**sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts**

**sFuzz：一种有效的Solidity智能合约自适应模糊器**

**Tai D. Nguyen**

**Singapore Management University,**

**[dtnguyen.2019@smu.edu.sg](mailto:dtnguyen.2019@smu.edu.sg)**

**Long H**

**Singapore Management University,**

**[longph1989@gmail.com](mailto:longph1989@gmail.com)**

**Pham Jun Sun**

**Singapore Management University,**

**[sunjunhqq@gmail.com](mailto:sunjunhqq@gmail.com)**

**Yun Lin**

**National University of Singapore,**

**Singapore [llmhyy@gmail.com](mailto:llmhyy@gmail.com)**

**Quang Tran Minh**

**Ho Chi Minh City University of**

**Technology, Vietnam [quangtran@hcmut.edu.vn](mailto:quangtran@hcmut.edu.vn)**

# ABSTRACT

# 摘要

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.

智能合约是在区块链基础设施上执行的图灵完备程序，通常用于管理有价值的数字资产。Solidity是在以太坊平台上编写智能合约最流行的编程语言之一。与传统程序一样，智能合约可能包含漏洞。与传统程序不同的是，智能合约一旦部署就无法轻松修补。因此，在部署之前彻底测试智能合约非常重要。在这项工作中，我们提出了一种在以太坊平台上实现智能合约的自适应模糊器sFuzz。与现有的实体模糊器相比，sFuzz结合了AFL模糊器中的策略和一种针对难以覆盖分支的轻量级多目标自适应策略。sFuzz已经应用于超过4000个智能合约，实验结果表明：（1）sFuzz是有效的，比最先进的工具快两个数量级；（2）sFuzz在实现高代码覆盖率和发现漏洞方面是有效的；sFuzz中不同的模糊化策略是相辅相成的。

# 1 INTRODUCTION

# 1简介

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].

如今，智能合约[11，28]被实现为在区块链基础设施上执行的图灵完备程序[33]。它提供了一个框架，允许任何程序（相当于合约）以自治、分布式和可信的方式执行。因此，智能合约有可能彻底改变许多行业。智能合约的流行应用包括众筹、网上赌博等。以太坊[1，31]是第一个引入智能合约功能的平台。基于以太坊平台平台，Solidity成为智能合约最流行的编程语言[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].

与传统的C或Java程序一样，智能合约可能包含漏洞。与传统程序不同的是，智能合约一旦部署到区块链上就不容易修改[23]。因此，一个漏洞使得智能合约永远易受攻击，这大大放大了问题。近年来，关于利用以太坊智能合约中的安全漏洞进行攻击的新闻报道越来越多。一个特别引人注目的例子是DAO攻击[12]，即攻击者利用DAO合约中的漏洞窃取了350多万以太币（相当于当时的4500万美元）。为了修复这个漏洞，推出了一个硬分叉，不仅价格昂贵，而且引起了很大争议[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:

因此，最好在部署智能合约之前，开发用于验证智能合约的工具，以识别漏洞。在一系列用于验证智能合约的互补技术中，我们将重点放在智能合约的自动化测试上，因为测试通常是最便宜的，因此也是最适用的。要自动测试智能合约，必须解决以下三个问题：

• the test automation problem (i.e., how to run test cases),

•测试自动化问题（即，如何运行测试用例），

• the test generation problem (i.e., what to test),

•测试生成问题（即，测试什么）

• and the oracle problem (i.e., what are vulnerabilities).

•预言机问题（即，什么是漏洞）

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.

在文献中，已经开发了几种自动测试智能合约的方法，每种方法都以稍微不同的方式回答这三个问题。例如，ContractFuzzer[18]使用预先部署的合约构建一个网络，并生成运行智能合约的交易，基于一组预定义的参数值生成测试用例，并针对一组特定于智能合约的预言机。Oyente[22]通过符号执行象征性地运行智能合约，通过约束求解生成覆盖单个函数中不同程序路径的测试用例，支持多个预言机识别4种漏洞。teEther[21]同样应用符号执行来生成覆盖程序路径的测试用例，并将重点放在与金融交易相关的预言机上。

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 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.

在这项工作中，我们提出了一个在以太坊上运行的智能合约的全自动测试引擎sFuzz。sFuzz的灵感来自AFL[7]，一个著名的C程序模糊器，也就是说，sFuzz是一个反馈引导的模糊引擎，并且应用成本低廉。sFuzz补充了现有的基于符号执行的测试引擎，比如Oyente和teEther，因为众所周知模糊化和符号执行是互补的[30，32]。虽然基于AFL的模糊处理通常是有效的，但它也有其局限性，即在覆盖严格条件保护的分支时往往代价高昂。为了解决这一问题，sFuzz将基于AFL的模糊技术与一种有效的轻量级自适应种子选择策略相结合。尽管受到基于搜索的软件测试的启发[16，24]，后者的独特之处在于它有一个轻量级的目标函数（设计时考虑了程序的稳定性特点）和一个新颖的多目标优化策略。

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.

SFuzz是基于Aleth[2 ]（即C++编写的以太坊VM）构建的，具有类似于AFL的系统结构，可扩展到不同的以太坊 VMS和预言机以及模糊策略。sFuzz已经系统地应用于超过4000个智能合约。实验结果表明，sFuzz平均比ContractFuzzer快两个数量级以上，覆盖更多的分支，暴露出更多的漏洞。对sFuzz和Oyente的比较表明它们是互补的。此外，通过延长模糊时间的实验表明，自适应策略提高了代码覆盖率。sFuzz可以在线获得，已经被多家公司采用。

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.

论文的其余部分组织如下。第2节通过示例说明了sFuzz是如何工作的。第3节介绍了该方法的细节。第4节展示了sFuzz的实现细节。第5节报告评估结果。第6节回顾相关工作并总结。

# 2 ILLUSTRATIVE EXAMPLES

# 2举例说明

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.

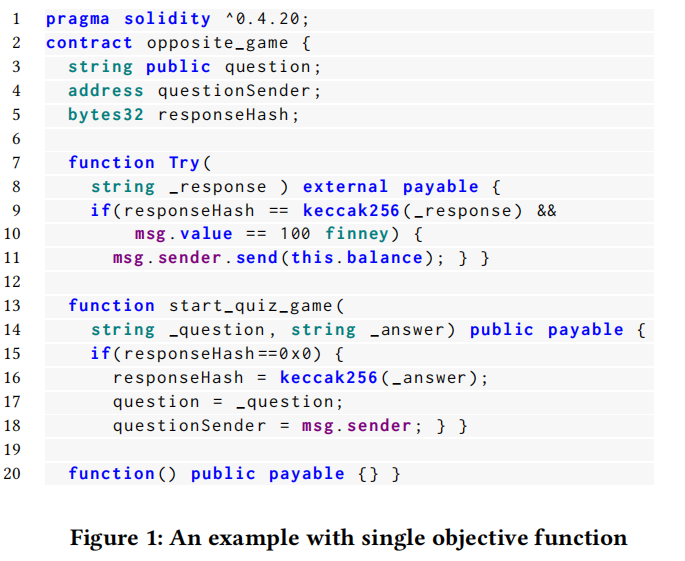
在本节中，我们将通过两个示例逐步说明sFuzz的工作原理。请注意，为了简单起见，显示了这两个示例的Solidity源代码。sFuzz只需要EVM（即以太坊虚拟机）字节码[1，31]来模糊化智能合约。

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.

给定一个智能合约，sFuzz会自动配置一个区块链网络，部署智能合约，并生成多个交易，每个交易调用合约中的一个函数。然后使用EVM执行交易，EVM中包含一组用于识别漏洞的预言机。sFuzz监视交易的执行，以收集某些反馈，例如，某个分支是否已被覆盖，以及该分支的覆盖程度。无论何时发现漏洞，都会保存交易和网络配置（即测试用例），并在稍后向用户报告。否则，根据特定的种子选择标准，根据交易执行期间收集的反馈选择一些测试用例作为种子。之后，种子经过变异，生成下一代测试用例。此过程反复，直到发生超时。

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 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 response Hash variable. The user then calls the try function with their answer as the argument and pays a fee of 100 finney (which[[1]](https://fanyi.baidu.com/" \l "_ftn1" \o ") is a unit of the token) for each try. If the answer is correct, a reward is sent to the user.

在下面，我们将描述sFuzz如何使用图1所示的合约。合约实现了一个简单的智力竞赛游戏。本合约以start\_quiz\_game合约为基础，为简单起见，稍作修改。可以通过调用函数start\_quick\_game（）来创建测验。然后保存在responseHash变量中。然后，用户调用try函数并以其答案作为参数，并支付100 finney（这是[[1]](https://fanyi.baidu.com/" \l "_ftn1" \o ")是代币的单位）。如果答案正确，将向用户发送奖励。

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.

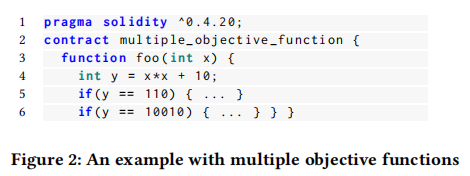
当执行第11行并调用代价高昂的回调函数时，此合约会遇到一个称为Gasless Send的漏洞。也就是说，当第11行的函数send（）被执行时，如果接收者是一个合约，那么它的回调函数将自动执行。因为函数send（）只转发2300个单位的gas（即，执行函数的代价），所以如果回调函数代价高昂（例如，花费超过2300个单位的gas），则抛出out-of-gas异常。在这种情况下，send（）函数只返回false，因为返回的值没有进行相应的检查和处理，所以合约的所有者可以自己保留报酬。

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 keccak256(\_answer) at line 16 and is compared to keccak256(\_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 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.

要暴露此漏洞，首先将网络配置为具有多个地址和相应的余额。然后在其中一个地址部署此合约。此外，攻击者会自动部署具有代价高昂的回调功能的合约。为了暴露该漏洞，具有这种网络配置的测试用例（即一系列交易）必须首先调用函数start\_ quick\_game，然后使用参数调用函数Try，以满足第9行和第10行函数Try中的所有2个条件。第9行的条件满足一个测试用例，该用例将所有参数和合约变量设置为默认值0。请注意，responseHash在第16行设置为keccak256（\_response），并与第9行的keccak256（\_response）进行比较。然而，通过随机生成的测试值来生成满足第二个条件的测试用例的可能性很小。变量msg.value的大小为32字节，因此我们只有概率生成值100（如果我们在所有可能值中生成均匀分布的随机值）。在这种情况下，AFL中现有的模糊策略也是无效的，即AFL选择覆盖新分支的测试用例作为种子。由于所有通过突变产生的测试用例不太可能覆盖第10行的分支，根据AFL种子选择策略，它们同样是“坏”的。

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

sFuzz用一种自适应策略来补充AFL的种子选择策略，该策略根据数量对种子进行优先排序测量种子离刚丢失的分支有多远的距离。对于本例，覆盖刚刚丢失的分支（即当时的分支）的距离计算为：| msg.value−100 |+1，基于测试用例中到达第10行分支时msg.value的值。直观地说，距离越小，测试用例覆盖分支的距离就越近（即msg.value接近100）。特别地，当msg.value正好为100时，距离值达到最小值1。基于这个测量，sFuzz迭代地选择种子，这些种子逐渐接近于满足第10行的条件。在我们的实验中，经过140代之后，sFuzz生成了一个覆盖分支的测试用例，并找出了漏洞。



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.

上面的例子显示了一个相对简单的情况，其中只有一个刚丢失的分支。一般来说，可能有多个刚刚丢失的分支，因此sFuzz测量每对测试用例和刚刚丢失的分支的距离，即，分支离测试用例覆盖的距离有多远。然后，对于每个刚刚丢失的分支，sFuzz选择具有最小距离的测试用例作为种子。例如，图2中的合约显示了一个执行一些基本算术运算的函数。有两个不同的分支，即第5行的条件用于比较y与110，第6行的条件用于比较y与10010。假设这两个分支都还没有被覆盖。给定任何测试用例，sFuzz计算两个距离，一个用于覆盖第一个分支，另一个用于覆盖第二个分支。给定一组测试用例，sFuzz为这两个分支中的每一个分支选择一个具有最小距离作为种子的测试用例，以生成进一步的测试用例。在多次重复这个过程之后，sFuzz生成了两个测试用例，覆盖了这两个分支。我们注意到，在这个例子中，由于第4行的非线性计算，基于符号执行的方法，如Oyente[22]和teEther[21]由于底层约束求解器的限制，是无效的。

# 3 FUZZING SMART CONTRACTS

# 3 模糊智能合约

## 3.1 Problem Definition

## 3.1 问题定义

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 ∈ N is the initial control location, i.e., the start of the contract; and E ⊆ N × C × 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.

智能合约通常有许多实例变量、一个构造函数和多个函数，其中一些是公共的。它可以等价地以控制流图的形式查看（CFG）S=（N，i，E），其中N是程序中的一组有限控制位置；i∈N是初始控制位置，即合约的开始；E⊆N×C×N是一组带标签的边，每个边的形式为（N，C，N），其中C是一个条件（对于if-then-else或while循环等条件分支）或一个命令（即。，任务）。请注意，为了简单起见，我们将智能合约定义为一个图，而不是为每个函数定义一个图，然后通过调用图将它们连接起来。图中的一个节点是分支的当且仅当它有多个子节点并且它的输出边用条件标记时。我们将分支节点的传出边称为分支。

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.

代码覆盖率。理想情况下，我们的目标是生成一个测试套件，以揭示合约中的所有漏洞。然而，由于我们不知道漏洞在哪里，我们必须以实现更加可衡量的目标为目标。在这项工作中，我们的答案是关注代码覆盖率，特别是分支覆盖率。我们注意到，我们的方法可以扩展到支持不同的覆盖率，而代价是额外的代码插装。当且仅当测试套件中有至少一次访问边缘的测试案例t时，S中的分支被测试套件覆盖。测试套件的分支覆盖率计算为覆盖的分支占分支总数的百分比。注意，在智能合约中静态地确定（可行的）分支的总数通常是不可行的，原因有两个。首先，一些分支可能是不可行的（即，不存在任何访问该分支的测试用例），并且知道一个分支是否可行是一个困难的问题。其次，EVM有一个基于堆栈的实现，这使得识别所有潜在可行的分支变得困难（我们将在第4节中详细解释）。因此，我们的问题被简化为生成一个最大化覆盖分支数量的测试套件。

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.

为了获得最大的代码覆盖率，一种方法是生成一个大型测试套件（例如，通过随机测试生成）。然而，在实践中，我们经常有有限的资源（在时间或计算机进程的数量方面），因此我们的问题被细化为“生成一个测试套件，尽可能有效地最大化覆盖分支的数量”。我们的解决方案是反馈引导自适应模糊化。

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 3.2.

模糊化是创建测试用例最流行的方法之一[20]。反馈引导模糊系统（又称fuzzer）将测试和初始测试套件作为输入，监控测试用例的执行以获得一定的反馈，以一定的方式基于现有的测试用例生成新的测试用例，然后重复该过程直到满足停止标准。我们将在第3.2节中详细介绍我们的反馈引导自适应模糊过程。

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.

预言机，剩下的问题是如何判断测试用例是否揭示了漏洞。在这项工作中，我们采用了以前的方法[18，22]中的一组预言，包括无gas发送、异常无序、时间戳依赖、块号依赖、危险的委托调用、重入、整数溢出/下溢和冻结。我们请读者参阅第4节了解详情。

## 3.2 Feedback-Guided Adaptive Fuzzing

## 3.2反馈引导自适应模糊

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 wellknown AFL fuzzer to evolve the test suite in order to iteratively improve its branch coverage.

反馈引导模糊化的基本思想是将测试生成问题转化为优化问题，并以某种形式的反馈作为优化问题的目标函数。我们的模糊策略是自适应的，因为我们根据反馈自适应地改变目标函数。在顶层，sFuzz采用了一种遗传算法[5]，该算法受著名的AFL-fuzzer的启发，对测试套件进行进化，以迭代地提高其分支覆盖率。

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.

整个工作流程如算法1所示。Variable suite是要生成的测试套件。它被初始化为空。每当一个测试用例覆盖一个新分支时，它就会被添加到套件中。可变种子是一组种子测试用例，在此基础上生成新的测试用例。首先，我们使用函数initPopulation（）生成一个初始测试套件。从第3行到第6行的循环然后迭代地演化测试套件。特别的，我们在第4行的套件中添加了种子中覆盖新分支（即套件中的测试用例未覆盖的任何分支）的测试用例。在第5行，我们通过函数fitToSurvive（）过滤种子中的测试用例，以便关注那些更可能导致测试用例稍后覆盖新分支的种子。在第6行，函数crossoverMuatation（）基于seeds中的测试用例生成更多的测试用例。循环将继续，直到触发预设的超时。虽然算法1与AFL中的算法相似，但不同之处在于每个函数的细节。下面，我们将详细介绍每个函数。

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 σas 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 = {0xf0}, SB = {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.

生成初始种群函数initPopulation（）生成包含多个测试用例的初始总体。如上所述，为了生成一个测试用例，我们需要生成一个初始配置σ以及一系列带有具体参数的（公共）函数调用。默认情况下，初始配置如下（十六进制）：b=0，ts=0，SA={0xf0}，SB={0xff00…}，v使用表示持久状态的每个变量声明的初始值进行设置。sFuzz还允许用户自定义初始配置，即允许用户提供一组初始测试用例。

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%).

接下来，我们生成多个交易序列，每个交易序列都是带有具体参数的函数调用。对于具有n个函数的合约，我们生成n个序列。在每个序列中，在调用构造函数之后调用一次不同的函数。这确保每个函数至少测试一次（即，函数覆盖率为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 uint256, 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.

对于每个函数调用，我们根据其类型为每个参数生成一个随机值。注意，如果参数类型具有固定长度，例如uint256类型，则这是简单的。如果类型没有固定长度（例如，数组或字符串），我们首先随机生成一个数字（范围从0到bound，其中bound是最大长度上的一个绑定，默认值为255），表示参数中的元素数（例如，字符数），然后生成相应数量的元素值。

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 ⌈logbound⌉ (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.

每个测试用例都以位向量的形式编码。在遗传算法的术语中，这样的位向量可以自然地看作为基因组。位向量的大小等于编码配置的位数加上编码函数调用的位数。请注意，对于每个测试用例，我们都会保留一个函数调用列表（在合约中始终包含构造函数），然后对每个参数值进行编码。如果参数值的长度可变，则使用⌈logbound⌉（其中bound是最大长度上的一个绑定，默认值为255）对参数值的长度进行编码。例如，给定图1所示的合约，测试用例的编码（部分）如图3所示，其中编码的每个部分都在图中标记。它包含192个字节，其中前96个字节是初始配置，后96个字节是两个函数调用和相应输入参数的序列。由于有三个字符串参数，前3个字节（包括0x05、0x05和0x05）分别编码\_response、\_rquestion和\_answer的长度。当动态变量超过3个时，将使用剩余的0x05值。

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.

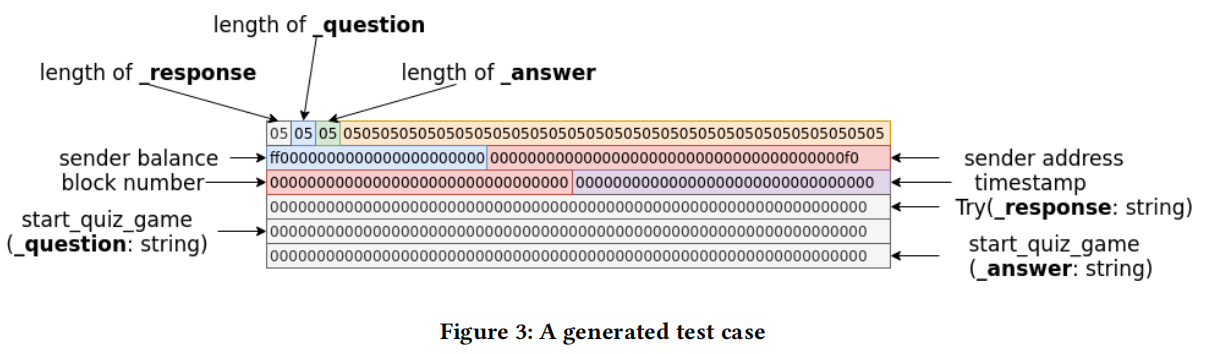
在执行测试用例之前，根据我们内部定义的协议将位向量解码为测试用例。注意，位向量中的位可以以多种方式彼此相关。例如，表示可变长度值长度的位必须等于该值的“长度”。

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.

在算法1的第4行执行种子后，调用函数fitToSurvive（）来根据适应度函数评估种子的适应度。注意，适应度函数起着极其重要的作用。

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).

在sFuzz中，我们结合了两种互补的策略。一种是采用AFL，其工作原理如下。在执行seed时，sFuzz监视执行并记录每个测试用例覆盖的分支。如果测试用例覆盖了合约中的一个新分支，例如套件中的任何测试用例都没有覆盖的分支，则该测试用例被认为是“适合生存的”。这种策略已经被证明在许多情况下是有效的[7]，而且我们的实验结果确实表明它在覆盖大多数分支时是有效的（见第5节）。



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 (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.

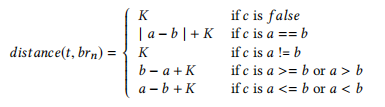
虽然AFL策略允许我们快速覆盖大部分分支，但它通常会在随后覆盖其余分支（通常是那些条件严格的分支）方面进展非常缓慢。原因是随机生成的测试用例很可能无法满足严格的条件。在这种情况下，上面的适应度函数对于如何生成新的测试用例提供很少的反馈和指导。例如，在图1的第10行满足第二个条件的概率很低（如果我们假设每个值生成的可能性相等）。然而，直观地看，很明显，msg.value=200的测试输入比msg.value=10000000的测试输入更接近于满足条件。因此，sFuzz集成了一个自适应策略，该策略基于测试用例距离覆盖任何刚刚丢失的分支多远的定量度量来选择种子。

Let br 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) where t is a test case to return a quantitative measure on how far the branch br is from being covered by t.

设br是S中一个刚刚丢失的分支，即S和n中分支节点n的一条未覆盖的出边被覆盖。其思想是定义一个函数距离（t，br），其中t是一个测试用例，用于返回一个关于分支br距离t覆盖多远的定量度量。

Assume that br 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 is assumed to be a just-missed branch, c must not be true (otherwise br must be covered already). Function distance(t,br) is then defined as follows.

假设br标记了一个条件c。注意c可以是true，false，a==b，a！=b、a>=b、a>b、a<=b或a<b在字节码级别，其中a和b是变量或常量。在我们的环境中，由于br假设是一个刚刚错过的分支，c 不能为true（否则必须已覆盖）。函数距离（t，br）定义如下。

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

其中K是表示最小距离的常数。在sFuzz中设置为1。直观地说，（t，br）的距离是这样定义的：分支越靠近被覆盖，结果值就越小。

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 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). 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.

在上述情况下，函数fitToSurvive（seeds）选择种子，如算法2所示。从第2行到第4行的循环遍历每个测试用例，以选择那些覆盖新分支的测试用例。之后，对于智能合约中每个刚刚错过的分支，从第5行到第11行的循环根据距离（t，br）从最接近覆盖分支的种子中选择一个测试用例。注意，为每个刚刚错过的分支选择一个种子，这使得该算法成为一种轻量级的多目标优化方法。所有选定的种子，然后用于交叉和变异，在下一步中生成更多的测试用例。我们请读者参考第2节中的示例。

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.

上述策略的灵感来自基于搜索的软件测试（SBST）[16，24]，但它在几个方面与SBST不同。造成这种差异的主要原因是，使用基于AFL的模糊化方法需要我们高效地运行测试用例，而现有SBST的种子选择策略非常耗时。此外，由于EVM基于栈的实现，现有SBST策略的实现是不可行的。下面，我们将详细介绍这些差异。

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 nto node nto be the minimum number of edges along any path from nto n. Let brbe 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.

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) 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 → B → 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]

其中br\_m是m的一个出边界，沿m到n的最短路径。请注意，如果m不是（即，在case br\_n刚刚错过），则br\_m就是br\_n。函数appr\_dist（t，br）是测试用例t覆盖branch br\_n的距离的度量，即m到n加1的距离。例如，给定如图4所示的控制流图，如果t仅覆盖边a→B→E，则appr\_dist（t，C）=1，因为从B到C有一个分支，从a到C有两个分支通过D。同样，appr\_dist（t，F）=2。最后，函数范数（x）是一个归一化函数，它将距离（t，br）的结果归一化为0到1之间的值。其中一个函数是norm（x）=1 − 1.001^−|x | .

Applying the above strategy in fuzzing Solidity smart contracts is inefficient, if not infeasible, for multiple reasons. First, calculating appr\_dist(t,br) 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) 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) 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) 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) is always 1 for any just-missed branchbrsince node n must have been covered. Furthermore, because it is constant for any uncovered branch, we can simply skip it in dist(t,br) and so that dist(t,br) is reduced to distance(t,br), without even the need to normalize. This further reduces the overhead.

由于多种原因，将上述策略应用于模糊Solidity智能合约是低效的，甚至是不可行的。首先，计算appr\_dist（t，br）需要构造完整的CFG。然而，仅基于字节码构造CFG是非常重要的。在EVM中，分支是用操作码jumpi实现的，一个在运行时动态地表示目标程序计数器的值。了解目标的唯一方法是完全模拟堆栈，这是非常昂贵的。第二，即使我们有CFG，计算appr\_dist（t，br）仍然是昂贵的。给定一个包含K个未覆盖节点的CFG。为了维护每个未覆盖节点的“最佳”测试用例列表，我们必须计算所有K个未覆盖节点的appr\_ dist（t，br），即，通过建立一个从所有节点到这些K个节点的最短路径表。此外，每当覆盖新节点时，必须更新appr\_ dist（t，br）。考虑到效率是基于AFL的模糊化的关键，开销是不合理的。通过将注意力集中在刚刚错过的分支上，sFuzz避免了这两个问题。也就是说，对于任何刚刚丢失的分支，appr\_dist（t，br）总是1，因为必须覆盖节点n。此外，因为它对于任何未覆盖的分支都是常量，所以我们可以简单地在dist（t，br）中跳过它，这样dist（t，br）就减少到distance（t，br），甚至不需要标准化。这进一步减少了开销。

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,b, ...,b}, a set of seeds {t,t, ...,t}, we say tis more Pareto-optimal than tif ∀k ∈ 0..m, distance(t,b) < distance(t,b). Otherwise, we say that tand tare 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.

sFuzz策略与现有SBST策略的另一个关键区别是多目标搜索策略。现有SBST中的多目标搜索策略以每个未覆盖的分支为目标，选择Pareto最优种子进行下一代进化。给定一组未覆盖的分支{b，b，…，b}，一组种子{t，t，…，t}，我们说它比tif∀k∈0..m，距离（t，b）<距离（t，b）更帕累托最优。否则，我们说tand-tare-Pareto等价。所有的帕累托等价种子形成一个帕累托前沿，种子可以分成几个帕累托前沿。现有的SBST选择帕累托最优的种子进行进化。这种策略的一个已知问题是，同一帕累托前沿中的种子数随着目标数（即未覆盖的分支）的增加而激增。例如，在Pareto最优前沿可能有数百个种子，只有3-5个目标，这使得很难选择最有希望的种子，并且增加了运行时开销。相反，sFuzz为每个刚刚错过的分支保留一个最佳种子（算法2中的第6-11行），因此种子的数量仍然很小（即，相当于刚刚错过的分支的数量）。我们的实验结果表明，这样的策略平衡了有效性，在确定好的种子和效率很好。

## 3.3 Crossover and Mutation

## 3.3 交叉与变异

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.

函数crossoverMutation（）通过交叉和变异，基于种子中的测试用例生成新的测试用例。sFuzz采用了AFL的所有交叉策略，并引入了针对智能合约的新闻策略。此外，由于测试用例参数之间的相关性，sFuzz还确保生成的测试用例是有效的。例如，sFuzz（1）随机选择两个来自种子的测试用例；（2）在选定的位置将两个测试用例分成两个片段；（3）交换第二个片段以形成两个新的测试用例。注意，由于代表测试用例的位之间的相关性，不能保证生成的测试用例是有效的，因此sFuzz总是检查有效性并丢弃那些无效的测试用例。

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.

突变是生成新测试用例的另一种方法。给定一个以位向量形式编码的种子，sFuzz支持一组变异算子来生成新的测试用例。所有变异算子如表1所示。

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.

回想一下，测试用例的形式是初始配置和一系列带有具体参数的函数调用。前三个变异操作符的目的是通过修剪一个函数调用、添加一个函数调用或交换两个函数调用来改变函数调用的顺序。当函数调用被删减（或添加或交换）时，相应的具体参数将被相应删减（或添加或交换）。

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 0xff0000000000000000000000.

对于测试用例中那些不代表被调用函数的值，sFuzz将它们分为两组。第一组包含具有固定长度的值（例如，uint256类型的参数）。sFuzz系统地应用表1中剩余的变异算子来生成新的值，这是受AFL中变异算子的启发。请注意，由于有特殊的格式要求，帐户地址（和余额）的处理略有不同（请参阅表中的最后一行）。每个地址有32个字节，其中最后20个字节包含地址值，前12个字节包含地址的余额。例如，值0xff00…00…00f0表示余额为0xff0000000000000000的地址0xf0。

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.

第二组包含长度可变的值（例如，数组类型的参数）。对于这些值，它们的长度也被编码为测试用例的一部分。因此，我们首先对表示长度的值进行变异，使得结果是0到255之间的随机值，其中255是上界。如果新长度小于当前长度，则通过修剪附加位来相应地缩短相应的值。如果长度大于当前长度，则相应地填充与随机类型兼容的值。

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 Walking-Byte 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].

请注意，我们丢弃了通过交叉或变异生成的相同测试用例。此外，虽然我们没有对从测试用例生成的突变数量设置限制，但是我们应用了从AFL中采用的多个启发式方法来减少突变的数量。例如，如果对32字节的块应用WalkingByte变异不会导致任何覆盖新分支的测试用例，那么在下一阶段sFuzz将不会对该块进行变异。我们请读者参考AFL，以了解这些启发式方法的详细信息[7]。

# 4 IMPLEMENTATION

# 4具体实现

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.

sFuzz在C++中实现，大概共有4347行代码。它是公开的[(https://sfuzz.github.io](https://sfuzz.github.io)网址). 它有3个主要组件：runner、libfuzzer和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.

组件运行器管理测试用例的执行。sFuzz将智能合约的字节码与合约的ABI（即应用程序二进制接口，可使用现有工具自动生成）一起作为输入。然后，运行程序生成一个bash脚本文件，其中包含用于分析ABI的命令列表，并为其他两个组件设置选项。

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.

运行程序建立了一个测试网络，在此基础上部署智能合约并执行事务。为了为带有地址类型参数的函数生成测试用例，sFuzz在测试网络中部署了一个带有随机余额的外部拥有的帐户池。池大小小于或等于地址类型参数的数目，因为可以将同一地址设置为多个地址类型参数。然后从这个池中随机选择地址类型参数的值。此外，sFuzz还部署了两个特殊的智能合约作为攻击者，即普通攻击者和重入攻击者。每个攻击者依次被设置为被测合约的所有者。普通攻击者在其他合约调用其回退功能时抛出异常。可重入攻击者回调调用其回退函数的函数。如果攻击者未能回电，它将充当普通攻击者。请注意，可重入攻击者的加载只是为了检测可重入漏洞。否则，加载普通攻击者以避免可重入攻击者的调用循环，从而显著降低sFuzz的速度。

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.

组件libfuzzer通过实现前面介绍的模糊化策略，解决了测试生成问题，即如何有选择地生成测试用例。它负责多项任务。

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.

首先，动态构造给定智能合约的CFG。理想情况下，我们希望在模糊化之前静态地构造CFG。然而，仅基于字节码构造CFG是非常重要的。在EVM中，分支是用操作码jumpi实现的，一个值在运行时动态地表示目标程序计数器。了解目标的唯一方法是完全模拟堆栈，这是非常昂贵的。因此，sFuzz在模糊化的同时动态构造CFG。也就是说，每当执行操作码jumpi时，记录两个目的地。如果这两个目的地还不是CFG的一部分，那么将相应地创建两个新节点来表示CFG中的两个目的地。

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.

其次，组件libfuzzer实现了第3节中讨论的模糊化算法。一个优化是我们识别视图函数（即那些不改变任何变量的函数）并将它们从测试用例生成中排除。理由是这些视图函数不会改变状态，并且拥有它们不会额外暴露sFuzz目标的那些漏洞（见下文）。请注意，视图函数由view、pure或constant关键字标记，sFuzz读取ABI文件来识别它们。

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), 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.

组件库liboracles 解决了预言机的问题，也就是说，它监视测试用例的执行，并根据sFuzz中使用的oracle可扩展库检查是否存在漏洞。sFuzz通过EVM支持的挂钩机制监视测试用例的执行。每当EVM执行一个操作码时，它都会创建一个包含只读执行信息的事件，例如堆栈、内存、程序计数器和当前执行的操作码的值。sFuzz监视这些事件以构建CFG和计算距离（t，br），并记录事件以进行漏洞检测。为了减少执行开销，漏洞检测是离线批量进行的（即每500个测试用例检测一次）。这种设计使得sFuzz能够轻松地支持不同版本的Solidity，也就是说，只需更换sFuzz中的EVM即可。

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.

sFuzz有一个可扩展的体系结构，允许它轻松地支持不同的预言机。目前，sFuzz支持8个预言，这些预言的灵感来自于之前的工作[18，22]。由于这些预言不是我们的主要贡献，我们请读者参阅[18，22]了解详情。

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).

这些预言是根据测试用例的日志来检查的。例如，为了检查测试用例是否暴露了Gasless Send漏洞，我们检查当gas等于2300时，测试用例是否使用大于0的数据执行调用指令。暴露合约中漏洞的测试用例保存在一个单独的测试套件中，并与它们暴露的漏洞一起报告给用户。请注意，根据我们对脆弱性的定义，sFuzz在设计上总是报告真正的积极因素，但在冻结乙醚的情况下除外。然而，在实践中，报告的漏洞可能是误报，因为它可能是用户想要的（即，我们对漏洞的定义过于严格）。在冻结乙醚的情况下，如果存在一些调用send（）或transfer（）的测试用例，但从未生成此类测试用例，则标识的“警告”可能是假阳性。从技术上讲，只有覆盖所有可行的操作码（通常是不可行的），才能解决检查是否存在漏洞的问题。

# 5 EXPERIMENTS AND EVALUATION

# 5实验与评价

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

在本节中，我们通过多个实验来评估sFuzz。实验旨在回答以下研究问题（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?

•RQ1: sFuzz的效率如何？

•RQ2: sFuzz在发现智能合约漏洞和获得高代码覆盖率方面是否有效？

•RQ3:自适应策略有用吗？

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.

我们的测试对象包括我们从EtherScan(一种以太坊浏览器)收集的4112个智能合约[4]。这些合约使用Solidity 4.2.24实现，这是Solidity最流行的版本。此外，这些合约的源代码是可用的，这使得评估更加准确。我们注意到sFuzz只能用字节码运行。为了进行基线比较，我们将sFuzz与[15]中报告的名为ContractFuzzer的fuzzer和[22]中报告的名为Oyente的符号执行工具进行了比较。在[21]中提到了智能合约的其他模糊器。然而，我们没有在网上或通过作者找到所报道的工具。我们进行了三次实验，并报告结果的平均值。下面报告的所有实验结果都是在带有Intel Core i7和16GB内存的Ubuntu18.04.1 LTS机器上获得的。我们使用第3.2节中介绍的默认初始配置。

## 5.1 Efficiency

## 5.1效率

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.

为了回答RQ1，我们系统地将sFuzz、ContractFuzzer和Oyente应用于所有4112个智能合约。为了节省时间，在这个实验中，每个合约运行2分钟。请注意，一般来说，自适应模糊策略需要时间来显示其有效性（我们将在后面演示），因此此设置为其他工具提供了优势。

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.

我们通过计算每秒生成和执行多少测试用例来衡量sFuzz的效率。自然地，一个更复杂合约的测试用例（例如，有许多循环迭代）需要更多的时间来执行。因此，我们展示了不同合约的效率是如何变化的。图5总结了结果，其中每个条表示10%（大约400）的模糊合约，y轴显示了每秒生成和执行的测试用例的数量。合约是根据模糊化的效率来排序的。从图中，我们观察到不同合约的效率有显著差异，即sFuzz平均每秒为前10%的合约生成和执行989个以上的测试用例，为后20%的合约生成和执行不到14个测试用例。平均而言，sFuzz每秒生成并执行208个以上的测试用例。

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].

我们进一步进行了一个实验来测量监视测试用例执行的开销（使用挂钩机制）和模糊化过程的总体开销（包括监视执行的总体开销、构造CFG、对测试用例进行变异和比较等等）。我们将sFuzz应用于一组60个随机选择的合约，并测量执行测试用例、监视执行和模糊化过程的其他步骤所花费的时间。结果表明，平均而言，监视消耗了约10%的执行时间，模糊化过程（包括监视）的开销约为14%. 与其他模糊器中报告的开销相比，这是非常有效的[32]。

## 5.2 Effectiveness

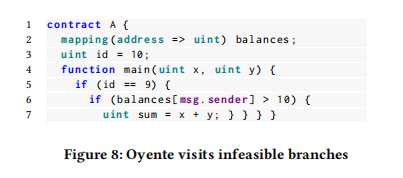
## 5.2 有效性

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.

为了回答RQ2问题，我们的目标是测量为每个智能合约生成的测试套件所实现的分支覆盖率，并计算已识别的漏洞数量。然而，精确地测量分支覆盖率是非常重要的，例如，由于分支不可行的问题。因此，我们测量生成的测试套件覆盖的不同分支的数量。图6总结了sFuzz和ContractFuzzer在不同分支数量方面的比较。y轴是sFuzz覆盖的分支数减去ContractFuzzer的分支数，x轴上的每个点表示智能合约。合约按其y轴值排序。类似地，图7显示了sFuzz和Oyente之间的比较。

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.

对于大多数智能合约（即4112个合约中的4077个），sFuzz覆盖的分支比ContractFuzzer更多。令我们惊讶的是，ContractFuzzer成功地为35份合约覆盖了更多的分支机构。



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.

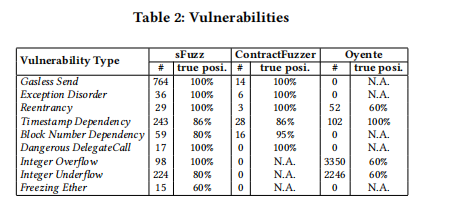
更细致的调查表明，ContractFuzzer 所覆盖的分支机构数量膨胀的原因如下。首先，由于sFuzz不执行视图函数（出于效率原因），因此这些函数中的所有分支都不计算在内。因为视图函数不修改智能合约的状态，所以它们被认为与漏洞无关。其次，ContractFuzzer有时会生成无效的测试用例，这些测试用例没有通过强制约束并覆盖其他分支。强制约束由编译器（即Solidity编译器）生成，并嵌入字节码中，以assert函数调用或数据类型的正确性逻辑。例如，ContractFuzzer调用非回调合约的回退函数，或将以太坊发送到未标记为payable关键字的函数。结果，强制约束失败，这导致分支发出覆盖的测试用例中的错误信号。

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.

就Oyente而言，在3402份合约中，Oyente覆盖的分支多于sFuzz。一项调查显示，Oyente分别分析每个函数，因此必须假设状态变量可以取任意值（不考虑其初始值或值更新方式的约束）。因此，Oyente可以轻松满足智能合约中的几乎所有条件。给定图8中的示例合约A，Oyente覆盖了99.1%的EVM代码，并发现了一个整数溢出漏洞。这意味着满足以下条件：id==9，余额[msg.sender]>10。但是，这是不可能的，因为无法更改id和balances[msg.sender]。通常，智能合约中的一个条件是比较局部/参数变量和状态变量，例如，balances[msh.sender]>值（sender是否有足够的以太币来推断）。在这种情况下，sFuzz必须调用函数，该函数在满足状态变量之前将某些值设置为状态变量，而Oyente则直接将任意值指定给状态变量。很明显，Oyente的方法是有缺陷的，会“覆盖”许多不可行的路径。

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.

下面，我们总结了sFuzz在每个类别中发现的易受攻击合约的数量。结果见表2。第一列显示了漏洞的类型。接下来的三列分别显示了sFuzz、ContractFuzzer和Oyente发现的易受攻击合约的数量。子列#根据每个漏洞类型显示具有漏洞的合约数量，第二子列是已识别漏洞的真实正面百分比。对于所有类别，sFuzz发现比contracterfuzzer更脆弱的合约。请注意，ContractFuzzer从源代码中删除冻结以太币，并且不检查整数上溢/下溢。sFuzz总共在1113份合约中发现了漏洞，是ContractFuzzer的24倍。

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.

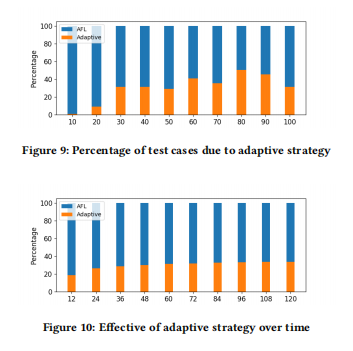
为了评估sFuzz的稳健性，我们手动检查已识别的易受攻击的合约，以检查它们是否为真阳性。但是，由于两个原因，我们无法手动检查所有已识别的漏洞。首先，存在大量漏洞。相反，我们随机抽取50个易受攻击的合约，每个类别中都包含源代码，并手动检查所识别的漏洞是否为真阳性。如果类别中包含源代码的易受攻击的合约少于50个，我们将检查所有这些合约。

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).

对于Gasless Send、异常无序和重入漏洞，所有50个抽样的易受攻击的合约都是真阳性。对于时间戳依赖，在50个抽样的易受攻击的合约中，有43个是真阳性。在剩下的7份合约中block.timestamp和/或现在用于与以太币发送部分无关的条件（即，无控制/数据相关性）。相反，它们的值保存在全局变量中，以记录特定事件的创建时间。sFuzz错误地宣称此类案件易受攻击。对于区块数依赖，在50个抽样的易受攻击合约中，有40个是真阳性。同样，10个误报的原因是块编号分配给全局变量，但它们与发送过程无关。对于危险的DelegateCall，所有17个抽样合约都确实易受攻击。整数溢出也是如此。对于整数下溢，50个确定的合约中有40个确实易受攻击。这10个误报的原因是，仅基于字节码识别变量的正确类型（例如，它是uint256还是uint128）是非常重要的，sFuzz保守地假设所有返回负值的算术运算都可能受到攻击。通过采用[29]中基于EVM字节码推断类型的方法，可以改进这一点。最后，对于冻结以太漏洞，15份合约中有9份是真阳性。这6个误报的原因是，尽管存在允许合约发送以太的程序路径，但该程序路径未被覆盖，并且sFuzz错误地假设没有这样的程序路径。如果sFuzz应用更长时间（覆盖更多分支），这种误报的百分比有望降低。

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.

表2中的最后一列显示了Oyente的结果。由于Oyente需要源代码，因此应该对结果持保留态度。例如，在源代码中知道变量的类型是很简单的，因此Oyente发现了更多整数上溢/下溢的问题。对于其余的漏洞，Oyente不支持其中的5个；识别出更多的用于重入的易受攻击合约，但具有更高的误报率；识别出更少的用于时间戳依赖的易受攻击合约。



## 5.3 Adaptiveness

## 5.3适应性

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.

为了回答RQ3，我们系统地分析了sFuzz为每个智能合约生成的测试套件。请注意，每个测试用例至少包含一个分支，而其他任何测试用例都不包含该分支。为了衡量在sFuzz中实现的两种模糊化策略是如何互补的，我们统计了由于AFL策略和自适应策略在生成的测试套件中生成的测试用例的数量。注意，当且仅当测试用例是基于由第11行在算法2中选择的种子生成时，测试用例被判断为是由于自适应策略引起的。

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.

结果如图9所示，其中y轴是策略生成的测试用例的百分比。每一条代表10%的合约。我们注意到这两种策略有不同的目标，因此它们是否有效很大程度上取决于智能合约中的分支条件。因此，我们根据sFuzz的速度对合约进行排序。因此，最右边的横条代表前10%的合约。我们观察到，正如预期的那样，AFL策略很容易覆盖大多数分支（因为执行大多数分支的条件并不严格）。对于大约80%的智能合约，自适应策略做出了显著的贡献，即平均贡献了31%的生成测试用例。考虑到sFuzz只适用于每个合约2分钟，结果令人鼓舞，因为我们假设如果sFuzz应用更长时间，自适应策略的效果会更明显。

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.

为了验证我们的假设，我们每12秒记录一次自适应策略生成的测试用例的百分比。结果如图10所示，其中x轴是模糊时间，每个条显示特定秒数后的百分比。我们可以观察到，随着模糊化时间的增加，自适应策略生成的测试用例的百分比增加。平均来说，这个百分比从12秒后的18%上升到2分钟后的33%。结果表明，自适应策略有助于提高测试集的覆盖率。

**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.

**有效性威胁**。我们的工作既有内部威胁，也有外部威胁。对于外部威胁，sFuzz的性能很可能会随着初始种群的选择而变化，正如其他研究人员所指出的[20]。对于内部威胁，表2中的真阳性百分比可能不准确，因为它们是由每种类型的漏洞的50个合约样本近似得出的。此外，合约作者的确切意图并不总是明确的，即使我们尽力阅读源代码。

# 6 RELATED WORK AND CONCLUSION

# 6相关工作及结论

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与现有的智能合约模糊器密切相关。ContractFuzzer[18]是一个fuzzer，可以检查7种不同类型的漏洞。然而，它的方法不使用任何反馈来改进测试套件。Echidna [3]是另一个fuzzer，据说它能够检查合约是否违反了一些用户定义的属性。然而，我们没有找到任何关于它的相关论文。sFuzz是对现有智能合约符号执行引擎的补充。在[22]中，Luu等人提出了一种在智能合约中发现潜在安全漏洞的引擎。然而，这个工具既不健全也不完整。在[21]中，Krupp和Rossow介绍了teEther，重点是金融交易和相关的漏洞。在[25]中，Nikolic等人提出了一个名为MAIAN的工具，可以发现3种类型的跟踪漏洞。在[29]中，Torres等人介绍了Osiris，一种结合符号执行和污点分析的工具，用于发现智能合约中的3种整数错误。与上述工作不同的是，sFuzz是一个模糊器，它可以与上述引擎结合形成一个混合模糊引擎。sFuzz与智能合约的正式验证相关。Zeus[19]是一个基于LLVM验证智能合约正确性和公平性的框架。Bhargavan等人提出了一个框架，通过将源代码和字节码转换为F\*（一种设计用于验证的语言）来正式验证智能合约[9]。在[17]中，作者尝试使用Isabelle/HOL[26]来验证合约合约。

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.

sFuzz与分析智能合约的工作有着广泛的联系。在[13]中，Delmolino等人证明了编写一个安全的智能合约并不是一项简单的任务。在[8]中，Atzei等人提供了智能合约中常见漏洞的分类法，其中包含真实世界的攻击。在[14]中，作者进行了调用图分析，结果表明只有40%的智能合约是不真实的，因为它们的控制流是不变的。在[10]中，Chen等人提出了7种gas成本规划模式，并表明大多数合约都受到这些gas成本模式的影响。

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.

最后，在这项工作中，我们提出了sFuzz，一个适用于EVM智能合约的自适应模糊引擎。实验结果表明，sFuzz比现有的模糊器更可靠、更快、更有效。sFuzz目前正处于快速发展阶段，已经引起了多家公司和研究机构的兴趣。

# ACKNOWLEDGMENTS

# 致谢

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

这项研究得到了新加坡教育部（MOE）Acemedic研究基金（AcRF）一级资助。

# REFERENCES

# 参考文献

[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 2005 ACM SIGPLAN conference on Programming language design and implementation. 213–223.

[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 {} {} . 1317–1333.USENIXSecurity Symposium (USENIXSecurity 18)

[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).