highly unlikely. The variable *msg.value* has a size of 32 bytes and thus we have only $\frac{1}{2^{256}}$ probability to generate the value 100 (if we generate random values with a uniform distribution among all possible values). Existing fuzzing strategy in AFL is ineffective in this case as well, i.e., AFL selects test cases that cover new branches as *seeds*. Since all test cases generated through mutation are unlikely to cover the then-branch at line 10, they are equally 'bad' according to the AFL seed selection strategy.

sFuzz complements AFL's seed selection strategy with an adaptive strategy that prioritizes the seeds according to a quantitative measure (i.e., a distance) on how far a seed is from covering

```
1  pragma solidity ^0.4.20;
2  contract multiple_objective_function {
3    function foo(int x) {
4      int y = x*x + 10;
5      if(y == 110) { ... }
6      if(y == 10010) { ... } } }
```

Figure 2: An example with multiple objective functions

any just-missed branch. For this example, the distance for covering the just-missed branch (i.e., the then-branch) is computed as: $|msg.value - 100| + 1$, based on the value of $msg.value$ when the branch at line 10 is reached in the test case. Intuitively, the smaller the distance is, the closer the test case is to cover the branch (i.e., with a $msg.value$ closer to 100). In particular, when $msg.value$ is exactly 100, the distance value reaches the minimum value of 1. Based on this measurement, sFuzz iteratively selects *seeds* which gradually gets closer and closer to satisfying the condition at line 10. In our experiment, after 140 generations, sFuzz generates a test case which covers the branch, and reveals the vulnerability.

The above example shows a simplistic situation where there is only one just-missed branch. In general, there may be multiple just-missed branches and thus sFuzz measures a distance for each pair of test case and just-missed branch, i.e., how far is the branch from being covered by the test case. Then for each just-missed branch, sFuzz selects the test case with the minimum distance as the *seed*. For instance, the contract in Figure 2 shows a function which performs some basic arithmetic operations. There are two different branches, i.e., the condition at line 5 for comparing y with 110 and the one at line 6 for comparing y with 10010. Assume that both then-branches are yet to be covered. Given any test case, sFuzz computes two distances, one for covering the first then-branch; and the other for covering the second then-branch. Given a set of test cases, sFuzz selects, for each of these two branches, a test case which has minimum distance as seed, to generate further test cases. After repeating the process multiple times, sFuzz generates two test cases that cover the two then-branches. *We remark that for this example, due to the non-linear computation at line 4, approaches based on symbolic execution like Oyente [22] and teEther [21] are ineffective due to the limitation of underlying constraint solvers.*

3 FUZZING SMART CONTRACTS

In this section, we define our problem and then present our approach in detail step-by-step.

3.1 Problem Definition

A smart contract 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) $S = (N, i, E)$ where N is a finite set of control locations in the program; $i \in N$ is the initial control location, i.e., 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). Note that for simplicity, we define the smart contract as one single graph rather than defining one graph for each function and then connecting them through a call graph. A node in the graph is branching if and only if it has multiple child nodes and its outgoing edges are labeled with conditions. We refer to an outgoing edge of a branching node as a branch.

Test cases. A test case for S is a pair (σ_0, Σ) where σ_0 is a 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, v) 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), SB is a function which assigns an initial balance to each address and v is the initial valuation of the persistent state. $\Sigma = \langle m_0(\overrightarrow{p_0}), m_1(\overrightarrow{p_1}), \cdots \rangle$ 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.

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 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 non-determinism). Furthermore, a trace generated by t is a sequence of pairs of the form $\langle(\sigma_0, n_0), (\sigma_1, n_1), \cdots \rangle$ where $(n_0, n_1, \cdots)$ 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 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 (as we will explain in more detail in Section 4). *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

Algorithm 1: The test generation algorithm

```
1 let suite be an empty test suite;
2 let seeds := initPopulation();
3 while not time out do
4     add tests in seeds which covers new branches into suite;
5     let seeds := fitToSurvive(seeds);
6     let seeds = crossoverMutation(seeds);
7 return suites;
```

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 3.2.

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 [18, 22] including *Gasless Send*, *Exception Disorder*, *Timestamp Dependency*, *Block Number Dependency*, *Dangerous DelegateCall*, *Reentrancy*, *Integer Overflow/Underflow*, and *Freezing Ether*. We refer the readers to Section 4 for details.

3.2 Feedback-Guided Adaptive Fuzzing

The general idea of feedback-guided fuzzing is to transform the test generation problem into an optimization problem and use some form of feedback as an *objective function* in solving the optimization problem. Our fuzzing strategy is adaptive as we change the objective function adaptively based on the feedback. At the top level, sFuzz employs a genetic algorithm [5] which is inspired by the well-known AFL fuzzer to evolve the test suite in order to iteratively improve its branch coverage.

The overall workflow is shown in Algorithm 1. Variable *suite* is the test suite to be generated. It is initially empty. Whenever a test case covers a new branch, it is added into *suite*. Variable *seeds* is a set of seed test cases, based on which new test cases are generated. First, we generate an initial test suite using function *initPopulation*(). The loop from line 3 to 6 then iteratively evolves the test suite. In particular, we add those test cases in *seeds* which cover new branches (i.e., any branch which is not covered by test cases in *suite*) into *suite* at line 4. At line 5, we filter the test cases in *seeds* through function $fitToSurvive()$ so as to focus on those seeds which are more likely to lead to test cases covering new branches later. At line 6, function *crossoverMuatation*() generates more test cases based on the test cases in *seeds*. The loop continues until a pre-set time out is triggered. While Algorithm 1 resembles the one in AFL, the differences are in the details of each function. In the following, we present each function in detail.

Generating Initial Population Function *initPopulation*() generates an initial population containing multiple test cases. As mentioned above, to generate a test case, we need to generate an initial configuration σ_0 as well as a sequence of (public) function calls with concrete parameters. The initial configuration by default is as follows (in hexadecimal): $b = 0$, $ts = 0$, $SA = \{\texttt{0xf0}\}$, $SB = \{\texttt{0xff00...}\}$ and v is set using the declared initial value for each variable representing the persistent state. sFuzz additionally allows a user to customize the initial configuration, i.e., the user is allowed to provide an initial set of test cases.

Next, we generate multiple sequences of transactions, each of which is a function call with concrete parameters. For a contract with n functions, we generate n sequences. In each sequence, a different function is called once after the constructor is called. This makes sure that each function is tested at least once (i.e., function coverage is 100%).

For each function call, we generate a random value for each parameter based on its type. Note that if the parameter type has a fixed-length, e.g., of type *uint*256, this is straightforward. If the type does not have a fixed length (e.g., an array or a string), we first randomly generate a number (with a range from 0 to *bound* where *bound* is a bound on maximum length with a default value of 255) representing the number of elements in the parameter (e.g., number of characters) and then generate a corresponding number of element values.

Each test case is encoded in form of a bit vector. In the terminology of genetic algorithms, such bit vectors can be naturally regarded as *chromosomes*. The size of the bit vector equals to the number of bits for encoding the configuration plus the number of bits encoding the function calls. Note that for each test case, we keep a list of function calls (which always includes the constructor in the contract) and then encode each parameter value. If the parameter value is of variable-length, we use $\lceil \log bound \rceil$ (where *bound* is a bound on the maximum length with a default value of 255) to encode the length of the parameter value. For example, given the contract shown in Figure 1, (part of) the encoding of a test case is shown in Figure 3 where each part of encoding is labeled in the figure. It contains 192 bytes, of which the first 96 bytes are initial configuration and the last 96 bytes are a sequence of two function calls and the corresponding input parameters. As there are three string parameters, the first 3 bytes including `0x05`, `0x05` and `0x05` encode the length of `_response`, `_question` and `_answer` respectively. The remaining `0x05` values are used when there are more than 3 dynamic variables.

Before executing the test case, the bit vector is decoded to a test case according to our internally defined protocol. Note that the bits in the bit vector may be correlated with each other in multiple ways. For instance, the bits presenting the length of a variable-length value must be equal to the 'length' of the value.

Fitness After executing the *seeds* at line 4 in Algorithm 1, function $fitToSurvive()$ is called to evaluate the fitness of the *seeds* according to a fitness function. Note that the fitness function plays an extremely important role.

In sFuzz, we combine two complementary strategies. One is adopted from AFL, which works as follows. While *seeds* are executed, sFuzz monitors the execution and records the branches that each test case cover. A test case is deemed 'fit to survive' if it covers a new branch in the contract, e.g., a branch which is not covered by any test case in *suite*. This strategy has been shown to be effective in many settings [7] and indeed our experimental results show that it is effective in covering most of the branches (see Section 5).

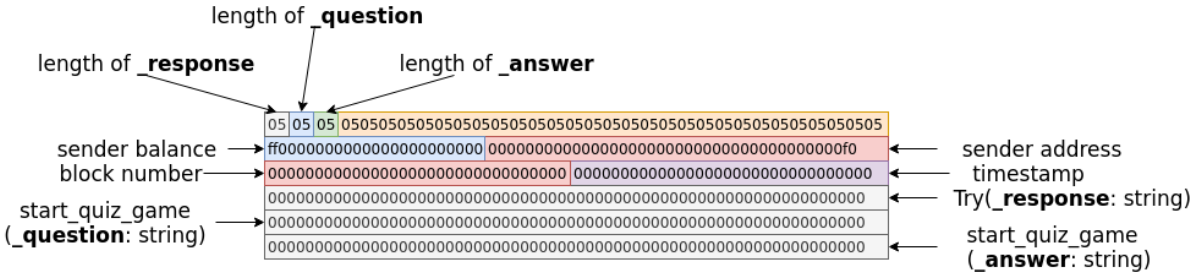

Figure 3: A generated test case

Although the AFL strategy allows us to quickly cover most of the branches, it often makes very slow progress in covering the remaining ones afterwards, i.e., often those branches which are with strict conditions. The reason is that most likely the randomly generated test cases would fail to satisfy the strict condition. In such a case, the above fitness function offers little feedback and guideline on how to generate new test cases. For instance, the probability of satisfying the second condition at line 10 of Figure 1 is as low as $\frac{1}{2^{256}}$ (if we assume that every value is equally likely to be generated). Intuitively, however, it is clear that a test input with $msg.value = 200$ is 'closer' to satisfy the condition than a test input with $msg.value = 10000000$. sFuzz thus integrates an adaptive strategy which selects *seeds* based on a quantitative measure on how far a test case is from covering any just-missed branch.

Let br_n be a just-missed branch in S, i.e., an uncovered outgoing edge from a branching node n in S and n has been covered. The idea is to define a function $distance(t, br_n)$ where t is a test case to return a quantitative measure on how far the branch br_n is from being covered by t.

Assume that br_n is labeled with a condition c. Note that c can be either $true$, $false$, $a == b$, a != b, $a >= b$, $a > b$, $a <= b$, or $a < b$ at the byte-code level where a and b are variables or constants. In our setting, since br_n is assumed to be a just-missed branch, c must not be $true$ (otherwise br_n must be covered already). Function $distance(t, br_n)$ is then defined as follows.

$$distance(t, br_n) = \begin{cases} K & \text{if } c \text{ is } false \\ |\ a - b\ | + K & \text{if } c \text{ is } a == b \\ K & \text{if } c \text{ is } a \text{ != } b \\ b - a + K & \text{if } c \text{ is } a >= b \text{ or } a > b \\ a - b + K & \text{if } c \text{ is } a <= b \text{ or } a < b \end{cases}$$

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

With the above, function $fitToSurvive(seeds)$ then selects the seeds as shown in Algorithm 2. The loop from line 2 to 4 goes through every test case to select those which cover a new branch. Afterwards, for each just-missed branch br_n in the smart contract, the loop from line 5 to line 11 selects a test case from *seeds* which is the closest to cover the branch according to $distance(t, br_n)$. Note that one seed is selected for each just-missed branch, which makes this algorithm a lightweight multi-objective optimization approach. All selected seeds are then used for crossover and mutation to generate more test cases in the next step. We refer the readers to Section 2 for an example.

Algorithm 2: Algorithm $fitToSurvive(seeds)$

```
1  let newSeeds be an empty set of test cases;
2  foreach seed in seeds do
3      if seed covers a new branch then
4          add seed into newSeeds;
5  foreach uncovered branches br_n do
6      let min be +∞; let t be a dummy test case;
7      foreach seed in seeds do
8          if distance(t, br_n) < min then
9              let min be dist(t, br_n);
10             let t be seed;
11     add t into newSeeds;
12 return newSeeds;
```

Remark The above-described strategy is inspired by search-based software testing (SBST) [16, 24] and yet it differs from SBST in several ways. The high-level reason for the difference is that having an AFL-based approach for fuzzing requires us to run test cases efficiently whereas existing SBST's seed selection strategy is time-consuming. Furthermore, due to the stack-based implementation of EVM, implementing existing the SBST strategy is infeasible. In the following, we present the differences in detail.

First, existing state-of-the-art SBST techniques (i.e., the one in EvoSuite [16]) measures how far a test case t is from covering any uncovered branch (not only those just-missed ones) in a more complicated way. That is, given CFG $S = (N, i, E)$, let the distance from a node n_1 to node n_2 to be the minimum number of edges along any path from n_1 to n_2. Let br_n be any uncovered branch and m be a node covered by t which is the nearest node to n, i.e., m has a minimum distance to n compared to any other node covered by t. SBST uses the following function to measure how far t is from covering br_n.

$$dist(t, br_n) = appr\_dist(t, br_n) + norm(distance(t, br_m))$$

where br_m is an outgoing edge of m which is along the shortest path from m to n. Note that if m is n (i.e., in case br_n is just-missed), br_m is simply br_n. Function $appr\_dist(t, br_n)$ is a measurement of how far branch br_n is from being covered by test case t, i.e., the distance from m to n plus 1. For instance, given a control flow graph as in Figure

4, if t covers only the edge $A \rightarrow B \rightarrow E$, $appr\_dist(t, C) = 1$ since there is one branch from B to reach C and there are two branches from A to reach C via D. Similarly, $appr\_dist(t, F) = 2$. Lastly, function $norm(x)$ is a normalization function which normalizes the results of $distance(t, br_m)$ to a value between 0 and 1. One such function is $norm(x) = 1 - 1.001^{-|x|}$ [16].

Applying the above strategy in fuzzing Solidity smart contracts is inefficient, if not infeasible, for multiple reasons. First, calculating $appr\_dist(t, br_n)$ would require us to construct the complete CFG. However, constructing the CFG based on bytecode only is highly nontrivial. In EVM, branches are realized with the opcode *jumpi*, with a value representing the target program counter dynamically at runtime. The only way to know the target is to fully simulate the stack, which is expensive. Second, even if we have the CFG, computing $appr\_dist(t, br_n)$ is still expensive. Given a CFG with K uncovered nodes. To maintain a list of 'best' test cases for each uncovered node, we have to calculate $appr\_dist(t, br_n)$ for all K uncovered nodes, i.e., by building a table of the shortest paths from all nodes to these K nodes. Furthermore, whenever a new node is covered, $appr\_dist(t, br_n)$ must be updated. The overhead is unreasonable given that efficiency is key for AFL-based fuzzing. By focusing on just-missed branches, sFuzz avoids both problems. That is, $appr\_dist(t, br_n)$ is always 1 for any just-missed branch br_n since node n must have been covered. Furthermore, because it is constant for any uncovered branch, we can simply skip it in $dist(t, br_n)$ and so that $dist(t, br_n)$ is reduced to $distance(t, br_n)$, without even the need to normalize. This further reduces the overhead.

Another key difference between sFuzz's strategy and existing SBST's is the multi-objective searching strategy. The multi-objective search strategies in existing SBST consider each uncovered branch as an objective and select Pareto-optimal seeds to evolve in next generation. Given a set of uncovered branch $\{b_1, b_2, ..., b_m\}$, a set of seeds $\{t_1, t_2, ..., t_n\}$, we say t_i is more Pareto-optimal than t_j if $\forall k \in 0..m$, $distance(t_i, b_k) < distance(t_j, b_k)$. Otherwise, we say that t_i and t_j are Pareto-equivalent. All Pareto-equivalent seeds form a Pareto frontier and the seeds can fall into several Pareto frontiers. Existing SBST selects the most Pareto-optimal seeds to evolve. A known problem for such a strategy [27] is that the number of seeds in the same Pareto frontier soars with the increase of the number of objectives (i.e., uncovered branches). For example, there could be hundreds of seeds in the most Pareto-optimal frontier with only 3-5 objectives, which makes it hard to select the most promising seeds and increases the runtime overhead. In contrast, sFuzz keeps one best seed for each just-missed branch (line 6–11 in Algorithm 2) and as a result, the number of seeds remains small (i.e., equivalent to the number of just-missed branches). Our experimental results show that such a strategy balances effectiveness in identifying good seeds and efficiency well.

3.3 Crossover and Mutation

Function *crossoverMutation*() generates new test cases based on those in *seeds* through crossover and mutation. sFuzz adopts all of the crossover strategies from AFL and introduces news ones specific for smart contracts. Furthermore, due to correlation between parameters of a test case, sFuzz additionally makes sure the generated test cases are valid. For instance, sFuzz (1) randomly chooses two test cases from *seeds*; (2) breaks the two test cases into two pieces at a selected position; and (3) swaps the second pieces to form two new test cases. Note that due to correlations between the bits representing a test case, there is no guarantee that the resultant test cases are valid and thus sFuzz always checks for validity and discard those invalid ones.

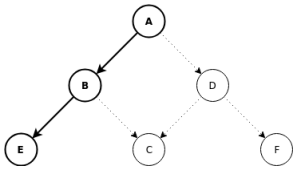

Figure 4: A control flow graph

Table 1: Mutations for fix-length values

Name
pruneMethodCall (new)
addMethodCall (new)
swapMethodCall (new)
singleWalkingBit, twoWalkingBit, fourWalkingBit 1/2/4 consecutive bits
singleWalkingByte, twoWalkingByte, fourWalkingByte
singleArith, twoArith, fourArith
singleInterest, twoInterest, fourInterest
overwriteWithDictionary
overwriteWithAddressDictionary

Mutation is another way of generating new test cases. Given a *seed* encoded in the form of a bit vector, sFuzz supports a set of mutation operators to generate new test cases. All mutation operators are shown in Table 1.

Recall that a test case is in the form of an initial configuration and a sequence of function calls with concrete parameters. The first three mutation operators aim to alter the sequence of function calls, by pruning a function call, adding a function call or swapping two function calls. When a function call is pruned (or added or swapped), the corresponding concrete parameters are pruned (or added or swapped) accordingly.

For those values in a test case other than those representing the called functions, sFuzz categorizes them into two groups. The first group contains those values which have fixed-length (e.g., a parameter of type $uint256$). sFuzz systematically applies the remaining mutation operators shown in Table 1 to generate new values, which are inspired by the mutation operators in AFL. Note that account addresses (and balances) are handled slightly differently (refer to the last row in the table) as there are special format requirements. Each address has 32 bytes, in which the last 20 bytes contain the address value and the first 12 bytes contain the balance of the address. For instance, the value `0xff00...00...00f0` represent an address `0xf0` with balance `0xff000000000000000000000000`.

The second group contains those values which have variable-length (e.g., a parameter of type *array*). For such values, their lengths are encoded as part of the test case as well. We thus first mutate the value representing the length in such a way that the result is a random value between 0 and 255 where 255 is an upper bound. If the new length is less than the current one, the corresponding value is shortened accordingly by pruning the additional bits. If

the length is more than the current one, random type-compatible values are padded accordingly.

Note that we discard identical test cases generated through either crossover or mutation. Furthermore, although we do not set a limit on the number of mutations generated from a test case, we apply multiple heuristics adopted from AFL to reduce the number of mutations. For instance, if applying the *WalkingByte* mutation to a block of 32 bytes does not result in any test case which covers a new branch, in the next stages sFuzz will not mutate that block. We refer the readers to AFL for details on these heuristics [7].

4 IMPLEMENTATION

sFuzz is implemented in C++ with an estimated 4347 lines of code. It is publically available (https://sfuzz.github.io). It has 3 main components: *runner*, *libfuzzer* and *liboracles*.

Component *runner* manages the execution of the test cases. sFuzz takes as input the bytecode of a smart contract along with the ABI (i.e., application binary interface, which can be generated automatically using existing tools) of the contract. The *runner* then generates a bash script file which contains a list of commands to analyze the ABI, and set options for the other two components.

The *runner* sets up a test network based on which smart contracts are deployed and transactions are executed. To generate test cases for functions with address-type parameters, sFuzz deploys a pool of externally owned accounts in the test network with random balances. The pool size is less than or equal to the number of address-type parameters because it is possible to set the same address to multiple address-type parameters. The values for address-type parameters are then chosen randomly from this pool. In addition, sFuzz deploys two special smart contracts as attackers, i.e., a *normal attacker* and a *reentrancy attacker*. Each attacker is set as the owner of the contract under test in turn. The *normal attacker* throws an exception whenever other contracts call its payable fallback function. The *reentrancy attacker* calls back the function which makes a call to its payable fallback function. If the attacker fails to call back, it acts as a *normal attacker*. Note that the *reentrancy attacker* is only loaded to detect *Reentrancy* vulnerability. Otherwise, the *normal attacker* is loaded to avoid call loops of *Reentrancy Attacker* which significantly reduces the speed of sFuzz.

Component *libfuzzer* solves the test generation problem, i.e., how to selectively generate test cases, by implementing the fuzzing strategy presented in the previous sections. It is responsible for multiple tasks.

First, it constructs the CFG of the given smart contract on-the-fly. Ideally, we would like to construct the CFG statically before fuzzing. However, constructing the CFG based on bytecode only is highly nontrivial. In EVM, branches are realized with the opcode *jumpi*, with a value representing the target program counter dynamically at runtime. The only way to know the target is to fully simulate the stack, which is expensive. Therefore, sFuzz constructs the CFG on-the-fly while fuzzing. That is, whenever the opcode *jumpi* is executed, the two destinations are recorded. If these two destinations are not part of the CFG yet, two new nodes are created accordingly representing the two destinations in the CFG.

Second, component *libfuzzer* implements the fuzzing algorithm discussed in Section 3. One optimization is that we identify *view* functions (i.e., those which do not change any variables) and exclude them from test case generation. The justification is that these view functions do not change the states and having them does not additionaly expose those vulnerabilities sFuzz targets at (see below). Note that view functions are marked by `view`, `pure` or `constant` keywords, sFuzz reads ABI file to recognize them.

Component *liboracles* solves the oracle problem, i.e., it monitors the execution of a test case and checks whether there is a vulnerability according to an extensible library of oracles used in sFuzz. sFuzz monitors the execution of test cases through the hooking mechanism supported by EVM. Whenever EVM executes an opcode, it creates an event containing read-only execution information, such as the values of the stack, memory, program counter, and the current executed opcode. sFuzz monitors these events for constructing the CFG and computing $distance(t, br_n)$, as well as logs the events for vulnerability detection. To reduce the execution overhead, vulnerability detection is conducted offline in batches (i.e., once for every 500 test cases). This design allows sFuzz to easily support different versions of Solidity, i.e., by simply replacing the EVM packed in sFuzz.

sFuzz has an extensible architecture which allows it to easily support different oracles as well. Currently, sFuzz supports 8 oracles inspired by the previous work [18, 22]. Since these oracles are not our main contribution, we refer the readers to [18, 22] for details.

These oracles are checked based on the logs of test cases. For instance, to check if a test case expose the *Gasless Send* vulnerability, we check that whether test case executes a *CALL* instruction with some data greater than 0 when the gas is equal to 2300. The test cases that expose vulnerabilities in the contract are kept in a separate test suite and reported to the user together with the vulnerabilities that they expose. Note that by design, sFuzz always reports true positives according to our definition of vulnerability except in the case of *Freezing Ether*. However, in practice, a reported vulnerability might be a false positive as it may be what the user intended (i.e., our definition of vulnerability is too strict). In the case of *Freezing Ether*, the identified 'warning' might be a false positive if there exist some test cases which call $send()$ or $transfer()$ but such test cases are never generated. Technically, the problem of checking whether there is *Freezing Ether* vulnerability can only be solved if we cover all feasible opcode (which is often infeasible).

5 EXPERIMENTS AND EVALUATION

In this section, we evaluate sFuzz through multiple experiments. The experiments are designed to answer the following research questions (RQ).

- *RQ1: How efficient is sFuzz?*
- *RQ2: Is sFuzz effective in finding smart contract vulnerabilities and obtaining high code coverage?*
- *RQ3: Is the adaptive strategy useful?*

Our test subjects include 4112 smart contracts which we collect from EtherScan [4]. These contracts are implemented using Solidity 4.2.24, which is the most popular version of Solidity. Moreover, the source code for these contracts are available, which makes

the evaluation more accurate. We note that sFuzz can run with bytecode only. For a baseline comparison, we compare sFuzz with a fuzzer named ContractFuzzer reported in [15] and a symbolic execution tool named Oyente reported in [22]. Other fuzzers for smart contracts have been mentioned in [21]. However, we fail to find the reported tools online or through the authors. We run the experiments 3 times and report the average as the result. All experimental results reported below are obtained on an Ubuntu 18.04.1 LTS machine with Intel Core i7 and 16GB of memory. We use the default initial configuration as presented in Section 3.2.

5.1 Efficiency

To answer RQ1, we systematically apply sFuzz, ContractFuzzer and Oyente on all 4112 smart contracts. To save time, each contract is run for 2 minute in this experiment. Note that in general the adaptive fuzzing strategy takes time to show its effectiveness (as we will show later) and thus this setting gives an edge to other tools.

We measure the efficiency of sFuzz by counting how many test cases are generated and executed per second. Naturally, a test case for a more complicated contract (e.g., with many loop iterations) takes more time to execute. Thus, we show how efficiency varies for different contracts. Figure 5 summarizes the result, where each bar represents 10% (about 400) of the fuzzed contracts and the y-axis shows the number of test cases generated and executed per second. The contracts are sorted according to how efficiently it can be fuzzed. From the figure, we observe that the efficiency varies significantly over different contracts, i.e., sFuzz generates and executes more than 989 test cases per second on average for the top 10% of the contracts, and less than 14 test cases for the bottom 20%. On average, sFuzz generates and executes more than 208 test cases per second.

Figure 5 also compares the efficiency of sFuzz with Oyente and ContractFuzzer. From the results, we observe that sFuzz is significantly more efficient than other tools. On average, ContractFuzzer and Oyente generate and execute 0.1 and 16 test cases per second respectively. There are multiple reasons why sFuzz is much faster. First, ContractFuzzer simulates the whole network and manages the blockchain (e.g., commit state changes to storage and append new mined blocks to blockchain after function calls), whereas sFuzz simulates only details of network or blockchain which are relevant to vulnerabilities in smart contracts. Second, sFuzz has a highly optimized implementation in C++, whereas ContractFuzzer is based on Node.js and Go language. In the case of Oyente, because it is a symbolic execution tool, Oyente is expected to run slower than a fuzzer like sFuzz.

We further conduct an experiment to measure the overhead of monitoring the execution of a test case (using the hooking mechanism) and the overall overhead of the fuzzing process (including the overall of monitoring the execution, constructing the CFG, mutating the test cases and comparing them, etc.). We apply sFuzz to a set of 60 randomly selected contracts and measure the time spent on executing the test cases, monitoring the execution and other steps of the fuzzing process. The results show that on average the monitoring consumes about 10% of the total execution time and the overhead of the fuzzing process (including monitoring) is about 14%. This is very efficient compared to the reported overhead in other fuzzers [32].

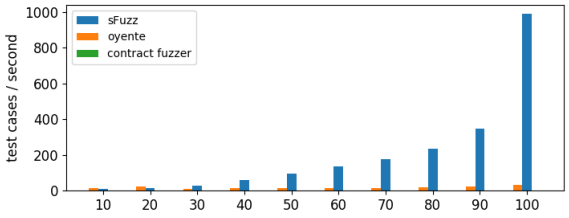

Figure 5: Efficiency comparison between sFuzz, Oyente, and ContractFuzzer

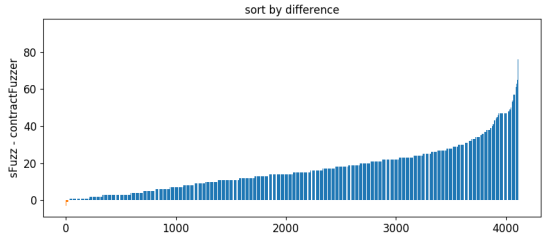

Figure 6: Coverage comparison between sFuzz and ContractFuzzer

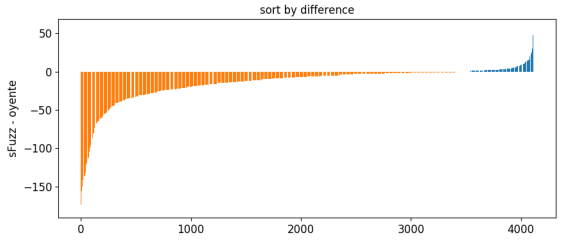

Figure 7: Coverage comparison between sFuzz and Oyente

5.2 Effectiveness

To answer RQ2, we aim to measure the branch coverage achieved by the test suite generated for each smart contract, as well as count the number of vulnerabilities identified. However, measuring branch coverage precisely is highly non-trivial due to, for instance, the problem of infeasible branches. Thus, we instead measure the number of distinct branches covered by the generated test suite. Figure 6 summarizes a comparison between sFuzz and ContractFuzzer in terms of the number of distinct branches covered. The y-axis is the number of branches covered by sFuzz minus that of ContractFuzzer and each point on the x-axis represents a smart contract. The contracts are sorted by their y-axis value. Similarly, Figure 7 shows the comparison between sFuzz and Oyente.

For most of the smart contracts (i.e., 4077 of 4112 contracts) sFuzz covers more branches than ContractFuzzer. To our surprise, ContractFuzzer managed to cover more branches for 35 contracts.

```
1  contract A {
2    mapping(address => uint) balances;
3    uint id = 10;
4    function main(uint x, uint y) {
5      if (id == 9) {
6        if (balances[msg.sender] > 10) {
7          uint sum = x + y; } } } }
```

Figure 8: Oyente visits infeasible branches

A closer investigation shows that the number of branches covered by ContractFuzzer is inflated for the following reasons. First, as sFuzz does not execute *view* functions (for efficiency reasons), all branches in these functions are not counted. Because *view* functions do not modify the state of a smart contract, they are considered irrelevant to vulnerabilities. Second, ContractFuzzer sometimes generates invalid test cases which fail mandatory constraints and cover additional branches. Mandatory constraints are generated by the compiler (i.e., the Solidity compiler) and are embedded in the bytecode to assert the correctness logic of function calls or data types. For example, ContractFuzzer invokes a *fallback* function of a non-fallback contract or sends Ethereum to functions which are not marked with the *payable* keyword. As a result, the mandatory constraints are failed which lead to branches which signal an error in the test case being covered.

In the case of Oyente, in 3402 contracts, Oyente covers more branches than sFuzz. An investigation shows that Oyente analyzes every function separately and thus has to assume that state variables can take arbitrary values (without considering their initial values or constraints on how the values are updated). As a result, Oyente can easily satisfy almost all conditions in smart contracts. Given the sample contract A in Figure 8, Oyente covers 99.1% EVM code and discovers an integer overflow vulnerability. It means that these conditions: $id == 9$ and $balances[msg.sender] > 10$ are satisfied. However, it is impossible as there is no way to change values of id and $balances[msg.sender]$. Often, a condition in smart contract is the comparison between local/parameter variables and state variables, e.g., $balances[msg.sender] > value$ (whether sender has enough Ethereum to deduce). In such cases, sFuzz must call the function which sets certain values to the state variables before satisfying them whereas Oyente assigns arbitrary values directly to state variables. It is apparent to us that Oyente's approach is flawed and would 'cover' many infeasible paths.

In the following, we summarize the number of vulnerable contracts discovered by sFuzz in each category. The results are shown in Table 2. The first column shows the type of vulnerability. The next three columns show the number of vulnerable contracts found by sFuzz, ContractFuzzer and Oyente respectively. The sub-column # show the number of contracts that have the vulnerability according to each vulnerability type and the second sub-column is the percentage of true positives of the identified vulnerabilities. For all categories, sFuzz finds more vulnerable contracts than ContractFuzzer. Note that ContractFuzzer removes *Freezing Ether* from their source code and does not check *Integer Overflow/Underflow*. In total, sFuzz finds vulnerabilities in 1113 contracts, i.e., 24 times more than that of ContractFuzzer.

Table 2: Vulnerabilities

Vulnerability Type	sFuzz		ContractFuzzer		Oyente	
	#	true posi.	#	true posi.	#	true posi.
Gasless Send	764	100%	14	100%	0	N.A.
Exception Disorder	36	100%	6	100%	0	N.A.
Reentrancy	29	100%	3	100%	52	60%
Timestamp Dependency	243	86%	28	86%	102	100%
Block Number Dependency	59	80%	16	95%	0	N.A.
Dangerous DelegateCall	17	100%	0	100%	0	N.A.
Integer Overflow	98	100%	0	N.A.	3350	60%
Integer Underflow	224	80%	0	N.A.	2246	60%
Freezing Ether	15	60%	0	N.A.	0	N.A.

To evaluate the soundness of sFuzz, we manually examine the identified vulnerable contracts to check whether they are true positives or not. However, we are unable to manually check all the identified vulnerability for two reasons. First, there is an overwhelming number of vulnerabilities. Instead, we randomly sample 50 vulnerable contracts with source code in each category and manually check whether the identified vulnerability is a true positive or not. If there are fewer than 50 vulnerable contracts with source code in the category, we check all of them.

For *Gasless Send*, *Exception Disorder* and *Reentrancy* vulnerability, all 50 sampled vulnerable contracts are true positives. For *Timestamp Dependency*, out of the 50 sampled vulnerable contracts, 43 of them are true positives. In the remaining 7 contracts, although *block.timestamp* and/or *now* is used in a condition, they are irrelevant to the Ether sending part (i.e., no control/data dependency). Rather their values are saved in global variables to record the creation time of specific events. sFuzz mistakenly claims that such cases are vulnerable. For *Block Number Dependency*, 40 out of the 50 sampled vulnerable contracts are true positives. Similarly, the reason for the 10 false positives is the value of *block.number* is assigned to global variables but they are irrelevant to Ether sending process. For *Dangerous DelegateCall*, all 17 sampled contracts are indeed vulnerable. Similarly so for *Integer Overflow*. For *Integer Underflow*, 40 of the 50 identified contracts are indeed vulnerable. The reason for the 10 false positives is because it is non-trivial to identify the correct type of a variable based on bytecode only (e.g., whether it is *uint256* or *uint128*), sFuzz conservatively assumes that all arithmetic operations returning a negative value may be vulnerable. This can be improved by adopting the approach in [29] to infer types based on EVM bytecode. Lastly, for *Freezing Ether*, 9 of the 15 identified contracts are true positives. The reason for the 6 false positives is that although there is a program path which allows the contract to send Ether, the program path is not covered and sFuzz falsely assumes that there is no such program path. This percentage of such false positives is expected to be reduced if sFuzz is applied for a longer time (with more branches covered).

The last column in Table 2 shows the results of Oyente. The results should be taken with a grain of salt since Oyente requires the source code. For instance, it is trivial to know the type of variables with the source code, and thus Oyente identifies many more problems with *Integer Overflow/Underflow*. For the remaining vulnerabilities, Oyente does not support 5 of them; identifies a higher number of vulnerable contracts for *Reentrancy* but with a higher false positive rate; and identifies much fewer vulnerable contracts for *Timestamp Dependency*.

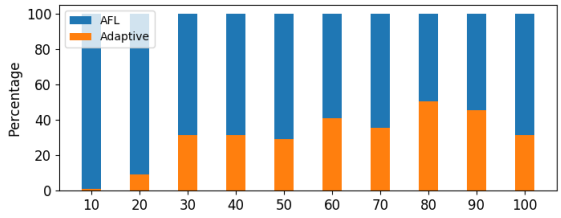

Figure 9: Percentage of test cases due to adaptive strategy

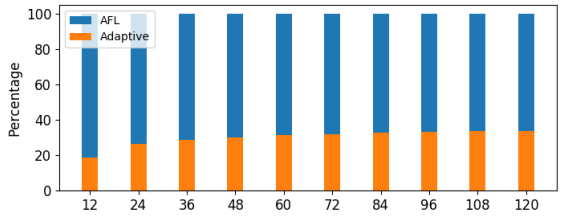

Figure 10: Effective of adaptive strategy over time

5.3 Adaptiveness

To answer RQ3, we systematically analyze the test suite generated by sFuzz for each smart contract. Note that each test case covers at least one branch which is not covered by any other test cases. To measure how the two fuzzing strategies implemented in sFuzz complement each other, we count how many test cases in the resultant test suites are generated due to the AFL strategy and how many are due to the adaptive strategy. Note that a test case is judged to be due to the adaptive strategy if and only if it is generated based on a seed selected by line 11 at Algorithm 2.

The results are shown in Figure 9, where the y-axis is the percentage of test cases generated by the strategy. Each bar represents 10% of the contracts. We remark that the two strategies have different targets and thus whether they are effective largely depends on what branching conditions are in the smart contracts. We thus sort the contracts according to the speed of sFuzz. The bar on the rightmost thus represents the top 10% contracts. We observe that, as expected, the AFL strategy easily covers most of the branches (since the conditions for executing most branches are not strict). For about 80% of the smart contracts, the adaptive strategy makes a noticeable contribution, i.e., contributing an average of 31% of the generated test cases. Given that sFuzz is applied for each contract only for 2 minutes, the result is encouraging as we hypothesize that the effect of the adaptive strategy would be more apparent if sFuzz is applied for a longer period of time.

To test our hypothesis, we record the percentage of test cases generated by the adaptive strategy every 12 seconds. The results are shown in Figure 10, where the x-axis is the fuzzing time and each bar shows the percentage after certain number of seconds. We can observe that the percentage of generated test cases by adaptive strategy increases with more fuzzing time. On average, the percentage rises from 18% after 12 seconds fuzzing to 33% after 2 minutes fuzzing. From the results, we conclude the adaptive strategy is useful in increasing the coverage of the generated test suites.

Threat to validity There are both internal threats and external threats to our work. For external threats, it is probable that sFuzz's performance will vary with the choice of the initial population, as other researchers have noted [20]. For internal threats, the percentage of true positives in Table 2 may not be accurate as they are approximated by a sample of 50 contracts for each type of vulnerability. In addition, the exact intention of the author of the contract is not always clear, even if we try our best to read the source code.

6 RELATED WORK AND CONCLUSION

sFuzz is closely related to existing fuzzers for smart contracts. ContractFuzzer [18] is a fuzzer which can check 7 different types of vulnerabilities. Its approach, however, does not use any feedback to improve the test suite. Echidna [3] is another fuzzer that is reportedly capable of checking if the contract violates some user-defined properties. However, we fail to find any publication about it.

sFuzz is complementary to existing symbolic execution engines for smart contracts. In [22], Luu *et al.* presented an engine to find potential security bugs in smart contracts. The tool, however, is neither sound nor complete. In [21], Krupp and Rossow presented teEther, which is focused on financial transactions and related vulnerabilities. In [25], Nikolic *et al.* presented a tool named MAIAN, which can find 3 types of trace vulnerabilities. In [29], Torres *et al.* presented Osiris, a tool which combines symbolic execution and taint analysis to discover 3 types of integer bugs in smart contracts. Different from the above works, sFuzz is a fuzzer and it can be combined with the above engines to form a hybrid fuzzing engine.

sFuzz is related to work on formal verification of smart contracts. Zeus [19] is a framework which verifies the correctness and fairness of smart contracts based on LLVM. Bhargavan *et al.* proposed a framework to verify smart contracts formally by transforming the source code and the bytecode to F*, a language designed for verification [9]. In [17], the author presented an attempt to verify the Deed contract using Isabelle/HOL [26].

sFuzz is broadly related to work on analyzing smart contracts. In [13], Delmolino *et al.* showed that writing a safe smart contract is not a trivial task. In [8], Atzei *et al.* provided a taxonomy for common vulnerabilities in smart contracts with real-world attacks. In [14], the authors performed a call graph analysis and showed that only 40% of smart contracts are truthless as their control flows are immutable. In [10], Chen *et al.* presented 7 gas-cost programming patterns and showed that most of the contracts suffer from these gas-cost patterns.

To conclude, in this work, we present sFuzz, an adaptive fuzzing engine for EVM smart contracts. Experimental results show that sFuzz is significantly more reliable, faster, and more effective than existing fuzzers. sFuzz is currently under rapid development and has already gained interest from multiple companies and research organizations.

ACKNOWLEDGMENTS

This research was supported by the Singapore Ministry of Education (MOE) Acemedic Research Fund (AcRF) Tier 1 grant.

REFERENCES

[1] [n. d.]. A Next-Generation Smart Contract and Decentralized Application Platform. ([n. d.]). Retrieved Feb 2020 from https://github.com/ethereum/wiki/wiki/White-Paper

[2] [n. d.]. Aleth - Ethereum C++ client, tools and libraries. ([n. d.]). Retrieved Feb 2020 from https://github.com/ethereum/aleth/

[3] [n. d.]. Echidna. https://github.com/crytic/echidna/. ([n. d.]).

[4] [n. d.]. Etherscan. ([n. d.]). Retrieved Feb 2020 from https://etherscan.io/

[5] [n. d.]. Genetic algorithm. https://en.wikipedia.org/wiki/Genetic_algorithm. ([n. d.]).

[6] [n. d.]. Solidity. ([n. d.]). Retrieved Feb 2020 from https://solidity.readthedocs.io/

[7] [n. d.]. Technical "whitepaper" for afl-fuzz. ([n. d.]). Retrieved Feb 2020 from http://lcamtuf.coredump.cx/afl/technical_details.txt

[8] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2016. A survey of attacks on Ethereum smart contracts. *IACR Cryptology ePrint Archive* 2016 (2016), 1007.

[9] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. ACM, 91–96.

[10] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.

[11] Christopher D Clack, Vikram A Bakshi, and Lee Braine. 2016. Smart contract templates: foundations, design landscape and research directions. *arXiv preprint arXiv:1608.00771* (2016).

[12] Phil Daian. [n. d.]. Analysis of the DAO exploit. ([n. d.]). Retrieved Feb 2020 from https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[13] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International Conference on Financial Cryptography and Data Security*. Springer, 79–94.

[14] Michael Fröwis and Rainer Böhme. 2017. In Code We Trust? In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 357–372.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 213–223. https://doi.org/10.1145/1065010.1065036

[16] Mark Harman and Phil McMinn. 2010. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering* 36, 2 (2010), 226–247.

[17] Yoichi Hirai. 2016. Formal verification of Deed contract in Ethereum name service. *November-2016.[Online]. Available: https://yoichihirai. com/deed. pdf* (2016).

[18] 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 (ASE 2018)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[19] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium (NDSS'18)*.

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2123–2138.

[21] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 1317–1333. https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[22] 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*. ACM, 254–269.

[23] Bill Marino and Ari Juels. 2016. Setting standards for altering and undoing smart contracts. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*. Springer, 151–166.

[24] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software testing, Verification and reliability* 14, 2 (2004), 105–156.

[25] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 653–663.

[26] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

[27] A. Panichella, F. M. Kifetew, and P. Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158.

[28] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First Monday* 2, 9 (1997).

[29] Christof Ferreira Torres, Julian Schütte, et al. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 664–676.

[30] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 291–302.

[31] Gavin Wood. [n. d.]. Ethereum: A Secure Decentralised Generalised Transaction Ledger. ([n. d.]). Retrieved Feb 2020 from https://ethereum.github.io/yellowpaper/paper.pdf

[32] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

[33] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, and Huaimin Wang. 2016. Blockchain challenges and opportunities: A survey. *Work Pap.–2016* (2016).