CONFUZZIUS: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts

Christof Ferreira Torres [1], Antonio Ken Iannillo [2], Arthur Gervais [2], and Radu State [2]

[1]University of Luxembourg
[2]Affiliation not available

October 30, 2023

Abstract

Smart contracts are Turing-complete programs that are executed across a blockchain. Unlike traditional programs, once deployed, they cannot be modified. As smart contracts carry more value, they become more of an exciting target for attackers. Over the last years, they suffered from exploits costing millions of dollars due to simple programming mistakes. As a result, a variety of tools for detecting bugs have been proposed. Most of these tools rely on symbolic execution, which may yield false positives due to over-approximation. Recently, many fuzzers have been proposed to detect bugs in smart contracts. However, these tend to be more effective in finding shallow bugs and less effective in finding bugs that lie deep in the execution, therefore achieving low code coverage and many false negatives. An alternative that has proven to achieve good results in traditional programs is hybrid fuzzing, a combination of symbolic execution and fuzzing. In this work, we study hybrid fuzzing on smart contracts and present ConFuzzius, the first hybrid fuzzer for smart contracts. ConFuzzius uses evolutionary fuzzing to exercise shallow parts of a smart contract and constraint solving to generate inputs that satisfy complex conditions that prevent evolutionary fuzzing from exploring deeper parts. Moreover, ConFuzzius leverages dynamic data dependency analysis to efficiently generate sequences of transactions that are more likely to result in contract states in which bugs may be hidden. We evaluate the effectiveness of ConFuzzius by comparing it with state-of-the-art symbolic execution tools and fuzzers for smart contracts. Our evaluation on a curated dataset of 128 contracts and a dataset of 21K real-world contracts shows that our hybrid approach detects more bugs than state-of-the-art tools (up to 23%) and that it outperforms existing tools in terms of code coverage (up to 69%). We also demonstrate that data dependency analysis can boost bug detection up to 18%.

CONFUZZIUS: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts

Christof Ferreira Torres*, Antonio Ken Iannillo*, Arthur Gervais†, and Radu State*
*SnT, University of Luxembourg
†Imperial College London

***Abstract*—Smart contracts are Turing-complete programs that are executed across a blockchain. Unlike traditional programs, once deployed, they cannot be modified. As smart contracts carry more value, they become more of an exciting target for attackers. Over the last years, they suffered from exploits costing millions of dollars due to simple programming mistakes. As a result, a variety of tools for detecting bugs have been proposed. Most of these tools rely on symbolic execution, which may yield false positives due to over-approximation. Recently, many fuzzers have been proposed to detect bugs in smart contracts. However, these tend to be more effective in finding shallow bugs and less effective in finding bugs that lie deep in the execution, therefore achieving low code coverage and many false negatives. An alternative that has proven to achieve good results in traditional programs is hybrid fuzzing, a combination of symbolic execution and fuzzing.**

In this work, we study hybrid fuzzing on smart contracts and present CONFUZZIUS, the first hybrid fuzzer for smart contracts. CONFUZZIUS uses evolutionary fuzzing to exercise shallow parts of a smart contract and constraint solving to generate inputs that satisfy complex conditions that prevent evolutionary fuzzing from exploring deeper parts. Moreover, CONFUZZIUS leverages dynamic data dependency analysis to efficiently generate sequences of transactions that are more likely to result in contract states in which bugs may be hidden. We evaluate the effectiveness of CONFUZZIUS by comparing it with state-of-the-art symbolic execution tools and fuzzers for smart contracts. Our evaluation on a curated dataset of 128 contracts and a dataset of 21K real-world contracts shows that our hybrid approach detects more bugs than state-of-the-art tools (up to 23%) and that it outperforms existing tools in terms of code coverage (up to 69%). We also demonstrate that data dependency analysis can boost bug detection up to 18%.

***Index Terms*—Ethereum, smart contracts, hybrid fuzzing, data dependency analysis, genetic algorithm, symbolic execution**

I. INTRODUCTION

The inception of immutable, blockchain-based smart contracts has shown how to enable multiple mistrusting parties to trade and interact without relying on a centralized, trusted third party. The immutability of a contract is crucial: if at least one of the engaging parties were allowed to modify a digital contract, the contract's trust would vanish. Unlike traditional legal contracts, smart contracts do not allow a dispute resolution with a neutral third party. Most importantly, smart contracts cannot be nullified — parties cannot revoke any deployed smart contract, even if its code figures undeniable software bugs. Therefore, this very immutability comes at a price: smart contracts must be tested extensively before exposing them and their users to significant monetary value. In the past, simple vulnerabilities (e.g. missing access control [36]) and subtle vulnerabilities (e.g. reentrancy [40]) have led to losses exceeding many tens of millions of USD.

We can verify the behavior of a smart contract with four different approaches. (i) *Unit testing* requires manual effort to cover the different sections of the code, but it unveils only a limited number of bugs within the test cases. (ii) *Symbolic execution* analyzes contract behavior abstractly but performs slowly on complex contracts (path explosion problem). (iii) *Static analysis* does not execute code and over-approximates the contract behavior — it can capture the entire contract execution surface, but it exhibits false positives that must be manually inspected. Finally, (iv) *fuzzing* tests a contract reasonably fast by automatically generating test cases, with a generally lower false positive rate than static analysis. Fuzzing, however, can suffer from low code coverage, especially when input is fuzzed at random and hence does not overcome simple input sanity verification.

When fuzzing smart contracts, we face the following three challenges: 1) *input generation*, 2) *stateful exploration*, and 3) *environmental dependencies*. When it comes to input generation, the input space can be significantly broad. However, the solution might be limited to a specific point. For example, if a condition requires an input value of type `uint256` to equate to 42, then the probability of randomly generating 42 as input is tremendously small. Moreover, smart contracts are stateful applications, i.e. the execution may depend on a state that is only achievable following a specific sequence of inputs. Finally, the runtime environment of smart contracts exposes additional inputs related to the underlying blockchain protocol, such as the current block timestamp or other contracts deployed on the blockchain. As a result, the execution flow of smart contracts may depend on environmental information besides transactional information.

We solve these three challenges as follows. In tandem with the fuzzing procedure, we employ symbolic taint analysis to generate path constraints on tainted inputs. Once we detect that the fuzzer is not progressing, we activate a constraint solver to solve the constraint in question. We collect this solution within a mutation pool, from which the fuzzer can draw to move past the challenging contract condition. Existing hybrid fuzzing approaches, e.g. Driller [43], cease the fuzzer when they are stuck and switch to concolic execution to get past the complex condition. Then, they restart the fuzzer once passed the condition. Our approach keeps the fuzzer running and only uses constraint solving to generate inputs on the fly, which will

eventually be picked by the fuzzer via the mutation pools. In addition to constraint solving, we perform a path termination analysis to purge irrelevant inputs from the mutation pools. To deal with the statefulness of smart contracts, we chose to take advantage of the selection and crossover operators of genetic algorithms. Genetic algorithms follow three main steps: selection, crossover, and mutation. The selection operator's task is to choose two individuals from the population, which are afterwards combined by the crossover operator to create two new individuals. The challenge here is to generate meaningful combinations of inputs. Therefore, data dependencies between individuals guide our selection and crossover operators that accept two individuals only if they follow a *read-after-write* (RAW) data dependency. Finally, to solve the third and last challenge, we instrument the execution environment (i.e. the Ethereum Virtual Machine) to fuzz environmental information and model the input to a contract as a tuple consisting of transactional *and* environmental data.

Contributions. Our main contributions are as follows:

- To the best of our knowledge, we propose the first design of a hybrid fuzzer for smart contracts.
- We present a novel method to efficiently create meaningful sequences of inputs at runtime by leveraging dynamic data dependencies between state variables.
- We introduce ConFuzzius, the first implementation of a hybrid fuzzer for smart contracts.
- We evaluate ConFuzzius on a set of 128 curated smart contracts as well as 21K real-world smart contracts, and demonstrate that our approach not only detects more vulnerabilities (up to 23%) but also achieves more code coverage (up to 69%) than existing symbolic execution tools and fuzzers.

II. Background

This section provides the required background on Ethereum smart contracts and fuzzing in order to better understand the approach proposed in this work.

A. Ethereum Smart Contracts

Smart Contracts. Ethereum [49] enables the execution of so-called *smart contracts*. These are fully-fledged programs stored and executed across the Ethereum blockchain, a network of mutually distrusting nodes. Ethereum supports two types of accounts, externally owned accounts (i.e. user accounts) and contract accounts (i.e. smart contracts). Smart contracts are different from traditional programs in many ways. They own a balance and are identifiable via a 160-bit address. They are developed using a dedicated high-level programming language, such as Solidity [48], that compiles into low-level bytecode. This bytecode gets interpreted by the Ethereum Virtual Machine. By default, smart contracts cannot be removed or updated once deployed. It is the task of the developer to implement these capabilities before deployment. The deployment of smart contracts and the execution of smart contract functions occurs via transactions. The data field of a transaction includes both, the name of the function to be executed and its arguments. Transactions are created by user accounts and afterwards broadcast to the network. They contain a sender and a recipient. The latter can be the address of a user account or a contract account. Besides carrying data, transactions may also carry a monetary value in the form of ether (Ethereum's own cryptocurrency).

Ethereum Virtual Machine. The Ethereum Virtual Machine (EVM) is a purely stack-based, register-less virtual machine that supports a Turing-complete set of instructions. Although the instruction set allows for Turing-complete programs, the instructions' capabilities are limited to the sole manipulation of the blockchain's state. The instruction set provides a variety of operations, ranging from generic operations, such as arithmetic operations or control-flow statements, to more specific ones, such as the modification of a contract's storage or the querying of properties related to the transaction (e.g. sender) or the current blockchain state (e.g. block number). Ethereum uses a *gas* mechanism to assure the termination of contracts and prevent denial-of-service attacks. The gas mechanism associates costs to the execution of every single instruction. When issuing a transaction, the sender specifies how much gas they are willing to spend to execute the smart contract. This amount is known as the *gas limit*.

B. Fuzzing

Evolutionary Fuzzing. Fuzzing, or fuzz testing, is an automated software testing technique that finds vulnerabilities in programs by feeding malformed or unexpected data as input to programs, executing them, and monitoring the effects. Evolutionary fuzzing aims at converging towards the discovery of vulnerabilities by using a genetic algorithm (GA). A generation of test cases is defined as a *population*, whereas a single test case is an *individual*. In short, every individual of a generation is evaluated based on a fitness function. At the end of each generation, solely the fittest individuals are allowed to breed, following Darwin's idea of natural selection, or "survival of the fittest". Eventually, the individuals will trigger vulnerabilities while converging towards an optimal solution. We briefly describe the main steps of a GA (see Algorithm 1). We start by creating an initial population of individuals, either generated at random or seeded via heuristics, and compute their fitness values (*line 1*). Based on the fitness value, we select two individuals from the current population, which act as parents for breading (*line 5*). Then, we apply crossover and mutation operators on the parents to generate two new individuals, also denoted as offsprings (*lines 6-7*). The generation of new individuals continues until the new population reaches the same size as the current one (*line 4*). Finally, the new population's fitness values are computed, and we replace the current population with the new population (*lines 9-10*). This entire process is repeated until a termination condition is met (*line 3*), e.g. the maximum number of generations is reached or a maximum amount of time has passed.

Algorithm 1 Pseudo-Code of a Genetic Algorithm

```
Create initial population and compute its fitness
Set initial population as current population
while termination condition is not met do
    while new population < current population do
        Select two parents from current population
        Recombine parents to create two new offsprings
        Mutate offsprings and add them to new population
    end while
    Compute fitness of new population
    Replace current population with new population
    Create a new empty population
end while
```

Hybrid Fuzzing. Although fuzzing is one of the most effective approaches to find vulnerabilities, it often has difficulties in getting past complex path conditions, resulting in low code coverage. A popular alternative to fuzzing is symbolic execution. It works by abstractly executing a program and supplying abstract symbols rather than actual (*concrete*) values. The execution will then generate symbolic formulas over the input symbols, which can be solved by a constraint solver to prove satisfiability and produce concrete values. In theory, symbolic execution is capable of discovering and exploring all potential paths in a program. However, in practice, symbolic execution is often not scalable since the number of explorable paths becomes exponential in more extensive programs (*path explosion problem*). Another limitation of symbolic execution is the limited capability to interact with the execution environment. Programs often interact with their environment by performing calls to libraries, for example. Correctly modeling these calls and other environmental information is extremely challenging. The goal of hybrid fuzzing, or hybrid fuzz testing, is to take advantage of both worlds. Hybrid fuzzing starts by performing traditional fuzzing until it saturates, *i.e.*, the fuzzer is not capable of covering any new code after running some predetermined number of steps. Hybrid fuzzing then automatically switches to symbolic execution to perform an exhaustive search for uncovered branching conditions. As soon as the symbolic execution finds an uncovered branching condition, it solves it, and the hybrid fuzzer reverts to fuzzing. The interleaving of fuzzing and symbolic execution counts on shallow program paths' quick execution via fuzzing and the execution of complex program paths via symbolic execution.

III. Overview

This section discusses the three main challenges of fuzzing smart contracts via a motivating example and presents our solution towards solving these challenges.

A. Motivating Example

Suppose a user participated in an initial coin offering (ICO) on the blockchain and now owns a number of tokens. Now assume the user wants to sell a certain amount of their tokens at a variable price that increases 1 ether per day. Fig. 1 shows a possible implementation of an Ethereum smart contract using

```
1  interface Token {
2    function transferFrom(address sender, address
         recipient, uint256 amount) external
         returns (bool);
3    function allowance(address owner, address
         spender) external view returns (uint256);
4  }
5
6  contract TokenSale {
7    uint256 start = now;
8    uint256 end = now + 30 days;
9    address wallet = 0xcafebabe...;
10   Token token = Token(0x12345678...);
11
12   address owner;
13   bool sold;
14
15   function Tokensale() public {
16     owner = msg.sender;
17   }
18
19   function buy() public payable {
20     require(now < end);
21     require(msg.value == 42 ether + (now - start)
           / 60 / 60 / 24 * 1 ether);
22     require(token.transferFrom(this, msg.sender,
           token.allowance(wallet, this)));
23     sold = true;
24   }
25
26   function withdraw() public {
27     require(msg.sender == owner);
28     require(now >= end);
29     require(sold);
30     owner.transfer(address(this).balance);
31   }
32 }
33
```

Fig. 1. Example of a vulnerable token sale smart contract. Lines highlighted in red represent complex conditions, whereas lines highlighted in gray illustrate read-after-write data dependencies and finally, lines highlighted in blue depict environmental dependencies.

Solidity. The smart contract allows a user to sell its tokens to an arbitrary user on the Ethereum blockchain. The contract sells the tokens to the first buyer willing to pay 42 ether, plus 1 ether for each day since the start of the sale. Moreover, the token sale should last no longer than 30 days. In this example, the smart contract acts as a simple mediator that automatically settles the trade between the user owning the tokens and the user willing to buy the tokens without both users requiring to know or trust each other. Smart contract based ICOs often follow a standard that is known as ERC-20 [15]. This standard provides an interface that standardizes function names, parameters, and return values. For example, the standard includes a function called `transferFrom`, which allows a user to transfer a limited amount of tokens to an arbitrary user on behalf of the owning user. Another example is the function `allowance`, which returns the number of tokens that a user can spend on behalf of the owning user. The smart contract in Fig. 1 works as follows. An arbitrary user can call

the function buy to purchase the tokens for 42 ether and a fee of 1 ether for the number of days passed since the launch of the token sale. The contract will automatically transfer the tokens by calling the function transferFrom on the ICO's contract. After the purchase, the smart contract owner can call the function withdraw to retrieve the 42 ether and the fee of the purchase.

The contract contains two vulnerabilities, one known as *block dependency* and another known as *leaking ether*. The latter is enabled via a bug in the function Tokensale (see line 15 in Fig. 1). Before Solidity version 0.4.22, the only way of defining a constructor was to create a function with the same name as the contract. The function Tokensale is supposed to be the constructor of the contract TokenSale. Due to a typo, the names do not match, and the compiler does not consider the function as the contract's constructor. As a result, the function Tokensale is considered a public function that any user on the blockchain can call. This type of programming mistake has led to multiple attacks in the past [1]. The first vulnerability, namely block dependency, occurs when the transfer of ether depends on block information, such as the timestamp (see line 28 in Fig. 1). Malicious miners can alter the timestamp of blocks that they mine, especially if they can gain advantages. Although miners cannot set the timestamp smaller than the previous one, nor can they set the timestamp too far ahead in the future, developers should still refrain from writing contracts where the transfer of ether depends on block information. The second vulnerability, namely leaking ether, occurs whenever a contract allows an arbitrary user to transfer ether, despite having never transferred ether to the contract before. The following sequence of transactions triggers both vulnerabilities:

- t_0: A non-malicious user calls the function buy with a value equals to 42 ether + fee;
- t_1: An attacker calls the function Tokensale;
- t_2: The same attacker calls the function withdraw after 30 days.

When running the above example using ILF [18] (a state-of-the-art smart contract fuzzer), it is not capable of finding the two vulnerabilities even after 1 hour. Inspecting the code coverage reveals that ILF achieves only 39%. For comparison, ConFuzzius achieves roughly 95% code coverage and correctly identifies the two vulnerabilities in less than 10 seconds.

B. Input Generation

Generating meaningful inputs is crucial for automated software testing. Fuzzers generate inputs in order to execute not-yet-executed code. This generation can be completely random (black-box fuzzers) or driven by runtime information (grey-box fuzzers). In both cases, the primary approach is to mutate previous inputs to generate new inputs to test. Thus, finding the right heuristics is of fundamental importance to efficiently explore the target input space and, eventually, find latent bugs in the code. However, real-world programs tend to contain conditions that are hard to trigger. These complex conditions need to be addressed by fuzzers in order to execute as much

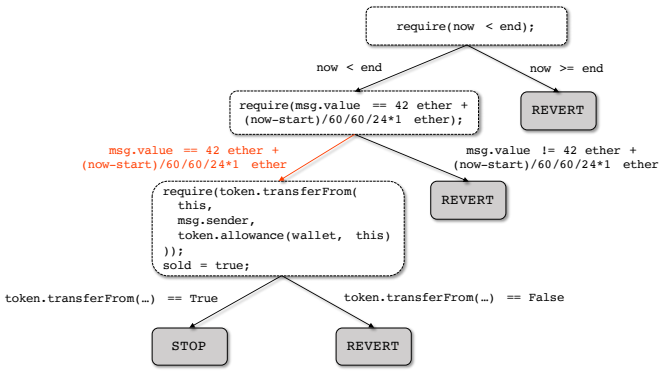

Fig. 2. Control-flow graph of the function buy(), where the complex condition is highlighted in red.

code as possible. Line 21 in Fig. 1 provides an example of a complex condition. Function buy requires the transaction value to be equal to 42 ether along with a variable fee that depends on the number of days that have past since the launch of the token sale. Fig. 2 illustrates the control-flow graph (CFG) of the function buy along with its branching conditions. The complex condition is highlighted in red in the CFG. A fuzzer following a traditional random strategy will fail to get past this condition since it will generate the desired value only once every 2^{256} trials.

Existing smart contract fuzzers such as Harvey [50] instrument the code and compute cost metrics for every branch to mutate the inputs. Our approach applies constraint solving to generate values for complex conditions on-demand. However, our fuzzer does not directly propagate these values, but instead, it stores them in so-called mutation pools. Mutation pools manage a set of values that the fuzzer can use to get past complex conditions. Every function has its own set of mutation pools, namely a mutation pool per function argument, transaction argument (e.g. transaction value), and environmental argument (e.g. block timestamp). Initially, all the pools are empty, and the fuzzer uses randomly generated values to feed the target functions. Once the fuzzer cannot discover new paths, it activates the constraint solver to generate new values. We use symbolic taint analysis to create the expressions required by the constraint solver to generate new values. We introduce taint in the form of a symbolic value whenever we come across an input during execution. This symbolic value is then propagated throughout the program execution, thereby forming step-by-step a symbolic expression that reflects the constraints on the particular input. Solving these expressions will result in new values that will be added to the mutation pools. The fuzzer will then pick these values from the mutation pools and generate new inputs that execute new paths. In Fig. 1, once ConFuzzius realizes that the code coverage is not increasing, it activates the constraint solver, which outputs the value 42 along with the current fee depending on the current block timestamp. It adds it to the mutation pool that manages the transaction value for the function buy. The value will be then picked up from the mutation pool by the fuzzer

in the next round, and the execution of the transaction will evaluate the condition at line 21 to `True`, which results in getting past the missing branch and executing new lines of code.

C. Stateful Exploration

Due to the transactional nature of blockchains, smart contract fuzzers must consider that each transaction may have a different output depending on the contract's current state, i.e. all the previously executed transactions. Appropriately combining multiple transactions is necessary to generate states that trigger the execution of new branches. Ethereum smart contracts have, besides a volatile memory model, also a persistent memory model called *storage*, which allows them to keep state across transactions. For example, the global variables `end`, `wallet`, `token`, `owner`, and `sold` in Fig. 1 are storage variables and their values might change across transactions. Let us consider the two vulnerabilities mentioned earlier. An attacker will only be able to extract the funds via the function `withdraw`, if the two variables `owner` and `sold` contain the address of the attacker and `True`, respectively. However, this is only possible if the functions `buy` and `Tokensale` are called before the function `withdraw`. Thus only a particular combination of the three functions will trigger the two vulnerabilities. Although this example may seem straightforward, automatically finding the right combination of function calls within contracts with many functions can become challenging as the number of possible combinations grows exponentially. We base our solution on a simple observation: a transaction influences the output of a subsequent set of transactions if and only if it modifies a storage variable that one of the subsequent transactions will use. This property is a known data dependency called *read-after-write* (RAW) [19]. In the first step, ConFuzzius traces all the storage reads and writes performed by a transaction along with the storage locations. Afterwards, ConFuzzius combines transactions so that transaction a is executed after transaction b, only if a reads from the same storage location where b writes to. The fuzzer always executes the combination of transactions on a clean state of the contract. Thus, a transaction sequence contains only transactions that change the state used by one of the subsequent transactions within the same sequence by construction. In the example of Fig. 1, ConFuzzius will progressively learn that:

- `buy` reads variable `end` and writes to variable `sold`;
- `Tokensale` writes to variable `owner`;
- `withdraw` reads variable `owner` and variable `sold`.

Using the information learned above and combining transactions based on RAW dependencies, ConFuzzius will eventually create the following transaction sequence:

`buy()` → `Tokensale()` → `withdraw()`

The directed graph in Fig. 3 presents the RAW dependencies to generate all the possible combinations. The graph shows that the functions `buy` and `Tokensale` must be executed before the function `withdraw`, but that the order between the two can be arbitrary.

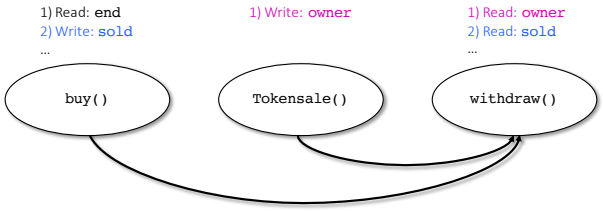

Fig. 3. A dependency graph illustrating the read-after-write (RAW) data dependencies contained in Fig. 1. A node represents a smart contract function and an edge indicates a RAW dependency between the two functions.

D. Environmental Dependencies

The execution of a smart contract does not only depend on the transaction arguments or the contract's current state. A smart contract's control-flow can also depend on input originating from the execution environment (e.g. a block's timestamp). Let us consider the contract in Fig. 1. Even though the function `withdraw` has no input argument, the transfer of the balance is bound to some requirements. The requirement at line 28 is only satisfied if the transaction that triggered the function call is part of a block created 30 days after the contract's deployment. Thus, the condition is bound to the mining mechanism of the Ethereum blockchain. While users submit transactions to the blockchain, miners aggregate them into blocks and distribute them to other nodes upon validation. When executing the transactions included in the block, the EVM accesses the block information contained therein. Block information includes the block hash, the miner's address, the block timestamp, the block number, the block difficulty, and the block gas limit. We solve this challenge by modeling this information as a fuzz-able input. These inputs follow the same fuzzing procedure as transaction inputs. We modified the EVM in order to be able to inject the fuzzed block information during the execution of the smart contract. However, modeling block information as fuzz-able inputs is not enough. The EVM also permits to call other contracts deployed on the blockchain. Thus the control-flow of a smart contract may depend on the result of calling other contracts. Consider line 29 in Fig. 1, where the state variable `sold` is required to be set to `True` in order for the attacker to be able to retrieve the funds. The variable `sold` can only be set to `True` if the two contract calls at line 22 (e.g. `token.allowance` and `token.transferFrom`) are successful. We solve this challenge in a similar way by instrumenting calls to contracts and modeling return values as fuzz-able inputs. Our modified EVM then injects the fuzzed return values at runtime.

IV. Design and Implementation

In this section, we provide details on the overall design and implementation of ConFuzzius.

Fig. 4. Overview of CONFUZZIUS's hybrid fuzzing architecture. The shadowed boxes represent the three main components and form together a feedback loop.

A. Overview

CONFUZZIUS's architecture consists of three main modules: the evolutionary fuzzing engine, the instrumented EVM, and the execution trace analyzer. Fig. 4 provides a high-level overview of CONFUZZIUS's architecture and depicts its individual components. CONFUZZIUS has been implemented in Python with roughly 6,000 lines of code[1]. CONFUZZIUS takes as input the source code of a smart contract and a blockchain state. The latter is in the form of a list of transactions and is optional. The blockchain state is convenient for fuzzing already deployed smart contracts or contracts that need to be initialized with a specific state. CONFUZZIUS begins by compiling the smart contract to obtain the Application Binary Interface (ABI) and the EVM runtime bytecode. The evolutionary fuzzing engine then starts by generating individuals for the initial population, based on the smart contract's ABI. After that, the engine follows a standard genetic algorithm (i.e., selection, crossover, and mutation) and propagates the newly generated individuals to the instrumented EVM. The instrumented EVM then executes these individuals and forwards the resulting execution traces to the execution trace analyzer. Next, the execution trace analyzer performs several analyses, e.g. symbolic taint analysis, data dependency analysis, etc. The execution trace analyzer is also responsible for triggering the constraint solver, running the vulnerability detectors, updating the mutation pools, and feeding information related to code coverage and data dependencies to the evolutionary fuzzing engine. This process is repeated until at least one of the two termination conditions is met: a given number of generations has been generated, or a given amount of time has passed. Finally, CONFUZZIUS outputs a report containing information about the code coverage and the vulnerabilities that it detected.

B. Evolutionary Fuzzing Engine

In the following, we provide details on the encoding, initialization, fitness evaluation, selection, combination, and mutation of individuals.

[1]Source code is available at https://github.com/christoftorres/ConFuzzius

Encoding Individuals. One of the most important decisions to make while implementing an evolutionary fuzzer is deciding on the representation of individuals. Improper encoding of individuals can lead to poor performance [27]. Fig. 5 illustrates our encoding of individuals. Vulnerabilities are usually triggered either by sending a single transaction or a sequence of transactions to a smart contract. However, transactions alone are not enough to trigger vulnerabilities (see Section III-D). Specific vulnerabilities depend on the execution environment to be in a specific state. Thus, our encoding represents an individual as a sequence of inputs. Every input consists of an environment and a transaction. Both are encoded as key-value mappings. The environment includes block information such as the current timestamp and block number, but it also includes call return values, data sizes, and external code sizes. The latter three are encoded as an array of mappings, where a contract address maps to a mutable value (e.g. a call result or a size). The transaction includes the address of the sending account ($from$), the transaction amount ($value$), the maximum amount of gas for the contract to execute (gas_limit), and the input data for the contract to execute ($data$). The input data is represented as an array of values where the first element is always the function selector, and the remaining elements represent the function arguments. The function selector is computed using the ABI and extracting the first four bytes of the Keccak (SHA-3) hash of the function signature. As an example, the function `test(string a, uint b)` has the string `test(string,uint)` as its function signature, which after hashing and extracting the first four bytes, results in `0x7d6cdd25` being its function selector.

Initial Population. The population is initialized with N individuals, each of which initially contains only a single input (i.e. a single transaction). The function selector to be included in the transaction is selected in a round-robin fashion. Function arguments are generated based on their type, which we obtain through the ABI. Depending on the type and size (i.e. fixed or non-fixed) of the argument, we apply different

generate test inputs automatically [20], [37]. On the other hand, KLEE [6] and SAGE [17] are white-box fuzzers, that execute code in a controlled environment. Driller [43] is a hybrid fuzzer that leverages selective concolic execution in a complementary manner. Symbolic execution based fuzzers produce meaningful inputs but tend to be slow [7]–[9], [35]. Fuzzers such as LibFuzzer [39], FuzzGen [21] and FUDGE [2] focus on fuzzing libraries, which cannot run as standalone programs, but instead are invoked by other programs.

Smart Contract Fuzzing. CONTRACTFUZZER [22] generates inputs based on a list of input seeds. While CONTRACTFUZZER deploys an entire custom testnet to fuzz transactions, CONFUZZIUS is more efficient and solely emulates the EVM. Moreover, CONFUZZIUS does not rely on user-provided input seeds but instead analyzes the execution traces and feeds constraints related to the execution to a constraint solver in order to generate new values specific to the contract under test. ECHIDNA [10] is a property-based testing tool for smart contracts that leverages grammar-based fuzzing. ECHIDNA requires user-defined predicates in the form of Solidity assertions and does not automatically check for vulnerabilities. HARVEY [50] predicts new inputs based on instruction-granularity cost metrics. In contrast, CONFUZZIUS exploits lightweight symbolic execution when the population fitness does not increase (see Section IV-D). Further, HARVEY fuzzes transaction sequences in a targeted and demand-driven way, assisted by an aggressive mode that directly fuzzes the persistent state of a smart contract. Instead, CONFUZZIUS relies on the read-after-write dependencies to guide the selection and crossover operators to create meaningful transaction sequences efficiently (see Section IV-B). ILF [18] is based on imitation learning, which requires a learning phase prior to fuzzing. ILF consists of a neural network that is trained on transactions obtained by running a symbolic execution expert over a broad set of contracts. CONFUZZIUS does not have the overhead of the learning phase and uses on-demand constraint solving while actively fuzzing the target. Moreover, ILF is limited to the knowledge that it learned during the learning phase, meaning that ILF has issues in getting past program conditions that require inputs that were not part of the learning dataset. CONFUZZIUS does not have this issue as it learns tailored inputs per target while fuzzing. SFUZZ [32] is an AFL based smart contract fuzzer, whereas ETHPLOIT [51] is a fuzzing based smart contract exploit generator. Both SFUZZ and ETHPLOIT have been developed concurrently and independently of CONFUZZIUS. While SFUZZ follows a random strategy to create transaction sequences, ETHPLOIT uses static taint analysis on state variables to create meaningful transaction sequences. However, static taint analysis has the disadvantage of being imprecise and analyzing parts that are not executable. Despite SFUZZ using a genetic algorithm as CONFUZZIUS, it employs a different encoding of individuals. SFUZZ only models block number and timestamp as environmental information. ETHPLOIT, on the other hand, instruments the EVM in a similar way to CONFUZZIUS. However, ETHPLOIT does not fuzz the size of external code or contract call return values.

Smart Contract Symbolic Execution. Apart from fuzzing, several other tools based on symbolic execution were proposed to assess the security of smart contracts [28] [33] [31] [45] [46] [46] [26] [16]. MPRO [52] combines symbolic execution and data dependency analysis to deal with the scalability issues that symbolic execution tools face when trying to handle the statefulness of smart contracts. MPRO has been developed concurrently and independently from CONFUZZIUS. There are two significant differences in our approach. First, MPRO retrieves data dependencies using static analysis and therefore requires source code, whereas CONFUZZIUS tries to infer data dependencies from bytecode. Second, MPRO works in two separate steps, first, it infers data dependencies via static analysis, and then it applies symbolic execution. CONFUZZIUS, on the other hand, applies a dynamic approach and infers data dependencies while fuzzing. ETHRACER [25] uses a hybrid approach with a converse strategy by primarily using symbolic execution to test a smart contract and using fuzzing only for producing combinations of transactions to detect vulnerabilities such as transaction order dependency. CONFUZZIUS's fuzzing strategy, compared to ETHRACER, is not entirely random but based on read-after-write dependencies, yielding faster and more efficient combinations.

Smart Contract Static Analysis. Besides symbolic execution and fuzzing, other works based on static analysis were proposed to detect smart contract vulnerabilities. ZEUS [23] is a framework for automated verification of smart contracts using abstract interpretation and model checking. SECURIFY [47] uses static analysis based on a contract's dependency graph to extract semantic information about the program bytecode and then checks for violations of safety patterns. Similarly, VANDAL [5] is a framework designed to convert EVM bytecode into semantic logic relations in Datalog, which can then be queried for vulnerabilities.

VII. CONCLUSION

We presented CONFUZZIUS, the first hybrid fuzzer for smart contracts. It tackles the three main challenges of smart contract testing: *input generation*, *stateful exploration*, and *environmental dependencies*. We solved the first challenge by combining evolutionary fuzzing with constraint solving to generate meaningful inputs. The second challenge is solved by leveraging data dependency analysis across state variables to generate purposeful transaction sequences. Finally, we solved the third challenge by modeling block related information (e.g. block number) and contract related information (e.g. call return values) as fuzzable inputs. We run CONFUZZIUS and other state-of-the-art fuzzers and symbolic execution tools for smart contracts against a curated dataset of 128 contracts and a dataset of 21K real-world smart contracts. Our results show that our hybrid approach not only detects more bugs than existing state-of-the-art tools (up to 23%), but also that CONFUZZIUS outperforms these tools in terms of code

coverage (up to 69%) and that data dependency analysis can boost the detection of bugs (up to 18%).

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments and feedback. This work was partly supported by the Luxembourg National Research Fund (FNR) under grant 13192291 and is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830927.

REFERENCES

[1] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[2] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985.

[3] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[4] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[5] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[6] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[7] 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.

[8] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 499–513.

[9] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 515–530.

[10] Crytic, "Echidna: Ethereum fuzz testing framework," February 2020. [Online]. Available: https://github.com/crytic/echidna

[11] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[12] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 530–541.

[13] Ethereum Foundation, "Py-EVM - A Python implementation of the Ethereum Virtual Machine," August 2019. [Online]. Available: https://github.com/ethereum/py-evm

[14] Etherscan, "Contracts With verified source codes only," November 2020, https://etherscan.io/contractsVerified.

[15] V. B. Fabian Vogelsteller, "Erc-20 token standard," February 2015, https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md.

[16] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[17] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008.

[18] 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*, ser. CCS '19. New York, NY, USA: ACM, 2019, pp. 531–548. [Online]. Available: http://doi.acm.org/10.1145/3319535.3363230

[19] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[20] A. K. Iannillo, R. Natella, D. Cotroneo, and C. Nita-Rotaru, "Chizpurfle: A gray-box android fuzzer for vendor service customizations," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017, pp. 1–11.

[21] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[22] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 259–269.

[23] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *NDSS*, 2018.

[24] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[25] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," *arXiv preprint arXiv:1810.11605*, 2018.

[26] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1317–1333.

[27] G. E. Liepins and M. D. Vose, "Representational issues in genetic optimization," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 2, no. 2, pp. 101–115, 1990.

[28] 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*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[29] Michal Zalewski, "American Fuzzy Lop (AFL)," December 2016. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[30] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[31] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," in *9th annual HITB Security Conference*, 2018.

[32] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," *arXiv preprint arXiv:2004.08563*, 2020.

[33] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *arXiv preprint arXiv:1802.06038*, 2018.

[34] T. of Bits, "Manticore - symbolic execution tool," jun 2018, https://github.com/trailofbits/manticore.

[35] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[36] S. Petrov, "Another parity wallet hack explained," nov 2017, https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c.

[37] 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.

[38] C. Reitwiessner, "Solidity 0.6.6 Documentation: Layout of State Variables in Storage," July 2020, https://solidity.readthedocs.io/en/v0.6.6/miscellaneous.html#layout-of-state-variables-in-storage.

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

[40] D. Siegel, "Understanding the dao attack," jun 2016, https://www.coindesk.com/understanding-dao-hack-journalists/.

[41] R. Sivaraj and T. Ravichandran, "A review of selection methods in genetic algorithm," *International journal of engineering science and technology*, vol. 3, no. 5, pp. 3792–3797, 2011.

[42] SmartContractSecurity, "SWC Registry," November 2020, https://swcregistry.io/.

[43] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, no. 2016, 2016, pp. 1–16.

[44] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[45] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 664–676. [Online]. Available: http://doi.acm.org/10.1145/3274694.3274737

[46] C. F. Torres, M. Steichen *et al.*, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1591–1607.

[47] 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*. ACM, 2018, pp. 67–82.

[48] G. Wood, "Solidity 0.6.11 Documentation," July 2020, https://solidity.readthedocs.io/en/v0.6.11/.

[49] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[50] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," *arXiv preprint arXiv:1905.06944*, 2019.

[51] Q. Zhang, Y. Wang, J. Li, and S. Ma, "Ethploit: From fuzzing to efficient exploit generation against smart contracts," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 116–126.

[52] W. Zhang, S. Banescu, L. Pasos, S. Stewart, and V. Ganesh, "Mpro: Combining static and symbolic analysis for scalable testing of smart contract," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 456–462.

APPENDIX A
VULNERABILITY DETECTORS

We elaborate on the implementation details of our 10 vulnerability detectors below.

Assertion Failure. We detect an assertion failure by checking if the execution trace contains an `ASSERTFAIL` or `INVALID` instruction.

Integer Overflow. Detecting integer overflows is not trivial, since not every overflow is considered harmful. Integer overflows may also be introduced by the compiler for optimization purposes. Therefore, we only consider an overflow as harmful, if it modifies the state of the smart contract, i.e. if the result of the computation is written to storage or is used to send funds. We follow the approach by Torres et al. [45] and start by analyzing if the execution trace contains an `ADD`, `MUL` or `SUB` instruction. We then extract the operands from the stack and use these to compute the result of the arithmetic operation. Afterwards, we check if our result is equivalent to the result that has been pushed onto the stack. If they are not the same, we know that an integer overflow has occurred and we keep track of the overflow by tainting the result of the computation. We report an integer overflow if the tainted result flows into an `SSTORE` instruction or a `CALL` instruction, as these instructions will result in updating the blockchain state.

Reentrancy. A reentrancy occurs whenever a contract calls another contract, and that contract calls back the original contract. We detect reentrancy by first checking if the execution trace contains a `CALL` instruction whose gas value is larger than 2,300 units and where the amount of funds to be transferred is a symbolic value or a concrete value that is larger than zero. Finally, we report a reentrancy if we find an `SLOAD` instruction that occurs before the `CALL` instruction and an `SSTORE` instruction that occurs after the `CALL` instruction, and which shares the same storage location as the `SLOAD` instruction.

Transaction Order Dependency. We detect transaction order dependency by checking if there are two execution traces with different senders, where the first execution trace writes to the same storage location from which the second execution trace reads.

Block Dependency. We detect a block dependency by checking if the execution trace contains either a `CREATE`, `CALL`, `DELEGATECALL`, or `SELFDESTRUCT` instruction, that is either control-flow or data dependent on a `BLOCKHASH`, `COINBASE`, `TIMESTAMP`, `NUMBER`, `DIFFICULTY`, or `GASLIMIT` instruction.

Unhandled Exception. We detect unhandled exceptions by first checking if the execution trace contains a `CALL` instruction that pushes to the stack the value 1 as a result of the call. A value of 1 means that an error occurred during the call (i.e. an exception). Afterwards, we check if the result of the call flows into a `JUMPI` instruction. If the result does not flow into a `JUMPI` instruction until the end of the execution trace, then this means that the exception of the call was not handled and we report an unhandled exception.

Unsafe Delegatecall. We detect an unsafe delegate call by checking if there is an execution trace that contains a `DELEGATECALL` instruction and terminates with a `STOP` instruction, but whose sender is an attacker address. Attacker and benign user addresses are generated at the start by the fuzzer.

Leaking Ether. We detect the leaking of ether by checking if the execution trace contains a `CALL` instruction, whose recipient is an attacker address that has never sent ether to the contract in a previous transaction and has never been passed as a parameter in a function by an address that is not an attacker.

Locking Ether. We detect the locking of ether by checking if a contract can receive ether but cannot send out ether. To check if a contract cannot send ether, we check if the runtime bytecode of the contract does not contain any `CREATE`, `CALL`, `DELEGATECALL`, or `SELFDESTRUCT` instruction. To check if a contact can receive ether, we check if the execution trace has a transaction value larger than 0 and terminates with a `STOP` instruction.

Unprotected Selfdestruct. Similar to the leaking ether or unsafe delegatecall vulnerability detectors, this detector relies on attacker accounts. We detect an unprotected selfdestruct by

checking if the execution trace contains a `SELFDESTRUCT` instruction where the sender of the transaction is an attacker and its address has not been previously passed as an argument by a benign user.