IEEE INFOCOM 2024 - IEEE Conference on Computer Communications | 979-8-3503-8350-8/24/$31.00 ©2024 IEEE | DOI: 10.1109/INFOCOM52122.2024.10621414

`ConFuzz`: Towards Large Scale Fuzz Testing of Smart Contracts in Ethereum

Taiyu Wong[1], Chao Zhang[1]*, Yuandong Ni[1], Mingsen Luo[2], HeYing Chen[3], Yufei Yu[1], Weilin Li[3], Xiapu Luo[4], Haoyu Wang[5]

[1]Tsinghua University
[2]University of Electronic Science and Technology of China [3]University of Science and Technology of China
[4]The Hong Kong Polytechnic University [5]Huazhong University of Science and Technology

***Abstract*—Fuzzing is effective at finding vulnerabilities in traditional applications and has been adapted to smart contracts. However, existing fuzzing solutions for smart contracts are not *smart* enough and can hardly be applied to large-scale testing since they heavily rely on source code or ABI. In this paper, we propose a fuzzing solution `ConFuzz` applicable to large-scale testing, especially for bytecode-only contracts. `ConFuzz` adopts Adaptive Interface Recovery (AIR) and Function Information Collection (FIC) algorithm to automatically recover the function interfaces and information, supporting fuzzing smart contracts without source code or ABI. Furthermore, `ConFuzz` employs a Dependence-based Transaction Sequence Generation (DTSG) algorithm to infer dependencies of transactions and generate high-quality sequences to trigger the vulnerabilities. Lastly, `ConFuzz` utilizes taint analysis and function information to help detect harmful vulnerabilities and reduce false positives. The experiment shows that `ConFuzz` can accurately recover over 99.7% of function interfaces and reports more vulnerabilities than state-of-the-art solutions with 98.89% precision and 93.69% accuracy. On all 1.4M unique contracts from Ethereum, `ConFuzz` found over 11.92% vulnerable contracts. To the best of our knowledge, `ConFuzz` is the *first* efficient and scalable solution to test *all* smart contracts deployed in Ethereum.**

***Index Terms*—Ethereum, Smart contract, Fuzzing**

I. Introduction

Supporting smart contracts is one of the most distinguishing features of blockchain 2.0. Smart contracts are programs running on blockchain systems (e.g., Ethereum [1], EOS [2], etc). Like traditional programs, smart contracts are inevitable to have vulnerabilities, which may lead to severe loss. For example, a reentrancy vulnerability was exploited by attackers to steal around $50 million worth of Ether in 2016 [3]. Therefore, finding vulnerabilities in smart contracts is critical to building a healthy blockchain ecosystem.

Various approaches have been proposed to detect vulnerabilities in smart contracts, which can be divided into two categories. First, approaches based on static analysis either look for patterns of vulnerable code in smart contracts [4], [5], [6] or leverage formal verification and symbolic execution [7], [8], [9] to determine whether smart contracts satisfy predefined vulnerability constraints. However, the former usually leads to many false positives due to the lack of reachability analysis, while the latter usually has many false negatives because of issues of path explosion and unsolvable path constraints. The second approach is dynamic analysis. They either detect known attacks [10], [11] at runtime or generate random inputs to trigger vulnerabilities in smart contracts (a.k.a. fuzzing) [12], [13], [14], [15], [16], [17]. However, the former can only detect a limited number of known attacks and has difficulty exploring the input space. In practice, as proved by traditional application testing, the latter approach – fuzzing is more effective and efficient at discovering vulnerabilities.

Although fuzzing can effectively discover vulnerabilities without false positives, existing approaches are not smart enough to test large-scale smart contracts (e.g. all contracts from Ethereum) and cannot have a reasonable performance in the testing process. Specifically, they suffer from the following three major limitations. First, existing approaches rely on contracts' ABI or source code, but only a small number of contracts reveal these. Second, triggering the vulnerabilities in smart contracts may involve multiple transactions (i.e., multiple invocations of multiple functions) in a certain order. Although some approaches have considered transaction sequences, there's plenty of room for improvement. Third, previous vulnerability detection oracles (or sanitizers) have defects that lead to false positives and false negatives.

Regarding the first problem, SigRec [18] utilizes the differences of the EVM instructions that can operate parameters and designs 31 rules to infer function parameters. But these rules do not apply to some situations and have false positives. ConFuzzius [16] combines symbolic taint analysis and a read-after-write transaction generation strategy, to partially address the second and third problems. However, it only works for open-source contracts in Ethereum, and its oracles have many false positives. SMARTIAN [17] also considered the transaction sequences with define-use order. It partially supports bytecode-only fuzzing but does not consider the impact of function parameter types, resulting in many meaningless inputs and mutations during the fuzzing process.

Our work. In this paper, we propose the first fuzzing approach, `ConFuzz`, able to large-scale testing, which utilizes static analysis and taint analysis to overcome the aforementioned limitations and address three challenges: **C-1:** gener-

* Corresponding author: chaoz@tsinghua.edu.cn

This work is supported in part by the National Key Research and Development Program of China (2021YFB2701000).

ating valid test cases for functions in smart contracts without ABI or source code, **C-2:** generating proper transaction sequences to meet expected dependencies and improve code coverage as well as the effectiveness of vulnerability discovery, and **C-3:** detecting harmful vulnerabilities only with lower false positives and avoiding harmless vulnerabilities.

To address **C-1**, we design a static analysis engine with taint analysis and a novel approach named Adaptive Interface Recovery (AIR, §IV-B) to infer function signatures and parameters from the smart contract's bytecode. It utilizes taint analysis and key instruction patterns to learn the usage of function parameters and infer their types.

To address **C-2**, we design a Dependence-based Transaction Sequence Generation (DTSG) algorithm. It first identifies dependencies among functions that access the same storage variables during contract execution and then generates proper transaction sequences satisfying the dependencies. It generates transaction sequences with a new *multi-write-then-read* order, which could better trigger branches and vulnerabilities with fewer fuzzing iterations and is suitable for large-scale fuzzing.

To tackle **C-3**, we propose a more precise vulnerability oracle to capture harmful vulnerabilities. We utilize taint analysis to detect not only whether a vulnerability is triggered but also whether it brings real harm to the state of the blockchain (i.e., values of storage variables and history of ETH transfers). Besides, we use the static analysis engine to discover the function attributes with Function Information Collection (FIC) to aid oracles in identifying vulnerabilities, further significantly reducing the false positives and false negatives. Our oracle can detect more types of vulnerabilities, including the ERC20 access control vulnerability.

Evaluation and Results. We implemented a prototype of `ConFuzz`, and conducted thorough experiments on both benchmarks and *all* Ethereum smart contracts. Experimental results showed that AIR can recover interfaces for more than 2.6 million functions compiled with different compiler versions, optimizations, and options, with an accuracy of 99.723% and a speed of 0.1 seconds per function, outperforming baselines. With the knowledge from the 4-bytes dataset [19], AIR can get an accuracy of 99.902% and speed of 0.01s per function. Further, ConFuzz armed with AIR can get similar code coverage (decrease less than 1%) as fuzzers depending on ABI or source code, indicating AIR is effective and useful.

To evaluate the performance of vulnerability discovery, we built the *largest* benchmark of vulnerable contracts, consisting of all contracts collected from previous works including Contractfuzzer [12], ConFuzzius [16], SMARTIAN [17] and others, and samples from our manual collection. This benchmark contains 1,080 contracts with 11 different types of vulnerabilities, including ERC20 access control vulnerability that the previous fuzzers do not support. Results show that, `ConFuzz` finds 142 and 156 more vulnerabilities compared to Confuzzius and Smartian, and has with higher precision (98.89%), higher recall (91.83%), and higher accuracy (93.69%).

Lastly, we apply `ConFuzz` to analyze all contracts deployed in Ethereum and conduct the *first* larget scale testing. Results show that, there are around 1.4M unique contracts deployed in Ethereum until May 17th, 2023, and `ConFuzz` finds that 11.92% of them (i.e., 165k contracts) are vulnerable and have 243k vulnerabilities. We manually audit hundreds of them and find that the false positive rate is consistent with the benchmark experiment, and find a newly revealed 1-day vulnerability in an active ERC20 project during the audit.

Contributions. We make the following contributions:

- We design `ConFuzz`, a new fuzzer for smart contracts with only bytecode, which is the *first* fuzzer applicable to large-scale testing of all smart contracts deployed in Ethereum.
- We propose a static analysis engine with a novel AIR and FIC algorithm to infer the interface of smart contracts to generate high-quality transactions.
- We propose a DTSG algorithm for constructing high-quality test cases consisting of transaction sequences.
- We design vulnerability oracles that can detect many types of vulnerabilities and trace their impact on the blockchain state to uncover harmful vulnerabilities. With the help of FIC, oracles can achieve lower false positives.
- We collected the largest dataset of vulnerable contracts with labels, and conducted the first large scale testing on all contracts deployed in Ethereum. We will open source our dataset and our prototype tool after publication.

II. Background

A. Ethereum smart contracts

Ethereum smart contracts are usually developed in high-level languages (e.g., Solidity [20], Vyper [21]) and then compiled into bytecode and deployed to Ethereum. Millions of smart contracts have been deployed on the Ethereum network.

To invoke a public or external function in a contract, the caller needs to know the interface, i.e., a 4-bytes function signature and related parameters [22]. Without the source code, it is difficult to get a contract's interface, which makes dynamic testing solutions unable to generate high-quality test cases.

Further, each contract account has its persistent storage for storing contract state variables, which will affect the execution of subsequent function invocations. For example, as shown in Listing 1, to trigger the vulnerable code in `close`, the attacker needs to send two transactions: (1) invocation of `setOwner`, which changes the value of state variable `owner` to the attacker address, and then (2) invocation of `close`. Without considering the state variables and the sequence of transactions, such vulnerabilities can hardly be triggered.

B. Fuzzing

Fuzzing is nowadays one of the most effective ways to detect vulnerabilities. A typical fuzzer generates a large number of random inputs, feeds them to the target program under test, monitors the program's execution state, and reports vulnerabilities when any security violation occurs [23]. This method has been proved efficient at finding vulnerabilities in software, as shown by many works, e.g., AFL [24], libFuzzer [25],

```
 1 contract Crowdsale {
 2   uint256 goal = 100000 * (10**18);
 3   uint256 phase = 0; // 0: Active, 1: Success, 2: Refund
 4   uint256 raised;
 5   uint256 end;
 6   address owner;
 7   mapping(address => uint256) investments;
 8 
 9   constructor() public {
10     end = now + 60 days;
11     owner = msg.sender;
12   }
13   function invest(uint256 donations) public payable {
14     require(phase == 0 && raised < goal);
15     investments[msg.sender] += donations;
16     raised += donations;
17   }
18   function setPhase(uint256 newPhase) public {
19     require( (newPhase == 1 && raised >= goal) ||
20              (newPhase == 2 && raised < goal && now > end)
21     );
22     phase = newPhase;
23   }
24   function withdraw() public {
25     require(phase == 1);
26     owner.transfer(raised);
27   }
28   function refund() public {
29     require(phase == 1);
30     msg.sender.transfer(investments[msg.sender]);
31     investments[msg.sender] = 0;
32   }
33   modifier onlyOwner()                          {require(msg.sender==owner); _;}
34   function setOwner(address newOwner) public {owner = newOwner; }
35   function close() public onlyOwner            {selfdestruct(owner); }
36 }
```

Listing 1: Motivation example.

Syzkaller [26] and many others [27], [28], [29], [30]. Despite its huge success in vulnerability discovery, designing an efficient fuzzer is non-trivial due to some challenges.

One is how to reach deeper states of the program. Randomly generated inputs [23] can only trigger a small part of the code, some of which cannot even pass input format checking, and only shallow program paths or states could be explored.

Another challenge is how to design oracles to capture vulnerabilities. Most fuzzers detect vulnerabilities by monitoring crashes which is effective for memory corruption vulnerabilities. However, vulnerabilities in smart contracts running on the EVM (Ethereum Virtual Machine) seldom cause crashes. Hence, new oracles are required for smart contract fuzzing.

III. Problem Definition

A. Motivation example

ConFuzz aims to detect vulnerabilities for bytecode-only smart contracts by smart fuzzing. Listing 1 is an example to illustrate the motivation and challenges. There are two vulnerabilities in this example, but triggering them requires the test case satisfying certain constraints.

The first vulnerability is an integer overflow at line 16 and 17. The adversary could call the invest multiple times or call it with a large argument donation to trigger the vulnerability, which will make the storage variable raised overflow and smaller than the goal. As a result, the phase of this contract cannot be set to 1 via invocation of setPhase, which further stops the users from withdrawing funds by invoking withdraw or refund. To trigger this vulnerability, the transaction should invoke the proper function (i.e., invest) with proper arguments (i.e., donations).

The second vulnerability lies in the function setOwner, which allows anyone to set the owner without proper checks. As a result, the adversary could further invoke close and cause an unprotected self-destruction. To trigger this vulnerability, the attacker should invoke these two functions in order.

Therefore, to discover these vulnerabilities, fuzzers in general need to take the following steps:

1) Get the list of functions that can be invoked and their interfaces, i.e., the number and types of their parameters;
2) Generate valid transactions to call those functions with proper parameters, and invoke them in a specific order to trigger the vulnerability;
3) Monitor the contract's execution to catch vulnerabilities with a pre-defined oracle (i.e., vulnerability sanitizer).
4) Collect function knowledge (e.g., onlyOwner) to assist oracles further reduce false positives and false negatives.

To discover the aforementioned vulnerability, ConFuzz first recovers the function's interface and collects the information through static analysis. Then it invokes the functions with appropriate parameters, sends the transactions following strategic sequences, and captures the harmful impact with oracles and function information specific to smart contracts by tackling the challenges introduced in §III-B - §III-E.

B. ABI Recovery

Before calling a function of a contract, the caller needs to prepare call data following the ABI. If mismatched parameter types are given, the smart contract will throw exception and terminate execution. For instance, to invoke function *invest(uint256_donation)* successfully, we need to prepare the properly data, which is encoded with a 4-bytes signature and a parameter by ABI specification [31]. Without the ABI, it is challenging to generate properly encoded data. Unfortunately, only a few deployed smart contracts have publicly available source codes or ABI information, and most deployed smart contracts have only bytecode and do not contain ABI information. Hence, ABI recovery is the first important step for efficient fuzzing.

C. Transaction Sequence Generation

The state variables (or global variables) stored in storage are shared by all functions in a smart contract, serving as dependencies across functions. Many vulnerabilities can only be triggered by a specific sequence of transactions (i.e., function invocations) which operate on the storage variables collaboratively. Without considering such dependency, it will be very difficult for fuzzers to generate proper transaction sequences and trigger the vulnerability. For example, in Listing 1, the unprotected self-destruct vulnerability cannot be triggered by just calling the *close* function. Instead, the attacker needs to first invoke the *setOwner* function with correct parameters and then invoke the *close* function to trigger it.

Note that, we could follow the access order of shared variables to determine transaction order. For example, the value of *raised* variable is read and checked in *setPhase*, but it can only be updated in *invest*. So, *invest* should be invoked

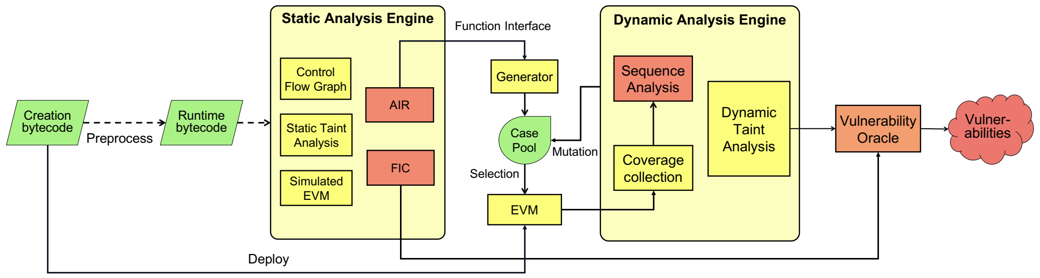

Fig. 1: Architecture of `ConFuzz`. It takes the contract byte code as input and reports harmful vulnerabilities as outputs.

first, even repeatedly, in order to increase the code coverage of *setPhase*. Therefore, a fuzzer should consider this *multi-write-then-read* order and generate transactions accordingly.

D. Oracles for Smart Contract

Besides feeding the target smart contract with proper input, fuzzers need an oracle to determine whether or not vulnerabilities have been triggered. Since smart contracts running in EVM rarely crash, we cannot use it as an oracle to determine vulnerability. Instead, we need to design smart-contract-specific oracles to capture the corresponding vulnerabilities. In Listing 1, we could first identify sensitive sinks (e.g., `selfdestruct` that might lead to the unprotected-selfdestruct vulnerability) via taint analysis, and verify whether it will cause real harm with an oracle that raises an alert if the *close* is called by the attacker's address (not owner).

E. Collecting function information

Developing a precise oracle to identify all potential vulnerabilities is an open challenge. For instance, if the *setOwner* function has an `onlyOwner` modifier, then the aforementioned oracle would have a false positive, since adversaries cannot change the ownership via `setOwner` and cannot trigger the vulnerability in `close`. Therefore, it is necessary to recognize certain function information, such as `onlyOwner`, in advance, to effectively reduce false positives and false negatives of the oracle.

IV. ConFuzz

A. Overview

Figure 1 shows the architecture of `ConFuzz`. To enable the direct deployment and execution of bytecode contracts on the local EVM, `ConFuzz` utilizes the contract's bytecode (deployed/creation bytecode) and then generates inputs to trigger potential vulnerabilities in it following the coverage-guided fuzzing strategy. `ConFuzz` only reports harmful vulnerabilities according to our contract-specific oracles and function information. The main process of `ConFuzz` is as follows.

Firstly, to facilitate analysis, preprocessing is required to transform the creation bytecode into runtime bytecode. Then, the static analysis engine runs the AIR and FIC algorithm and processes the contract bytecode to extract the function interface, information and builds the CFG (Control Flow Graph) with static taint analysis in our simulated EVM.

The generator takes in the recovered function interfaces to generate appropriate transactions with corresponding parameter types, which are then stored in the test case pool. Each input data in the transaction starts with a 4-byte function signature to call the corresponding contract function. After that, the EVM executes the bytecode according to the transactions fetched from the test case pool.

During the execution, the dynamic analysis engine performs data tracking for taint analysis, and the coverage collection module collects the coverage information to guide the test-case generation. The sequence analysis module executes the DTSG algorithm (§IV-C) to determine proper sequences for higher coverage. Meanwhile, the oracle continuously collects critical instructions and tainted results during the execution process. It combines this information with the corresponding function details discovered by FIC to comprehensively identify the types of vulnerabilities. This process stops after executing the preset maximum time of test cases, and `ConFuzz` reports the found vulnerabilities.

B. Static Analysis Engine

The static analysis engine analyzes the runtime bytecode of the contract, recovers the contract's function interfaces, and collects information on functions.

1) Simulated EVM and Taint analysis: We implement a group of stacks, memory, and storage to simulate the EVM, including all EVM instructions, to track the propagation and location of external data in the stack, memory, and storage, and help recover interfaces. To achieve this goal, we do not need precisely simulate the semantics of all instructions. Instead, we only implemented the placeholder of each instruction for stack, memory, and storage with fix value. For instructions that have input values pushed onto the stack, such as `CALLDATALOAD`, `CALLER`, `ADDRESS`, etc., we use different constant values to simulate them for distinction and analysis.

The static analysis engine takes the bytecode as input and disassembles the bytecode to our EVM and returns a complete CFG. It also utilizes a shadow stack (memory and storage) for taint analysis. The shadow stack performs similar to the EVM, but it only operates on tainted values from `CALLDATALOAD` and `CALLER`. By conducting taint analysis on their values, we could trace and understand the execution and propagation path of function parameters, which aids in the recognition of parameter types and collection of function information.

2) Adaptive Interface Recovery (AIR): Note that, different types of parameters will be used differently by instructions within smart contracts. So, the static analysis engine can recover the function signatures and parameter types via taint analysis. Our solution AIR first reads the function selector and finds the function signature with `CALLDATALOAD`, `PUSH4 signature`, `EQ`, `PUSH2`, `JUMPI` pattern from the contract's CFG. Then, it obtains the CFG of all public/external functions in the contract. Whenever the `CALLDATALOAD` instruction is executed, AIR taints its *loc* (top of stack) and pushes it into our stack. AIR analyzes the execution process of this external input and infers the types of function parameters following several pre-defined rules.

To distinguish between different parameter types, the AIR algorithm tracks each instruction and deals with them according to different rules as shown in figure 2, and rules R1-R6, R8, R16, R18 are the same as the rules in `Sigrec`[18]:

R7: R7 is used to infer a one-dimensional static array in a public function. We use a series of instructions to recover it. For a one-dimensional static array, the smart contract executes a `MUL` instruction, a `MLOAD` instruction, a `MSTORE` instruction, and a `MUL` instruction to compute the size of the array. Therefore, when we meet the second `MUL` instruction, we check the two values x and num at the top of the stack. If $x == 32$, the type is refined to a one-dimensional static array with num items, because every array item is 32 bytes.

R9: R9 means that for a `LT` instruction, it is used for bound check for the dimension, so we can record this value num when we meet a `LT` instruction. When a `LT` instruction is executed and a `CALLDATALOAD` instruction is inside a loop, this type will be refined to a one-dimensional static array with num items in a public function.

R10: R10 is similar to R7, which is used to infer a multi-dimensional static array in a public function. The smart contract executes the same instructions like R7 to compute the size of the array, except that one of them is high dimension and the other is the lowest dimension. Therefore, this type is refined to a multi-dimensional static array.

R11: R11 is also used to infer a multi-dimensional static array in a public function. R11 is similar to R8, except that the `CALLDATACOPY` instruction is inside a loop. When a `LT` instruction is executed and a `CALLDATACOPY` instruction is inside a loop, a `LT` instruction record a value num which is the lowest dimension and a `CALLDATACOPY` instruction also read three values at the top of the stack, where the third value len is the length of the high dimension array. Therefore, this type is refined to a array[$len/32$][num].

R12: R12 shows that a `LT` instruction means each dimension of a static array. Hence, when a `LT` instruction is executed and the `LT` instruction is inside a loop, we record the value each time and finally, this type is refined to a multi-dimensional static array whose dimensions are the values we have recorded.

R13: R13 is used to infer a static array in an external function. When a series of consecutive instructions which are `DUP1`, `PUSH`, `ADD`, `SWAP1` and `SWAP2` are executed, this type is refined to a static array in a external function.

R14: Because of the difference in the compilation options for smart contracts, we also use a series of consecutive and different instructions to recover a static array in an external function. A series of consecutive instructions are `PUSH`, `PUSH`, `MUL`, `DUP`, `ADD`.

R15: R15 is used to infer a string/bytes. If R6 is fulfilled and a `MLOAD` instruction and two `MSTORE` instructions are executed later, we believe that this type is string/bytes.

R17: R17 is used to infer a one-dimensional dynamic array. If R6 is fulfilled and the smart contract executes a series of instructions like R7, the type is refined to a one-dimensional dynamic array.

R19: R19 will be used to infer a multi-dimensional dynamic array. If R6 is fulfilled and a `CALLDATACOPY` instruction or a `LT` instruction is inside a loop, we will record the value which is the high dimension of a multi-dimensional dynamic array and it can be refined to a multi-dimensional dynamic array.

R20: We find that address is the same as a uint160. Therefore, we use the same way to recover the address. We do not distinguish between the two types.

3) Function Information Collection (FIC): The static analysis engine also employs taint analysis to analyze the CFG of functions and collect corresponding function information. For each function, we collect three types of information and provide them to the oracle for vulnerability detection. We choose static analysis because it is more suitable for efficiently analyzing single or multiple segments of bytecode to obtain crucial information about functions.

CALL Instruction The `CALL` instruction plays a crucial role in identifying vulnerabilities such as reentrancy and leaking. When a function uses the CALL instruction, the Function Information Collection (FIC) will analyze this `CALL` instruction to determine whether it is a function call or a native call/send. To achieve this, FIC simulates the execution of the function's CFG and examines whether there is a function signature after the 7th element on the top of the stack when encountering the `CALL` instruction. If a function signature exists, it indicates a function call. On the other hand, it implies a native call/send.

Payable Function A function with the "payable" attribute in a smart contract can accept Ether as part of its execution. It allows the caller to send Ether along with the function call. The "payable" attribute of a function can be helpful in determining locking ether and ERC20 access control vulnerabilities. A function that does not support "payable" will check in its second basic block whether `CALLVALUE` (the amount of Ether sent with the function call) is greater than 0. If it is greater than 0, the function will enter the branch of revert.

Sender Check When certain sensitive functions are invoked, they may check the identity information of the caller, such as using the "onlyOwner" modifier. Sender check information can assist the oracle in reducing false positives and false negatives while identifying vulnerabilities such as unsafe

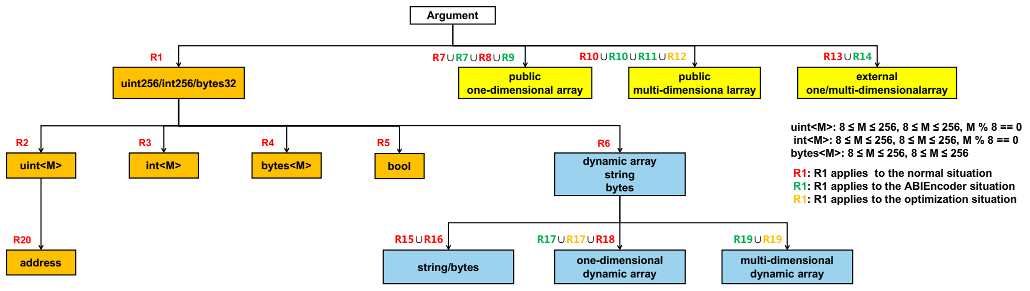
Argument
R1
uint256/int256/bytes32
R7∪R7∪R8∪R9
public one-dimensional array
R10∪R10∪R11∪R12
public multi-dimensiona larray
R13∪R14
external one/multi-dimensionalarray
R2
uint<M>
R3
int<M>
R4
bytes<M>
R5
bool
R6
dynamic array string bytes
R20
address
R15∪R16
string/bytes
R17∪R17∪R18
one-dimensional dynamic array
R19∪R19
multi-dimensional dynamic array
uint<M>: 8 ≤ M ≤ 256, 8 ≤ M ≤ 256, M % 8 == 0
int<M>: 8 ≤ M ≤ 256, 8 ≤ M ≤ 256, M % 8 == 0
bytes<M>: 8 ≤ M ≤ 256, 8 ≤ M ≤ 256
R1: R1 applies to the normal situation
R1: R1 applies to the ABIEncoder situation
R1: R1 applies to the optimization situation

Fig. 2: Hierarchy of rules applied by AIR.

delegatecall, leaking ether, self-destruct, etc. Specifically, during the process of simulating execution, when encountering the `CALLER` instruction, a taint is marked on the shadow stack to trace its propagation path. When the `EQ` instruction is encountered, the static analysis engine checks whether the shadow stack contains the taint propagated from `CALLER` (including taint propagated indirectly through storage). This analysis is employed to determine whether the function performs identity checks on the sender, contributing to more accurate vulnerability detection and minimizing false positives and false negatives.

Algorithm 1 The DTSG algorithm

```
while $instruction \neq STOP$ do
    $storage\_index \leftarrow stack[0]$
    if $instruction = SLOAD$ then
        $read\_maps[storage\_index].add(function\_signature)$
    else if $instruction = SSTORE$ then
        $write\_maps[storage\_index].add(function\_signature)$
    end if
end while
for $storage\_index \in read\_maps$ do
    $write\_functions \leftarrow select(write\_maps[storage\_index])$
    $read\_function \leftarrow select(read\_maps[storage\_index])$
    $seq \leftarrow (n * write\_functions, read\_function)$
    $sequences.add(seq)$
end for
```

C. Dependence-based Transaction Sequence Generation

We propose the Dependence-based Transaction Sequence Generation algorithm (DTSG) to generate proper transaction sequences in Dynamic Analysis Engine, which is listed in Algorithm 1. The DTSG infers function dependencies and generates sequences according to how the functions read and write the global variables. More precisely, if a conditional branch in one function is determined by a storage variable that can only be modified in other functions, only fuzzing a single function is useless. The DTSG algorithm generates possible sequences to modify this storage variable before reading, thus improving the coverage.

The DTSG first collects storage variable dependencies among functions and then generates transaction sequences based on the collected dependencies. Keys of maps are storage indexes, and values are function lists with the corresponding read or write operations to the storage.

The DTSG first start a While loop that tracks the active instruction until STOP to collect storage variable dependencies. `SLOAD` and `SSTORE` are instructions that read from and write to storage variables. They access the target storage variable index from the EVM stack. When either `SLOAD` or `SSTORE` is executed, DTSG records the current function into the *read_maps* or *write_maps*. When the *While* stops, all the dependence relationships among functions will be stored into *read_maps* and *write_maps*. Since not all storage slot indexes are fixed in the contract code, the dependence information can only be collected dynamically at runtime.

After collecting the dependence, the algorithm generates possible transaction sequences within the *For* loop. For each *storage_index* in *read_maps*, several corresponding function signatures in *write_maps* are randomly selected to the *write_functions* list. One of the *storage_index* related function signatures is selected to the *read_function* variable as well. At this point, each function in *write_functions* has a "write-read" operation dependence on the function in *read_function*. The algorithm then marks them as a possible transaction sequence pair *seq* in the order of writing first and reading later, and stores the pair to *sequences*. We repetitively add *write_functions* *N* times (default two) to the sequence, to address situations where certain branches or vulnerabilities require multiple writes before they can be triggered. Due to the large scale of the experiments, each contract is allocated a relatively short fuzzing time. This "multi-write-read" transaction sequence allows us to trigger branches and vulnerability with fewer fuzzing loops and solve the challenge in III-A, which is suitable for short-time and large-scale fuzzing.

D. Oracles for Vulnerability Detection

`ConFuzz` can detect 11 important types of vulnerabilities. We design oracles for each of them, and each oracle provides two kinds of guidance: vulnerability detection and harmfulness tracking. The former checks whether a vulnerability is triggered, and the latter checks whether the vulnerability is indeed harmful, i.e., can lead to ether transfer and/or storage modification. We explain the oracles as follows.

Integer Overflow (IO). The generalized integer overflow includes overflow and underflow. When arithmetic instructions like `ADD`, `MUL`, and `SUB` are executed, `ConFuzz` checks their operands and results. If overflow or underflow happens, the operation result will be marked as a taint source of *potential integer overflow*. The `ConFuzz` performs a taint check on operations like branch jump or storage writing and reports a harmful integer overflow vulnerability only if the *potential integer overflow* taint hits. The oracle keeps recording the operands involved in arithmetic instruction and checks whether they are in the conditional statement with tainted results (LT,GT.etc) to avoid false positives like overflow-checking functions (require/assert in Safemath's add()).

Locking Ether (LO). If a smart contract can only receive ether but cannot transfer it out, the ether is frozen in the smart contract. A contract with locking ether vulnerability should meet two conditions: (1) it can receive ether (2) it does not contain any ether transfer instructions. For the first condition, `ConFuzz` use FIC to check whether there are payable functions in contracts and transaction value larger than 0. Then, `ConFuzz` searches the `CREATE`, `CALL`, `DELEGATECALL` and `SUICIDE/SELFDESTRUCT` instructions in the static analysis engine. Without any of these instructions, a contract cannot transfer the ethers out. If both conditions are satisfied, `ConFuzz` reports a locking ether vulnerability.

Unsafe Delegatecall (UD). Contracts use the `DELEGATECALL` instruction to execute external code within their own environment. If the code executed by `DELEGATECALL` could be controlled by attackers, the smart contract has the unsafe delegatecall vulnerability. The oracle first checks if the execution trace contains a DELEGATECALL instruction called by the attacker's address. Then, the oracle will check if the attacker's address is present as a parameter in the transaction sequence by benign address and utilize the function information to determine whether the function has a sender check. If it checks the sender, the DELEGATECALL is benign.

Reentrantry (RE). Reentrancy is a serious vulnerability that led to an Ethereum hard fork. It happens when a contract invokes the functions of another contract, and that contract calls back the original contract. We detect reentrancy by check the gas in the `CALL` larger than 2300 or not and if the transaction value is larger than zero. Furthermore, we detect whether the `CALL` instruction is a call/send or a function call based on the function information from FIC. We only filter `SLOAD` and `SSTORE` for the same location that occurs before `CALL` instructions to avoid false positives.

Unprotected Selfdestruction (US). Similar to unsafe delegatecall, if the `SELFDESTRUCT/SUICIDE` instruction is executed by the attacker and the transaction sequence does not contain a sender check, it results in vulnerabilities.

Leaking Ether (LE). We detect the leaking vulnerability by checking if the execution trace contains a *CALL* instruction, whose object is an attacker address that has never sent ether to contract and is not the contract creator. We also examine the transaction sequence to determine if there is a sender check transaction that includes the attacker's address as a parameter with FIC to avoid false positives and false negatives.

ERC20 Access Control. Access Control vulnerabilities occur when certain critical or sensitive functions lack appropriate access control mechanisms. It allows unauthorized or malicious parties to call sensitive functions that should be restricted to specific users or authorized entities.

In ERC20 token, it usually happens in the transfer function. We believe that for a safe transfer, users should only be allowed to transfer tokens from their own addresses. Therefore, the oracle checks if the "from" address in the "Transfer" event matches the "msg.sender". We achieve this by executing the "LOG3" instruction from the trace to identify the "transfer" event and then compare "msg.sender" with the "from" address.

Furthermore, except for cases involving "transferFrom," the "from" address should not have taint propagated from external inputs. We exclude this by employing the FIC static analysis to identify the "approve" pattern that the taint from the external input address indirectly enters the stack through storage and makes subtraction with the second uint256 parameter. We also perform sender checks on transactions to filter safety functions like "mint" which can only be allowed for the contract owner and check "payable" to exclude buy() that swap ether to token.

For the Unhandled Exception (UE), Block Dependency (BD), Assertion Failure (AF), and Transaction Order Dependency (TO), we followed the oracle design of Confuzzius, and do not discuss here.

V. Evaluation

We conduct extensive experiments to answer the following research questions (RQ):

- **RQ1:** How does AIR perform in terms of recovering function interfaces and efficiency?
- **RQ2:** How is `ConFuzz`'s performance compared to other state-of-the-art smart contract fuzzing tools?
- **RQ3:** Can `ConFuzz` efficiently analyze all contracts deployed in the Ethereum network?

A. Performance of AIR (RQ1)

a) Interface Recovery: To evaluate our algorithm, we collected a total of 12.3k unique contracts with source code as ground truth, including 48k contract source codes from SmartBug [32] and an additional 100k contract source codes from Etherscan [33]. We also collect their bytecode and ABI from Etherscan and finally get 2.6M functions to evaluate. Unlike SigRec [18], we do not deduplicate the function signatures because even for the same function, the bytecode can be significantly different under various compilation options and compiler versions, and some function signatures even have collisions. Thus, we need to test AIR's performance when faced with these diverse scenarios. We have designed three tests to evaluate the efficiency and accuracy of AIR. The result is shown in Table I.

In the first test, we query the 4-bytes database [19] to get the ABI information. This database has ABI information for 94.919% of functions in our dataset. The query takes 0.003 seconds per function. **In the second test**, we only use the AIR algorithm to recover the ABI information. It recovers ABI information with 99.723% accuracy, i.e., more ABI than that collected by the 4-bytes database, and spends 0.093s per function. **In the last test**, we combine these two to recover ABI. In other words, if a function is not in the 4-bytes database, then we use AIR to recover it. In this way, the accuracy reaches 99.902%, and the speed is less than 0.01s per function. **We also evaluated SigRec as a baseline.** Since it is not open-source, we followed its paper to implement a

TABLE I: Performance of AIR.

	SigRec	4-byte database	AIR	AIR+4-byte
Accuracy	98.353	94.919	99.723	99.902
time/function	0.083	0.003	0.093	0.01

prototype on our own. The results showed that, its accuracy is 98.353% and speed is 0.083s per function.

b) Impact of AIR to fuzzing's code coverage: Note that, AIR cannot completely and accurately recover function interface information, and thus may affect the fuzzer's performance, e.g., the ability to explore more code during fuzzing. We conducted another measurements to assess the impact of AIR on the code coverage during fuzzing. We evaluated `ConFuzz` and Confuzzius (with our DTSG) on a ground truth dataset, with the same setup as Section V-B0a, and measured the code coverage achieved by each fuzzer. The results showed that, Confuzzius achieved a code coverage and branch coverage of 81.58% and 77.10%, while `ConFuzz` achieved 79.83% and 76.33%. The difference is approximately only 1%, indicating that AIR has a minimal impact on the coverage and can help the fuzzer to get a result close to the ideal.

Answer to RQ1: AIR can recover function interface information with high precision and negligible time overhead, which could help the fuzzer get comparable performance as fuzzers depending on source code or ABI.

B. Effectiveness of Vulnerability Detection (RQ2)

In this section, we conducted experiments to evaluate the vulnerability detection capability. As Confuzzius and Smartian have demonstrated superior performance compared to other fuzzers such as ContractFuzzer, sFuzz, and IFL in their papers, we focused on conducting comparative experiments with Confuzzius and Smartian.

a) Dataset: To create ground truth and avoid bias, we integrate datasets from previous studies to evaluate the performance of `ConFuzz`. We collected contracts from previous works including Contractfuzzer [12], ConFuzzius [16], SMARTIAN [17], JiuZhou [34], VeriSmart [35], TMP [36], SWC registry [37] and SmartBugs [38], and extended it with our manual collection. The dataset contains 1,080 contracts with 11 different types of vulnerabilities.

Note that, some of these benchmarks from previous papers have errors, i.e., some contracts are marked as vulnerable but instead are not. We therefore did a thorough manual audit to double-check whether each contract is vulnerable, and classified these contracts into different groups, based on the type of vulnerabilities it is marked with. The manual audit process is time-consuming and requires expert knowledge. We had two Ph.D students and two master students familiar with smart contract vulnerabilities to perform this audit. The audit takes about two months and the results are cross-checked. After the audit, we find a number of contracts wrongly marked as vulnerable and name them with negative samples. In total, we have 737 positive samples and 343 negative samples. The numbers of positive samples and negative samples of each vulnerability category are listed in Table II.

b) Evaluation Setup: We feed the source code and ABI of contracts to Confuzzius and Smartian, but feed `ConFuzz` with only contract bytecode.

c) Results: Table II lists the performance of `ConFuzz`, ConFuzzius, and Smartian in the full ground truth dataset. *Among all the vulnerability types,* `ConFuzz` *achieved the highest number of vulnerabilities detected with the lowest false positive and false negative rates.* ConFuzzius exhibited significant false negatives in detecting reentrancy vulnerabilities, which can be attributed to its overly strict oracle (i.e., reentrancy could occur without storage reads and writes). `ConFuzz` has much lower false negatives compared to the other two tools in terms of detecting unsafe delegatecall, unprotect selfdestruct and leaking ether, because it utilizes function information provided by FIC so that its oracle can accurately recognize owner transfer functions lacking a sender check and filter harmful vulnerabilities. In addition, `ConFuzz` achieved precision, recall, and accuracy rates of 98.82%, 91.83%, and 93.69%, respectively, surpassing the performance of the Confuzzius and Smartian.

Answer to RQ2: `ConFuzz` is effective at finding vulnerabilities on the verified ground-truth dataset, and greatly outperforms ConFuzzius and Smartian. The oracle and FIC of `ConFuzz` are effective and useful in vulnerability detection.

C. Efficiency and Large-Scale Study (RQ3)

We use `ConFuzz` to fuzz the bytecode of all deployed smart contracts in the Ethereum blockchain to evaluate its scalability.

a) Dataset: We collect nearly 1.4M unique contracts deployed in Ethereum (until May 17th, 2023) from Ethereum transaction database [39] of Google big query. We extracted input data from all contract creation transactions and deduplicated them to obtain the creation bytecode of these contracts.

b) Evaluation Setup: Due to the large number of contracts, each contract is fuzzed at most 10 seconds. Once the fuzzing stops, the vulnerability results will be reported.

c) Results: `ConFuzz` successfully fuzzed 75% of the contracts, while the remaining 25% could not be deployed due to local EVM environment configurations. `ConFuzz` discovered over 165k vulnerable contracts (11.92%), with nearly 243k vulnerabilities in total. Table III shows the distribution of vulnerability types. `ConFuzz` discovered over 600k AF vulnerabilities. This is mainly due to a large number of `INVALID` instructions are executed when errors occur. Since AF are not considered particularly severe vulnerabilities, we temporarily excluded them from the vulnerability statistics. For the rest data, BD is the most widespread, followed by IO and UE. Figure 3 shows the distribution of the number of vulnerabilities in the contracts. It shows about 2.63% of contracts have more than five vulnerabilities but most contracts have only 1 to 3 vulnerabilities.

d) Result Verification: To validate the experimental results, we conducted manual sampling checks on the vulnerability detection results. To ease the burden of manual verification, we sampled contracts with source code from the results and get 100 contracts with 132 vulnerabilities.

TABLE II: True positives, false negatives and false positives detected by Confuzz, ConFuzzius and Smartian per vulnerability type

	AF	BD	IO	LE	LO	RE	UE	US	UD	TO	ERC20	Prec/Rec/Acc
Smartian (tp/fn/fp)	8/6/0	76/48/24	104/25/21	27/13/3	34/29/2	24/95/0	67/3/8	45/13/24	3/14/27	2/1/0	NA	79.05/62.91/65.69
Confuzzius (tp/fn/fp)	12/2/0	118/6/2	96/33/16	25/15/0	47/16/0	13/106/0	63/7/3	47/11/0	9/8/0	2/1/0	NA	95.41/68.49/76.40
`ConFuzz` (tp/fn/fp)	**12/2/0**	**118/6/2**	**110/19/3**	**34/6/0**	**55/8/0**	**112/7/0**	**67/3/3**	**54/4/0**	**13/4/0**	**2/1/0**	**100/0/0**	**98.82/91.83/93.69**
Ground-truth positives/negatives	14/6	124/54	129/38	40/4	63/13	119/1	70/99	58/44	17/33	3/1	100/50	737/343

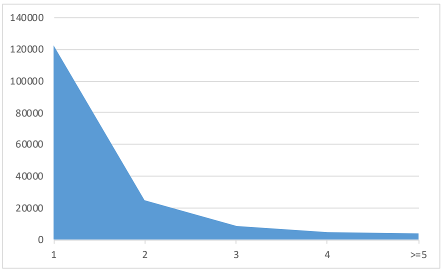

Fig. 3: Distribution of vulnerable contracts. About 11.92% smart contracts have at least one vulnerability. 122821 contracts have one vulnerability, 24754 contracts have two, and 18279 contracts have more than two vulnerabilities.

After manual verification, we find that 87% of them are true positives. We also found that the contracts with vulnerabilities were primarily playground contracts that were deployed early and had few transactions. On the other hand, contracts that have a higher number of transactions mostly belong to projects that are no longer active. Table IV presents some vulnerable contracts that we sampled and verified. It is worth noting that, among the sample vulnerable contracts, we have identified a 1-day access control vulnerability in an active ERC20 token project, which is recently found by [40]. It shows the results of our large-scale analysis is reliable, and the vulnerabilities found by `ConFuzz` are worth further inspection.

Answer to RQ3: `ConFuzz` can efficiently analyze deployed contracts in Ethereum and discover real vulnerabilities.

VI. Related Work

Fuzzing is the most popular solution to vulnerability discovery. There are many related works in this field. Due to the space limit, here we only discuss and compare fuzzing solutions that are specific to smart contracts.

ContractFuzzer [12] is one of the state-of-the-art fuzzer for smart contracts. It generates input according to the ABI information or the interface information extracted from the source code of smart contracts. Learning to Fuzz [13] combines symbolic execution, neural network, and fuzz to generate high-quality test cases. However, it is only effective for smart contracts that are similar to the training set. HARVEY [15] defines cost metrics with instrumentation to guide the input generation and focuses on the targeted fuzzing by using static lookahead analysis [15], which is efficient for fuzzing specifically targeted vulnerability. sFuzz [14] is an efficient feedback-guided fuzzer for smart contracts motivated by AFL, which improves the efficiency of input generation by measuring the distances between the input and the just-missed branch.

TABLE III: Distribution of vulnerability types

AF	BD	IO	LE	LO	RE	UE	US	UD	TO	ERC20
619072	108682	50092	2456	11427	11341	47583	6061	1233	1607	2088

TABLE IV: Examples of sampled vulnerable contracts

Type	Address	Function
IO	0x712c......ead4	getToken
IO	0x0c0a......ca07	DepositETH
IO	0x9c30......dc9c	mintToken
RE	0xdcb1......df02	execute
LE	0xcea8......5387	StartGame–StopGame
US	0x0f66......a9f4	kill
ERC20	0xb579......a4a3a	everyCoin
ERC20	0x3212......8923	distributeRevenue

Compared with these recent studies, `ConFuzz` recovers ABI information from the bytecode automatically and uses it to guide the generation of valid input, so that it can fuzz nearly all smart contracts. Moreover, `ConFuzz` generates efficient transaction sequences by analyzing the "write-read" operation dependency on state variables among functions and has more precise contract-specific oracles, which empower `ConFuzz` to effectively and efficiently discover vulnerabilities that cause real harm in a large scale of smart contracts.

ConFuzzius [16] is a data dependency-aware hybrid Fuzzer for contracts' source code, which combines symbolic execution and fuzzing. It uses evolutionary fuzzing to find bugs that exist deep in the whole execution process and takes constraint solving to generate the inputs that bypass complex judgment conditions. Moreover, ConFuzzius applies dynamic data dependency analysis to generate the different sequences of transactions to detect vulnerabilities more efficiently. `ConFuzz` also applies transaction sequence generation and taint analysis. The evaluation result shows that `ConFuzz` has a better performance and does not rely on source code.

SigRec is a tool for the automatic recovery of function signatures in smart contracts without the need for source code and function signature databases. It proposes the type-aware symbolic execution (TASE) which utilizes the differences of the EVM instructions that can operate parameters and designs 31 rules in total for specific EVM opcodes. However, its experiments were not thorough enough, and its performance was not as effective as AIR.

VII. Conclusion

We design and develop `ConFuzz`, a smart fuzzing tool for discovering vulnerabilities in smart contracts without source code or ABI. With the AIR algorithm to recover ABI information, the DTSG algorithm to generate proper transaction sequences, and the contract-specific oracles armed with FIC, `ConFuzz` could better detect harmful vulnerabilities. Experiments on a ground-truth dataset show that `ConFuzz` achieves very high precision and recall, greatly outperforming state-of-the-art tools. We also conducted the first extensive fuzz testing on all Ethereum smart contracts and found that around 11.92% of them have vulnerabilities.

References

[1] Ethereum. (2016) Ethereum white paper: a next generation smart contract & decentralized application platform. [Online]. Available: https://github.com/ethereum/wiki

[2] EOSIO. (2018) Eos.io technical white paper. [Online]. Available: https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md

[3] ConsenSys, "Ethereum smart contract best practices," https://consensys.github.io/smart-contract-best-practices/known_attacks/, 2018.

[4] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 254–269.

[5] ConsenSys. (2018) mythril: Security analysis tool for evm bytecode. [Online]. Available: https://github.com/ConsenSys/mythril

[6] C. F. Torres, J. Schütte *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 664–676.

[7] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing Safety of Smart Contracts." Internet Society, 2018.

[8] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy, SP*, 2020, pp. 18–20.

[9] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[10] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-Entrancy attacks," in *Proc. NDSS*, 2019.

[11] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang, "Soda: A generic online detection framework for smart contracts," in *Proc. NDSS*, 2020.

[12] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. Montpellier, France: ACM Press, 2018, pp. 259–269.

[13] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to Fuzz from Symbolic Execution with Application to Smart Contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security - CCS '19*. London, United Kingdom: ACM Press, 2019, pp. 531–548.

[14] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and M. Q. Trans, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. ICSE*, 2020.

[15] M. C. Valentin WÃ¼stholz, "Targeted greybox fuzzing with static lookahead analysis," in *Proc. ICSE*, 2020.

[16] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 2021, pp. 103–119. [Online]. Available: https://doi.org/10.1109/EuroSP51992.2021.00018

[17] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *Proceedings of the International Conference on Automated Software Engineering*, 2021.

[18] T. Chen, Z. Li, X. Luo, X. Wang, T. Wang, Z. He, K. Fang, Y. Zhang, H. Zhu, H. Li, Y. Cheng, and X. Zhang, "Sigrec: Automatic recovery of function signatures in smart contracts," *IEEE Trans. Software Eng.*, vol. 48, no. 8, pp. 3066–3086, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3078342

[19] (2023) Ethereum signature database. [Online]. Available: https://www.4byte.directory/

[20] Solidity, *Solidity, the Contract-Oriented Programming Language*, 2016. [Online]. Available: https://github.com/ethereum/solidity

[21] Vyper. (2017) Vyper documentation. [Online]. Available: https://vyper.readthedocs.io/en/latest/

[22] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger byzantium version," https://ethereum.github.io/yellowpaper/paper.pdf.

[23] V. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2020.

[24] M. Zalewski, "American fuzzy lop," *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[25] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *Cybersecurity Development (SecDev), IEEE*. IEEE, 2016, pp. 157–157.

[26] G. Inc., "syzkaller - kernel fuzzer." https://github.com/google/syzkaller.

[27] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[28] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[29] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "Greyone: Data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston, MA. https://www.usenix. org/conference/usenixsecurity20/presentation/gan*, 2020.

[30] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[31] Solidity. (2019) Contract abi specification. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.3/abi-spec.html

[32] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 1349–1352. [Online]. Available: https://doi.org/10.1145/3324884.3415298

[33] Etherscan. (2019) Ethereum (eth) blockchain explorer. [Online]. Available: https://etherscan.io/

[34] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in ethereum smart contracts," in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 2020, pp. 139–150. [Online]. Available: https://doi.org/10.1109/ICSME46990.2020.00023

[35] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 417–434.

[36] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network," in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, C. Bessiere, Ed. International Joint Conferences on Artificial Intelligence Organization, 7 2020, pp. 3283–3290, main track. [Online]. Available: https://doi.org/10.24963/ijcai.2020/454

[37] (2020) Swc registry. [Online]. Available: https://swcregistry.io

[38] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47, 587 ethereum smart contracts," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 530–541. [Online]. Available: https://doi.org/10.1145/3377811.3380364

[39] G. Cloud. (2019) Ethereum in bigquery: a public dataset for smart contract analytics. [Online]. Available: https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

[40] K. Qin, Z. Ye, Z. Wang, W. Li, L. Zhou, C. Zhang, D. Song, and A. Gervais, "Towards automated security analysis of smart contracts based on execution property graph," *arXiv preprint arXiv:2305.14046*, 2023.