Towards Smart Hybrid Fuzzing for Smart Contracts

Christof Ferreira Torres[1], Antonio Ken Iannillo[1], Arthur Gervais[2], Radu State[1]
[1]*SnT, University of Luxembourg*
[2]*Imperial College London*

***Abstract*—Smart contracts are Turing-complete programs that are executed across a blockchain network. Unlike traditional programs, once deployed they cannot be modified. As smart contracts become more popular and carry more value, they become more of an interesting target for attackers. In recent years, smart contracts suffered major exploits, costing millions of dollars, due to programming errors. As a result, a variety of tools for detecting bugs has been proposed. However, majority of these tools often yield many false positives due to over-approximation or poor code coverage due to complex path constraints. Fuzzing or fuzz testing is a popular and effective software testing technique. However, traditional fuzzers tend to be more effective towards finding shallow bugs and less effective in finding bugs that lie deeper in the execution.**

In this work, we present CONFUZZIUS, a hybrid fuzzer that combines evolutionary fuzzing with constraint solving in order to execute more code and find more bugs in smart contracts. Evolutionary fuzzing is used to exercise shallow parts of a smart contract, while constraint solving is used to generate inputs which satisfy complex conditions that prevent the evolutionary fuzzing from exploring deeper paths. Moreover, we use data dependency analysis to efficiently generate sequences of transactions, that create specific contract states in which bugs may be hidden. We evaluate the effectiveness of our fuzzing strategy, by comparing CONFUZZIUS with state-of-the-art symbolic execution tools and fuzzers. Our evaluation shows that our hybrid fuzzing approach produces significantly better results than state-of-the-art symbolic execution tools and fuzzers.

***Index Terms*—Ethereum, smart contracts, hybrid fuzzing**

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 trust in the contract would vanish. Contrary to traditional legal contracts, smart contracts do not allow a dispute resolution with the help of a neutral third party. More importantly, smart contracts are not subject to all-encompassing legislation — smart contracts hence cannot be nullified, even if their code figures undeniable software bugs or vulnerabilities, parties cannot revoke any deployed smart contract. This very immutability, therefore, 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, such as missing access control [29], as well as subtle vulnerabilities, such as reentrancy [32], have led to losses exceeding many tens' of millions of USD.

Smart contract (and software) testing can be categorized into four distinct techniques: (i) *Unit tests* require manual effort and cover particular contract behavior, which uncovers only a limited number of bugs within those test-cases. (ii) *Symbolic execution* analyzes contract behavior abstractly; however, it performs slowly on complex contracts (path explosion problem). (iii) *Static analysis* does not execute code and over-approximates the contract behavior — it can, therefore, capture the entire contract execution surface; however, it exhibits false positives that must be manually inspected. Finally, (iv) *fuzzing* allows testing a contract reasonably fast, with a generally lower false-positive rate than static testing. Fuzzing, however, can suffer from low code coverage, especially if input is fuzzed at random and hence does not overcome simple input sanity verification.

When analyzing smart contracts, we face three challenges:

1) **Input generation**: While the possible input space can be significant, the solution might be limited to a specific value. 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.
2) **Stateful exploration**: Smart contracts are stateful applications, i.e. the execution may depend on a specific contract's state that is only achievable following a specific sequence of inputs.
3) **Environmental dependencies:** The runtime environment of smart contracts, exposes them to additional inputs related to the underlying blockchain protocol, such as for example, the current block timestamp or block number. As a result, the execution flow of smart contracts may depend on this environmental information.

The first two challenges are commonly faced by traditional fuzzers and are taken into consideration by current state-of-the-art smart contract fuzzers [14], [17], [41], while to the best of our knowledge the third challenge has not yet been formalized by existing literature on smart contract fuzzing.

We solve the three challenges as follows. Parallel to the fuzzing procedure, we employ symbolic taint analysis to generate path constraints. Once we detect that the fuzzer is not progressing, we activate a constraint solver to solve the path 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 [35], 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 makes use of constraint solving to generate complex inputs that will even-

tually be picked by the fuzzer through the mutation pools. In addition to constraint solving, we perform a path termination analysis to purge irrelevant input from the mutation pools. To solve the statefulness of smart contracts, we chose to take advantage of the crossover operator of genetic algorithms. Genetic algorithms follow three steps: selection, crossover, and mutation. The crossover operation combines individuals (a sequence of inputs) to create new individuals. The challenge is to generate meaningful combinations of inputs. Our crossover operator is guided by data dependencies between individuals, and only combines two individuals together if combined they follow a *Read-after-Write* data dependency. Finally, in order 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 not only as a transaction but as a tuple, consisting of transactional data *and* environmental data.

Contributions. Our main contributions are as follows:

- We propose a novel hybrid fuzzing approach that uses on-demand constraint solving together with mutation pools to guide the fuzzer through complex conditions.
- We present a new method based on data dependencies to create meaningful sequences of inputs that efficiently fuzz the state of a smart contract.
- We introduce ConFuzzius, the first implementation of a hybrid fuzzer for Ethereum smart contracts.
- We evaluate ConFuzzius on a benchmark of real-world contracts, and demonstrate that it detects more vulnerabilities and achieves significantly more code coverage (14% more than ILF) than existing tools.

II. Background

In this section, we provide background on Ethereum smart contracts and hybrid fuzzing.

A. Ethereum Smart Contracts

Smart Contracts. Ethereum [40] enables the execution of so-called *smart contracts*. These are fully-fledged programs that are stored and executed across the Ethereum blockchain, a network of mutually distrusting nodes. Ethereum supports two types of accounts, 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 [39], 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 as well as 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 value.

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 capabilities of the instructions 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 arithmetics 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 makes use of a *gas* mechanism to assure the termination of contracts and to 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 he or she is willing to spend for the execution of the smart contract. This amount is known as the *gas limit*. Gas can be converted to ether (Ethereum's internal currency) through the so-called *gas price* of a transaction. The gas price multiplied by the gas limit, determines the maximum amount of ether that the user will be able to pay for the inclusion of his or her transaction into the blockchain.

B. Hybrid fuzzing

Fuzzing. Fuzz testing, or fuzzing, 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. A fuzzer can be classified in several ways. It can be generation-based or mutation-based, depending on whether inputs are generated from scratch or by modifying existing ones. It can be smart or dumb, depending on whether it is aware of the structure of the input or not. Also, a fuzzer can be white-box, grey-box, or black-box, depending on whether the fuzzer is aware of the whole program structure, some parts of the program structure or no program structure. Fuzzing has developed as one of the most effective approaches to find vulnerabilities in programs. However, fuzzers often have difficulties in getting past complex paths conditions contained in programs and therefore achieve low code coverage.

Symbolic Execution. A popular alternative to fuzzing is symbolic execution. It works by abstractly executing a program, and supplying and tracking abstract symbols rather than actual (*concrete*) values. The execution will then generate symbolic formulas over the input symbols, which in turn can potentially be solved by a constraint solver to produce the concrete values. Symbolic execution is capable of discovering and exploring all potential paths in a program. However, symbolic execution is not practically scalable since the number of explorable

```
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 end = now + 30 days;
8    address wallet = 0x12345678...;
9    Token token = 0xcafebabe...;
10
11   address owner;
12   bool sold;
13
14   function Tokensale() public {
15     owner = msg.sender;
16   }
17
18   function buy() public payable {
19     require(now < end);
20     require(msg.value == 42 ether);
21     sold = true;
22     require(token.transferFrom(this, msg.sender,
           token.allowance(wallet, this)));
23   }
24
25   function withdraw() public {
26     require(msg.sender == owner);
27     require(sold);
28     require(now >= end);
29     owner.transfer(address(this).balance);
30   }
31 }
```

Fig. 1. Example of a vulnerable tokensale smart contract.

paths becomes exponential in more extensive programs (path explosion problem).

Hybrid Fuzzing. The goal of hybrid fuzzing, or hybrid fuzz testing, is to take advantage of both worlds. Traditional hybrid fuzzing starts by performing fuzzing until it saturates, that is, does not produce any new coverage points after running some predetermined number of steps. The hybrid fuzzer then automatically switches to symbolic execution to perform an exhaustive search for an uncovered coverage point. As soon as it reaches one, the hybrid fuzzer then reverts to fuzzing. The interleaving of fuzzing and symbolic execution uses both the advantage of fuzzing to quickly execute shallow program paths and the advantage of symbolic execution to explore more complex program paths.

III. Overview

In this section, we discuss the three main challenges of fuzzing smart contracts through a simple motivating example, and present our approach to solve them.

A. Motivating Example

Suppose a user participated in an initial coin offering (ICO) on the blockchain and now owns a large number of tokens. Now assume the user wants to sell a certain amount his or her tokens at a fixed price to an arbitrary user on the blockchain. Fig. 1 shows a possible implementation of an Ethereum smart contract in Solidity, that allows a user to sell its tokens to an arbitrary user on the Ethereum blockchain. The idea of the contract is to sell the tokens to the first buyer that is willing to pay 42 ether within 30 days. The smart contract acts as a simple mediator for the trade between the user owning the tokens and the user willing to buy the tokens. Smart contract based ICOs follow a given standard that is known as ERC20 [11]. The standard provides an interface that standardizes function names, parameters, and return values. For example, the standard includes a function called `transferFrom`, that allows a user to transfer a limited amount of tokens to an arbitrary user on behalf of the owning user. In our case, it is our smart contract that is allowed to transfer a specific number of tokens to an arbitrary user on behalf of the user that is currently wanting to sell its tokens. Another example is the function `allowance`, which returns the remaining number of tokens that a user is allowed to 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 the contract will automatically do the transfer of the tokens by calling the function `transferFrom` on the contract of the ICO. Then, after the purchase, the owner of the smart contract can simply call the function `withdraw` to retrieve the 42 ether of the purchase. However, the contract contains two vulnerabilities, one known as *block dependency* and another one known as *leaking ether*. The first vulnerability 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 their blocks, especially if they can gain advantages by doing so. 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 occurs whenever a contract allows an arbitrary user to transfer ether, despite the user 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;
- t_1: An attacker calls the function `Tokensale`;
- t_2: The same attacker calls the function `withdraw` after 30 days.

The two vulnerabilities are enabled through a bug (see line 14 in Fig. 1) in the function `Tokensale`. 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`. However, due to a typo in the function name, the contract name and function name do not match, and therefore the compiler does not consider the function as being the constructor of the contract. As a result, the function `Tokensale` is considered a normal function

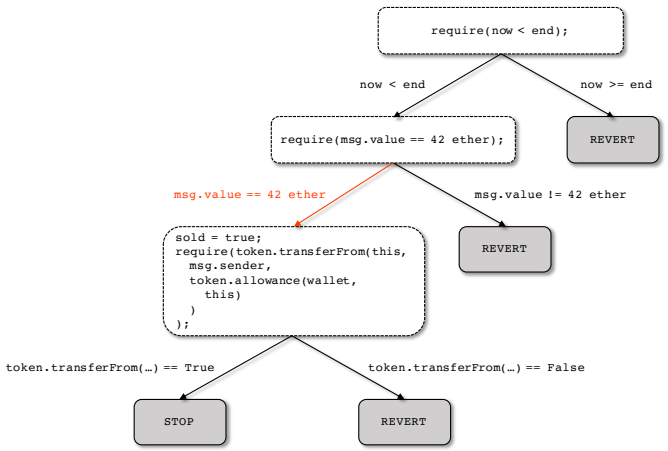

Fig. 2. Control-flow graph of the function `buy()`, where the complex condition `msg.value == 42 ether` is highlighted in red.

that is callable by any user on the blockchain. This type of programming mistake has led to multiple attacks in the past [1]. When running the above example using, ILF [14] (an imitation learning based 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 53%. For comparison, CONFUZZIUS achieves roughly 97% code coverage and correctly identifies the two vulnerabilities in less than 5 seconds.

B. Input Generation

Generating meaningful inputs is crucial for automated software testing. Fuzzers randomly generate new 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 values in order to generate new values 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 code as possible. Line 20 in Fig. 1 gives an example of a complex condition. Function `buy` requires the transaction value to be equals to 42 ether. Fig. 2 illustrates the control-flow graph (CFG) of the function `buy`. 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 value 42 only once every 2^{256} trials.

Existing smart contract fuzzers such as HARVEY [41] 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, the 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 a mutation pool per function argument and input field (e.g. transaction value or transaction sender). Initially, all the pools are empty and the fuzzer uses randomly generated inputs to feed the target functions. Once the fuzzer is not able to discover new paths, it activates the constraint solver in order to generate new values. Symbolic taint analysis creates the expressions required by the constraint solver in order 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, forming a symbolic expression that reflects the constraints on the input. Solving these expressions will result in new values that are added to the mutation pools. The fuzzer will then use these values when available for the specific function arguments, to 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 and adds it to the mutation pool of the first argument of function `buy`. Eventually, the value will be picked up by the fuzzer in the next round, and the execution of the transaction will evaluate the condition at line 20 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 take into consideration that each transaction may have a different output depending on the current state of the contract, i.e. all the previously executed transactions. Appropriately combining multiple transactions is necessary in order 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* that 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` are equals to the address of the attacker and `True`, respectively. However, this is only the case if the functions `buy` and `Tokensale` are called before. Thus only a combination of the three transactions will be able to trigger the vulnerabilities. Although this case may seem simple, automatically finding the right combination of transactions within contracts with a large number of functions can become very 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 following transactions is going to use. This property is a known data dependency called Read-after-Write (RAW) [15]. CONFUZZIUS traces all the storage reads and writes performed by a transaction along with the storage locations. Afterwards, it tries to combine transactions that read from a particular

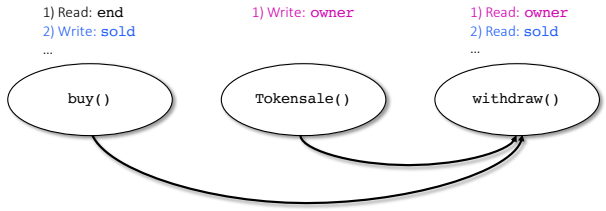

Fig. 3. A dependency graph illustrating the Read-after-Write (RAW) dependencies contained in Fig. 1. A node represents a smart contract function and an edge indicates a RAW dependency between two functions.

storage location after another transaction writes to the same storage location. The fuzzer always executes the combination of transactions on a clean state of the contract. By construction, a transaction sequence contains only transactions that change the state used by one of the subsequent transactions within the same sequence. 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.

D. Environmental Dependencies

The execution of a smart contract does not only depend on the transaction arguments or the contract's current state. The control-flow of a smart contract can also depend on input originating from the execution environment (e.g. 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 triggering the function call is part of a block that was created 30 days after the deployment of the contract. 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 hash of the block, the miner's address, the block timestamp, the block number, the block difficulty, and the block gas limit. We solve this challenge by modelling this information as a fuzzable input. These inputs follow the same fuzzing procedure as transaction inputs. We had to modify the EVM in order to inject the fuzzed block information during the execution of the smart contract.

IV. DESIGN AND IMPLEMENTATION

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

A. Overview

CONFUZZIUS's architecture is divided into two main parts: the evolutionary fuzzing engine and the execution trace analyzer. Figure 4 provides a high-level overview of CONFUZZIUS's architecture and depicts its different 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 later 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 in order to obtain the Application Binary Interface (ABI) and the EVM runtime bytecode. Afterwards, the evolutionary fuzzing engine starts by generating an initial population of individuals, 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 a number of analyses, such as symbolic taint analysis or data dependency analysis, from the execution traces that it received. Moreover, the execution trace analyzer is also responsible for triggering the constraint solver, running the vulnerability detectors, as well as maintaining 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 created, or a given number of seconds has passed. Finally, CONFUZZIUS outputs a report containing information about the code coverage and the vulnerabilities that it detected.

B. Evolutionary Fuzzing Engine

The evolutionary fuzzing engine is one of the main components of our hybrid fuzzing system. Evolutionary fuzzing aims at converging towards the discovery of vulnerabilities by using a genetic algorithm (GA), to produce successive generations of test cases. A generation of test cases is defined as a *population*, whereas a single test case is defined as 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, thus following the idea of natural selection, also known as Darwinism or the "survival of the fittest". Eventually, the individuals will trigger vulnerabilities while converging towards an optimal solution. In the following, we briefly describe the main steps that a typical GA follows (see Algorithm 1).

[1]The source code will be publicly available under an open-source license.

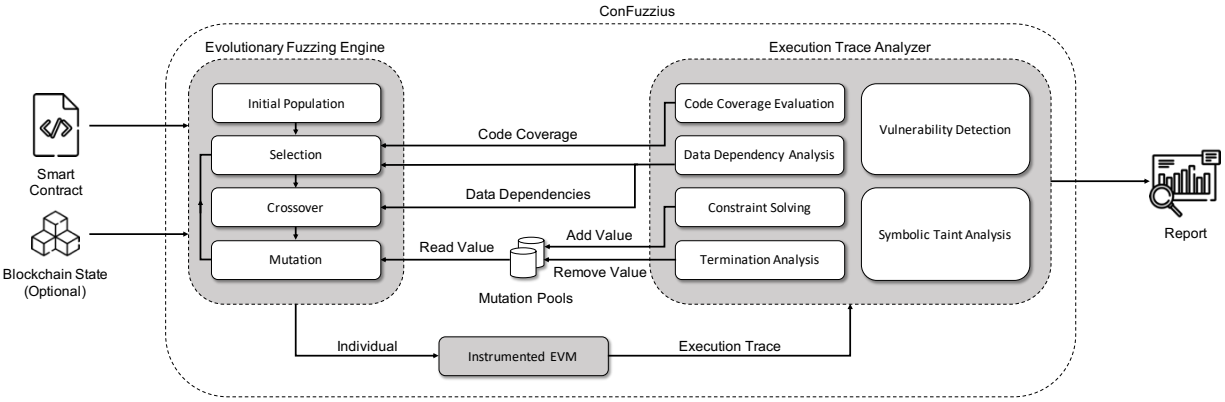

Fig. 4. Overview of CONFUZZIUS's hybrid fuzzing architecture.

Algorithm 1 Pseudo-Code of a Genetic Algorithm

```
1:  Create initial population and compute its fitness
2:  Set initial population as current population
3:  while termination condition is not met do
4:      while new population < current population do
5:          Select two parents from current population
6:          Recombine parents to create two new offsprings
7:          Mutate offsprings and add them to new population
8:      end while
9:      Compute fitness of new population
10:     Replace current population with new population
11:     Create a new empty population
12: end while
```

We start by creating an initial population of individuals, that are either generated at random or seeded via heuristics, and compute their fitness values (*line 1*). Then, based on the fitness, we select two individuals from the current population, which act as parents for breading (*line 5*). Afterwards, 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 of the current one (*line 4*). Then, the fitness values of the new computation are computed and we replace the current population by the new population (*lines 9-10*). The whole process continues 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. In the following, we provide details on the encoding, initialization, fitness evaluation, selection, combination, and mutation of individuals.

Encoding Individuals. One of the most important decisions to make while implementing an evolutionary fuzzer, is deciding on the representation of the individuals. It has been observed that the improper encoding of individuals can lead to a poor performance [21]. Figure 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). Certain vulnerabilities depend on the execution environment to be in a specific state. For this reason, our encoding represents an individual as a sequence of inputs, where every input consists of an environment and a transaction. Both are represented via a key-value mapping. The environment includes the current timestamp and block number. 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. 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 strategies to generate valid arguments for each function. For example, if the argument type is a fixed size `uint32`, then we randomly choose a value either from the valid input domain (e.g. between 0 and $2^{32}-1$) or from a set of inputs that trigger edge cases of the valid input domain (e.g. 0, 1, 4294967295, etc.). The population is reinitialized whenever there has been no increase in terms of code coverage for the past k generations. This introduces back diversity and helps procrastinating premature convergence when the population has become homogeneous.

Fitness Evaluation. The fitness evaluation of individuals plays a crucial role in evolutionary fuzzing. The computation of the fitness function is done repeatedly and must be therefore

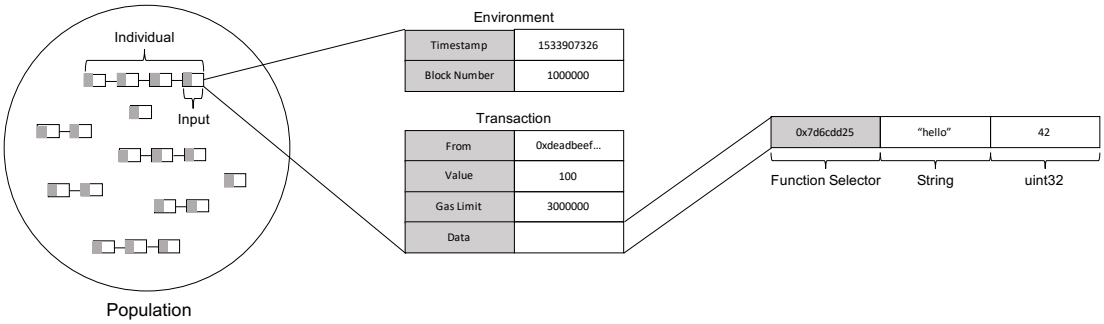

Fig. 5. Encoding of our population and its individuals. The shadowed boxes represent immutable values, whereas the non-shadowed boxes represent values that are mutable.

sufficiently fast. A slow computation can adversely make the fuzzing exceptionally slow. The fitness function is supposed to represent the landscape of the problem. In general, evolutionary fuzzers aim to achieve complete coverage of the code. While obtaining full code coverage does not necessarily mean that all vulnerabilities will be found, it is certainly true that no vulnerabilities will be found in code that has not been explored. Our fitness function is based on branch coverage (a form of code coverage) and data dependencies. We define our fitness function for an individual i as follows:

$$fit(i) = fit_{branch}(i) + fit_{write}(i) \tag{1}$$

The fitness fit_{branch} is computed by counting the number of branches that remain unexplored by the individual. In order to compute this number, we keep track of all the branches that have been executed so far by all the individuals. Then, we iterate through the execution trace of the individual and analyze every conditional jump instruction (i.e. `JUMPI` instruction). A conditional jump has always two destinations, one for the `True` branch and one for the `False` branch. We obtain the jump destination of the `True` branch by extracting it from the stack, and the jump destination of the `False` branch by increasing the program counter by one. We increase the individual's fitness value fit_{branch} by 10 for every jump destination that is not in our list of executed branches. The idea behind this approach is to give individuals that require more exploration a high fitness, since these individuals will allow us to explore new parts of the contract. However, this metric alone is not enough, since we are also interested in preserving individuals that allow us to create useful sequences of transactions (e.g. sequences with Read-after-Write dependencies), although they may have been already explored extensively. Therefore, we compute the fitness fit_{write}, which takes this into account by using the data dependencies detected by our analyzer. We start with a fit_{write} of 0 and add a value of 1 for every write to storage that the individual has performed during execution. By adding 1 instead of 10, we put less importance on the fitness fit_{write} and put more importance on fit_{branch}. Ultimately, it is the combination of the two that allows us to drive the genetic algorithm towards the exploration of unexplored code. Thus, the final fitness of an individual is the sum of fit_{branch} and fit_{write}.

Selection. The process of choosing two individuals for the crossover step is called selection. A number of selection operators has been proposed in literature [34]. We choose linear ranking selection as our selection operator, since it considers the whole population during selection and not just a subset as it is for example the case in tournament selection. In linear ranking selection, individuals with high fitness values will be ranked high and those with low fitness values will eventually have lower ranks. Then the individuals are selected with a probability that is linearly proportional to the rank of the individuals in the population. In other words, the worst individual has a rank of 1, the second-worst a rank of 2, etc. The best performing individual has a rank of N, where N is the size of the population. This results in all the individuals having a chance of being selected, although the higher-ranked individuals will be slightly preferred. After having selected the two individuals, we propagate them to the crossover operator.

Crossover. The purpose of our crossover operator is to create two new individuals by recombining the input sequences of two existing individuals. Instead of randomly combining two individuals together, we combine two individuals only if one individual performs a write to a storage location from which the other individual performs a read (also know as read-after-write dependency). There are only two possible combinations, individual a combined with individual b, or vice versa. If a combination yields a read-after-write relation, then we combine both individuals by first selecting the individual whose input sequence performs the write and then appending the individual whose input sequence performs the read. This results in concatenating individuals, rather than splitting them apart and swapping their input sequences. This way we preserve the read-after-write relations within the individuals themselves and create individuals with new read-after-write relations. If there is no relation between two individuals, then we simply return one of the two individuals unmodified. However, it should be noted that individuals are not always combined even though they might have a read-after-write relation. Individuals are

combined based on a given crossover probability p_c. Moreover, to prevent individuals from growing indefinitely large, we check before combining, if the sum of their length exceeds a maximum size l, and only combine them if the sum of their length is lower or equals to l.

Mutation. The mutation randomly modifies parts of a single individual, in order to create a new individual. Mutation is used to introduce diversity in the population. Our mutation operator works by iterating through the sequence of inputs of an individual, and mutating every environmental and transnational value based on a given mutation probability p_m. A value can be mutated in two ways, either by replacing the original value with a random one, or by replacing the original value with a value from a *mutation pool*. Mutation pools act as a form of short-term memory. They allow the fuzzer to reuse values that have been previously observed or learned during past executions. There are in total six different mutation pools, one per transactional and environmental value. Thus, our fuzzer has a mutation pool for senders, amounts, gas limits, function arguments, timestamps and block numbers. All mutation pools are implemented as a mapping of a function selector and a circular buffer, except for the mutation pool on function arguments. The implementation is similar, except that we do not directly map the function selector to a circular buffer, but to another mapping that maps to an argument index, and only then to a circular buffer. Thus, the pool for function arguments, first maps to a function selector, then to an argument index, and then to a circular buffer. This is because functions can have more than just one argument and we want to keep track of interesting values for every argument separately. Circular buffers help us ensure that the values contained therein are rotated in a round-robin like fashion and that old values are overwritten by newer ones (i.e. mimicking short-term memory). Our buffers can hold by default up to 10 values. The mutation pools are initially all empty, except for the mutation pool tracking the amounts, which gets initialized with the values 0 and 1. When mutating a transactional or environmental value, we first check if the associated mutation pool is empty. If the pool is empty, then we inject a randomly generated value based on the type information extracted from the ABI. Otherwise, we inject the current value contained at the head of the circular buffer and rotate the buffer.

C. Instrumented EVM

The EVM is responsible for executing the transactions generated by the individuals on the runtime bytecode of the contract that is under test. Its efficiency has a significant impact on the overall performance of the fuzzer. Hence, the EVM must achieve a high processing rate of transactions. Every official Ethereum client implementation provides the capability to deploy a smart contract locally and send transactions to it. However, all of these clients require transactions to be mined and passed via a JSON-RPC interface or being encoded using the Recursive Length Prefix (RLP) format. The actual EVM execution time is negligible compared to the effort that it takes to encode and decode a transaction to and from the RLP format. So instead, we decided to reuse an existing official Python implementation of the EVM [10], and reuse it within our fuzzer. This removes the burden from mining blocks as well as encoding transactions, and thus significantly speeds up the execution. Moreover, we slightly modified the EVM in order to be able to retrieve the execution trace. An execution trace consists of an array, where every element contains the name of the executed instruction, the program counter, the execution stack, the call-stack depth and a flag stating if an internal error occurred during execution. The EVM itself is by default stateless and uses the blockchain to preserve states. However, since we are not interested in the internal mechanisms of the Ethereum blockchain, we implemented a simple storage emulator that is used by our EVM, in order to preserve the state changes that are performed during execution. All state changes are kept in memory to further improve the speed of execution. Besides preserving the state of smart contracts, the storage emulator also allows us to modify environmental information such as the block number or the block timestamp. Moreover, the storage emulator also allows us to create snapshots of the current state of the EVM. This enables us to quickly reset the state of the EVM to an initial state, without having to redeploy the smart contract every time from scratch when executing the transactions of an individual.

D. Execution Trace Analyzer

The execution trace analyzer performs a variety of analyses from the execution trace obtained by the instrumented EVM. The analyses include, code coverage evaluation, data dependency analysis, symbolic taint analysis, vulnerability detection, constraint solving, and termination analysis. In addition to that, it also manages the values stored within the mutation pools and provides the information required by the evolutionary fuzzing engine to compute the fitness and perform the crossover of individuals. Finally, it is also responsible for generating a report containing statistics about the code coverage and the vulnerabilities that have been found.

Code Coverage Evaluation. Code coverage is not only necessary for computing the fitness of individuals, but also for detecting when the evolutionary fuzzing engine gets "stuck", and constraint solving should be applied. The code coverage is computed by counting the number of unique program counter values contained within the execution trace.

Data Dependency Analysis. The fitness evaluation as well as the crossover operator require information about data dependencies. The data dependency analysis returns an object consisting of two sets, one containing all the storage indexes that have been read from, and another one containing all the storage indexes that have been written to. We retrieve these indexes by iterating through the execution trace and checking for `SLOAD` and `SSTORE` instructions. We extract for both instructions the storage index from the stack and add them to the respective set, i.e. indexes from an `SLOAD` are added to

the set of reads, whereas indexes from an SSTORE are added to the set of writes.

Symbolic Taint Analysis. The purpose of our symbolic taint analysis is to produce symbolic constraints that are later useful for other components such as constraint solving and vulnerability detection. We introduce symbolic values and track their flow between instructions. We leverage light dynamic taint analysis by injecting taint only whenever we come across instructions that can be fuzzed, such as for example CALLDATALOAD, CALLVALUE or TIMESTAMP. The taint is propagated throughout the interpretation of the execution trace along with the value that was assigned to it. We faithfully propagate taint across stack and memory, and storage. The propagation of taint across storage allows us to do inter-transactional taint analysis. We implemented the stack using an array structure that follows LIFO logic. To represent memory and storage, we simply used a Python dictionary that maps memory and storage addresses to values. As the EVM is a stack-based and register-less virtual machine, as such the operands of instructions are always passed via the stack. Therefore, our taint propagation method identifies the operands of each EVM bytecode instruction and propagates the taint according to the semantics of each instruction as defined in [40]. The taint propagation logic follows an over-tainting policy, which simply tags the output of an instruction as tainted if at least one of the inputs of the instruction are tainted.

Constraint Solving. There are situations where the evolutionary fuzzing engine converges since it is not able to advance past a particular condition. We often say that the landscape has become flat. The role of the constraint solver is to generate a valid input that allows the evolutionary fuzzing engine to get past the complex condition. The symbolic taint analysis tries to build a logical formula that describes the execution path of an execution trace, thereby reducing the problem of reasoning about the execution to the domain of logic. These logical formulas are often called path constraints. We implemented our own light weight symbolic execution engine, that only executes instructions related to arithmetic operations (e.g. ADD, MOD, EXP), comparison logic (e.g. LT, EQ) and bitwise logic (e.g. AND, NOT). The engine consists of an interpreter loop that gets instructions from the execution trace and symbolically executes them. The loop continues until all the instructions contained in the execution trace have been executed. As a result, we obtain a logical formula for the current execution trace. To produce a valid input that satisfies the complex condition, we simply negate the last condition contained in the logical formula, substitute the symbolic variables in the rest of the logical formula with concrete values that have been used as input to trigger the execution trace, and use the Z3 SMT solver [9] to produce a solution. Concretization helps us reduce the complexity of the formula and therefore alleviate the path explosion problem. The solution is then added to a mutation pool, along with the previous solution that was not capable of triggering the other part of the branch. Eventually, in one of the following generations, the mutation operator will pick up the new solution and our evolutionary fuzzing engine will now be able to get past the complex condition.

Termination Analysis. The execution traces may contain valuable feedback on the validity of the inputs generated by the fuzzer. Our fuzzer makes use of the execution traces as a way to obtain feedback and to learn which inputs are meaningful and which are not. The goal of our termination analysis is to inspect the execution traces for opcodes that indicate either correct or incorrect termination of an execution. Invalid inputs will result in the execution trace terminating with a REVERT, INVALID or ASSERTFAIL instruction, whereas valid inputs will result in the execution terminating with either a STOP or RETURN instruction. If we detect that an execution terminates incorrectly, then we analyze the last path condition before the termination and infer from the symbolic values which inputs are responsible for the termination. We then remove the responsible input values from the the respective mutation pools, since mutating using input values that result in an incorrect termination will prevent the fuzzer from exploring deeper parts of the code.

Vulnerability Detection. We detect vulnerabilities by analyzing the execution traces and using the information returned by the different components, such as the symbolic taint analysis or the data dependency analysis. We define a detector per vulnerability and currently implement 11 different detectors: arbitrary memory access, assertion failure, integer overflow, reentrancy, transaction order dependency, block dependency, unhandled exception, unsafe delegatecall, leaking ether, locking ether, and unprotected selfdestruct. A detailed explanation on the implementation of each vulnerability detector is provided in Appendix A. The detection capabilities of our fuzzer can easily be extended by simply adding more detectors.

V. Evaluation

In this section, we provide details on the benchmark we used and explain our experimental setup. Then, we compare ConFuzzius to two current state-of-the-art fuzzers as well as three symbolic execution tools for smart contracts and asses the effectiveness and performance of ConFuzzius by answering two research questions. Finally, we highlight potential threats to the validity of our evaluation.

A. Dataset

We performed our evaluation using the benchmark proposed by Wüstholz et al. [41]. The benchmark consists of 27 real-world open-source smart contracts collected from 17 different GitHub repositories. The repositories have been selected based on their popularity on GitHub and among the Ethereum community (e.g. Ethereum Name Service, ConsenSys *multi-sig* wallet, MicroRaiden payment service). Furthermore, the benchmark also includes contracts that have been hacked in the past or are known to be malicious or buggy (e.g. The

TABLE I

SECURITY ANALYSIS TOOLS EVALUATED IN THIS WORK. TOOLS MARKED WITH ● SUPPORT THE VULNERABILITY DETECTOR, WHILE TOOLS MARKED WITH ○ DO NOT SUPPORT THE VULNERABILITY DETECTOR. TOOLS MARKED WITH ◐ CLAIM TO SUPPORT THE DETECTOR BUT CANNOT BE VERIFIED.

Toolname	Type	Open-Source	Requires ABI	Vulnerability Detectors										
				AM	AF	IO	RE	TD	BD	UE	UD	LE	LO	US
HARVEY [41]	Fuzzer	✗	✗	●	●	◐	◐	○	○	◐	○	○	○	○
ILF [14]	Fuzzer	✓	✓	○	○	○	○	○	●	●	●	●	●	●
OYENTE [22]	Symbolic	✓	✗	○	●	●	●	●	●	○	○	○	○	●
MYTHRIL [26]	Symbolic	✓	✗	●	●	●	●	○	●	●	●	●	○	●
MANTICORE [28]	Symbolic	✓	✗	○	●	●	●	●	●	●	●	●	○	●

DAO, Parity wallet, and USCC[2]). Moreover, the authors affirm that they followed the guidelines on evaluating fuzzers when selecting the smart contracts [19]. Table V in Appendix B, provides an overview of the 27 selected contracts. The first column lists the benchmark IDs and the second column, the project acronym. The third and fourth columns show the number of public functions and the lines of source code (LoSC), respectively. Finally, the last column provides a small description of the project. The benchmark is very diverse, as it contains contracts ranging from 57 LoSC up to 3065 LoSC.

B. Experimental Setup

We followed the published guidelines by Klees et al. [19] on evaluating fuzz testing. For each experiment, we performed 24 runs, each with independent seeds and a time limit of 1 hour per run. The experiments were carried out using our high-performance computing infrastructure. We run our experiments on ten different nodes with 128 GB of memory. Every node is running CentOS Linux release 7.6.1810 and has 2 Intel® Xeon® Gold 6132 CPUs with 14 cores, each clocked at 2.60 GHz. We run CONFUZZIUS with a population size that is double the number of functions contained in the ABI of the respective contract that is under test. Moreover, we set the probability of crossover and the probability of mutation, to 0.9 and 0.1, respectively. Further, we set the number of generations to be held before reinitializing the population to 10 and the maximum length for the individuals to 5. We used Z3, version 4.8.5, as our constraint solver with a timeout of 100 milliseconds per Z3 request. We compare CONFUZZIUS to the security tools listed in Table I. We selected two advanced smart contract fuzzers (HARVEY and ILF), and three popular open-source symbolic execution tools for smart contacts (OYENTE v0.2.7, MYTHRIL v0.21.20 and MANTICORE v0.3.2). Although HARVEY is not open-source at the time of writing, by using the benchmark provided by Wüstholz et al. [41] we can compare ourselves and other tools to HARVEY. Both ILF and CONFUZZIUS require the ABI as input in order to be able to fuzz the contract under test. Finally, Table I also compares the different types of vulnerabilities detected by each of the tools. CONFUZZIUS implements a total of 11 vulnerability detectors. From the comparison in Table I we can see that none of the analyzed security tools is able to detect all of the 11 vulnerabilities that are currently detectable by CONFUZZIUS.

[2]Underhanded Solidity Coding Contest – https://u.solidity.cc/

C. Experimental Results

RQ1: Does CONFUZZIUS achieve higher code coverage than state-of-the-art fuzzers and symbolic execution tools?

Comparing to Fuzzers. We start by comparing CONFUZZIUS to HARVEY, a grey-box fuzzer for smart contracts. HARVEY uses input prediction and demand-driven sequence fuzzing to test smart contracts efficiently. We report the instruction coverage for the 27 real-world contracts in Table II. HARVEY is not open-source, and the results listed here are taken directly from the paper [41]. CONFUZZIUS outperforms HARVEY in 21 contracts out of the 27 (i.e. about 78% of the cases).

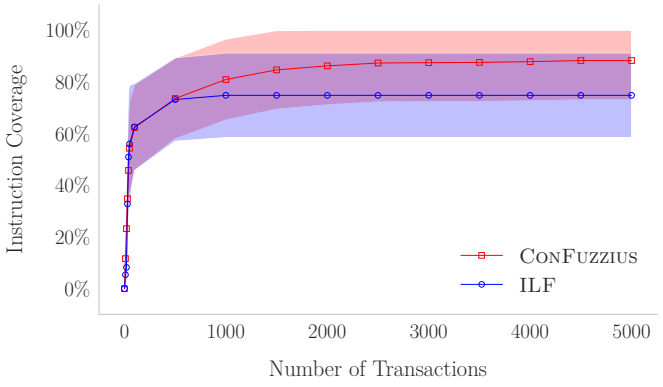

Fig. 6. Instruction coverage of CONFUZZIUS and ILF.

Next, we compare CONFUZZIUS to ILF, a fuzzer based on imitation learning. ILF uses neural networks to learn a fuzzing policy from a dataset of inputs generated by a symbolic execution expert. We evaluate ILF on the HARVEY benchmark and perform the Wilcoxon-Mann-Whitney U test [23] to determine if the differences in the results are statistically significant. We report the instruction coverage and p-values for the results on the 27 real-world contracts in Table IV in Appendix F. Traditionally, p-values below 0.05 are considered good enough to show a statistically significant difference between the two populations. However, p-values alone are not sufficient to draw throughout statistical conclusions, since they say nothing about the extent of divergence, also known as the effect size. Therefore, we also compute Vargha Delaney's A measure [38], which computes the effect size. Intuitively these show the

probability of a tool a being better than a tool b, and vice versa. We report the Vargha-Delaney effect sizes in Table IV ($A_{\text{ILF,ConFuzzius}}$ and $A_{\text{ConFuzzius,ILF}}$). Our results show that for 24 out of the 27 contracts (i.e. about 89% of the cases), CONFUZZIUS achieves significantly higher code coverage than ILF. Fig. 6 illustrates the median instruction coverage of CONFUZZIUS and ILF across the HARVEY benchmark, for a number of 5K transactions. For the first 500 transactions ILF performs slightly better than CONFUZZIUS. However, from 500 transactions onwards ILF starts converging and CONFUZZIUS consistently outperforms ILF. CONFUZZIUS achieves 88% code coverage, 14% higher than ILF. We analyzed the contracts where CONFUZZIUS achieves low coverage compared to both, HARVEY and ILF. We conclude that these contracts require transaction sequences where their inputs need to be consistent throughout the sequence. CONFUZZIUS's constraint solving approach is currently not able to detect this dependency and will generate inputs randomly. HARVEY and ILF generate inputs from a limited set of values, eventually reusing the same inputs across transactions.

Comparing to Symbolic Executors. Table III presents the instruction coverage for CONFUZZIUS, OYENTE, MYTHRIL and MANTICORE for the HARVEY benchmark. The cases in which one tool is performing better than the other differ tremendously, which makes it hard to compare to one another and draw any conclusions. Moreover, we do not want to compare CONFUZZIUS to every symbolic executor separately, but rather all together. Therefore, to evaluate which tool is performing better overall, we employ Copeland's method [31]. Copeland's method is a Condorcet voting method in which candidates are ordered based on the number of pairwise victories minus the number of pairwise defeats. Table VI in Appendix C shows the victories and defeats for all the possible pairwise combinations of the four tools. For example, we determine the winner of the pairwise comparison between OYENTE and MYTHRIL by counting the cases in which OYENTE achieves higher instruction coverage than MYTHRIL, and vice versa. This results in 7 cases where OYENTE performs better vs. 20 cases where MYTHRIL performs better. Thus, the winner of this pairwise comparison is MYTHRIL. Now, we can compute the ranking by subtracting the number of defeats from the number of victories. Table VII in Appendix D shows the number of victories, defeats, and the ranking for the four tools. The overall winner is CONFUZZIUS with 3 victories and 0 defeats. MYTHRIL is ranked second with 2 victories and 1 defeat, then OYENTE is ranked third with 1 victory and 2 defeats, and finally, MANTICORE is ranked fourth with 0 victories and 3 defeats.

Result. CONFUZZIUS achieves higher code coverage than current state-of-the-art fuzzers and symbolic execution tools with statistical significance.

RQ2: Does CONFUZZIUS find more vulnerabilities than state-of-the-art fuzzers and symbolic execution tools?

TABLE II
INSTRUCTION COVERAGE COMPARISON BETWEEN HARVEY AND CONFUZZIUS.

BID	HARVEY	CONFUZZIUS	Ratio	Total
1	**3868.0**	3105.0	0.80	10001
2	4005.5	**4936.0**	1.23	5212
3	3487.0	**4562.0**	1.31	4767
4	3773.0	**5052.5**	1.34	5335
5	3501.0	**4635.5**	1.32	4890
6	1949.0	**2330.0**	1.20	2631
7	1524.0	**1587.0**	1.04	1658
8	2205.0	**2561.0**	1.16	2934
9	3468.0	**3627.0**	1.05	4109
10	**7360.5**	6748.0	0.92	8573
11	8716.0	**8893.5**	1.02	19066
12	**5165.0**	4303.0	0.83	24303
13	**4510.0**	3417.5	0.76	12293
14	**4655.0**	3783.0	0.81	9400
15	5078.5	**5086.5**	1.00	6208
16	496.0	**505.0**	1.02	752
17	2754.0	**3072.0**	1.12	3077
18	**2930.0**	2419.0	0.83	2715
19	2611.0	**3345.0**	1.28	8164
20	3018.0	**3157.0**	1.05	3243
21	434.0	**447.0**	1.03	448
22	1274.0	**1309.0**	1.03	1317
23	2095.0	**2149.0**	1.03	3962
24	754.0	**939.0**	1.25	946
25	1192.0	**1347.0**	1.13	1351
26	1606.0	**4486.0**	2.79	5034
27	5499.5	**7752.0**	1.41	8492
Median	3018.0	**3539.0**	1.14	5959

Comparing to Fuzzers. Fig. 7 depicts the number of true and false positives on the HARVEY benchmark reported by CONFUZZIUS and the security tools analyzed in this work. We manually analyzed the results reported by each tool and checked for true and false positives per contract. When comparing CONFUZZIUS to HARVEY, we see that CONFUZZIUS identifies the same number of arbitrary memory access vulnerabilities, but fails to detect one assertion failure vulnerability. We assume that this is due to CONFUZZIUS achieving lower code coverage than HARVEY on one of the contracts, which results in CONFUZZIUS not being able to detect the assertion failure. Next, we compare CONFUZZIUS to ILF. CONFUZZIUS detects more block dependency vulnerabilities than ILF and reports no false positives on unprotected selfdestruct vulnerabilities. The former is because ILF does not consider block dependency on self-destructs. The latter is because ILF considers transactions as malicious, even if it is a benign user that sets the attacker as the destination of a self-destruct. Fig. 8 in Appendix E, illustrates the length of transactions per vulnerabilities found by CONFUZZIUS. Although most vulnerabilities were found through one single transaction, about 48% of the vulnerabilities were found through a combination of at least two transactions.

Comparing to Symbolic Executors. In Fig. 7, we see that symbolic execution tools produce a lot more false positives than fuzzers. This is because fuzzers such as CONFUZZIUS execute the program with concrete values, and false positives

TABLE III
INSTRUCTION COVERAGE COMPARISON BETWEEN CONFUZZIUS AND SYMBOLIC EXECUTION TOOLS.

BID	OYENTE	MYTHRIL	MANTICORE	CONFUZZIUS
1	2749.0	**8880.0**	1840.0	3085.0
2	3841.0	0.0	0.0	**4936.0**
3	3618.0	0.0	0.0	**4562.0**
4	3953.0	0.0	0.0	**5052.5**
5	3731.0	0.0	0.0	**4635.5**
6	2175.0	**2628.0**	2502.0	2330.0
7	1107.0	1520.0	1457.0	**1587.0**
8	2444.0	2160.0	2023.0	**2561.0**
9	2448.0	**4099.0**	3403.0	3627.0
10	4480.0	6154.0	4121.0	**6728.0**
11	5403.0	6115.0	2797.0	**8856.0**
12	3006.0	**4818.0**	3547.0	4263.0
13	2477.0	**3510.0**	2759.0	3394.0
14	5495.0	**5808.0**	0.0	3763.0
15	3935.0	**5723.0**	3821.0	5086.5
16	737.0	549.0	**738.0**	505.0
17	2550.0	3044.0	0.0	**3072.0**
18	2147.0	2176.0	0.0	**2419.0**
19	2059.0	**5340.0**	2172.0	3325.0
20	2026.0	**3231.0**	2210.0	3157.0
21	358.0	446.0	441.0	**447.0**
22	1164.0	**1311.0**	1025.0	1309.0
23	1988.0	1725.0	0.0	**2109.0**
24	610.0	**942.0**	480.0	939.0
25	1305.0	**1350.0**	1258.0	1347.0
26	2758.0	2794.0	1855.0	**4486.0**
27	5553.0	7299.0	4070.0	**7752.0**
Median	2745.0	3023.0	1575.0	**3531.0**

are not possible, assuming that the implemention of the detectors is correct. Besides, we see that CONFUZZIUS detects more vulnerabilities than the analyzed symbolic execution tools, with zero false positives. This is because CONFUZZIUS often achieves high code coverage, and in some cases more than symbolic execution tools.

Result. CONFUZZIUS detects more vulnerabilities than current state-of-the-art fuzzers and symbolic execution tools with significantly less false positives.

D. Threats to Validity

We identified threats to both, internal and external validity, due to the choice of the benchmark. We chose HARVEY's benchmark [41] from the necessity to compare CONFUZZIUS to HARVEY. However, we detected inconsistencies in the number of executed instructions and the number of total instructions. This might be due to different initial states used by CONFUZZIUS and HARVEY, which results in different results. In addition, HARVEY's benchmark might not be sufficiently generalizable. For instance, the DAO version included in the benchmark is not affected by the reentrancy attack. Therefore, we will publicly disclose the entire benchmark, seeds, and the initial state that we used for our experiments, in order to allow for the reproducibility of our results.

VI. RELATED WORK

Since its introduction by Miller et al. [25], practitioners applied fuzzing to many different and heterogeneous targets.

Software Fuzzing. KLEE [4] is a white-box fuzzer, which executes the target code within a virtual environment and forks this every time it finds a branch, in an attempt to explore all paths. SAGE [12] uses a record&replay framework to negate one of the logical conditions across a path and generates new inputs to explore different paths. American Fuzzy Loop (AFL) [24], one of the most widespread fuzzers, is based on evolutionary fuzzing and exploits execution data to guide the generation/mutation of fuzzed inputs. AFLFast [3] models the probability of fuzzing an input with a Markov chain. AFLGo [2] is a directed fuzzing solution that generates inputs to reach a given set of target program locations efficiently. Besides AFL and its offsprings, other fuzzers use evolutionary approaches to generate test inputs automatically. VUzzer [30] uses an application-aware evolutionary strategy by exploiting static analysis and dynamic taint analysis. Driller [35] is a hybrid fuzzer that leverages selective concolic execution in a complementary manner, triggering it only when the fuzzer has difficulties in exploring further the input space. Chizpurfle [16] is an evolutionary fuzzing approach that targets service APIs and introduces the concept of community, enabling the concurrent evolution of populations with individuals physically located in the same target but impossible to combine due to syntactic constraints.

Smart Contract Fuzzing. The first to propose a fuzzer for smart contracts were Jiang et al. [17]. CONTRACTFUZZER generates inputs based on the ABI. While their fuzzer uses a custom Ethereum testnet, CONFUZZIUS directly emulates the blockchain using an implementation of the EVM. Also, CONFUZZIUS does not only make use of random values, but also analyzes the execution traces (i.e. the list of executed instructions together with information about the stack) to feed a constraint solver and learn new values specific to the contact under test. ECHIDNA [8] is a property-based testing tool for grammar-based fuzzing. Wüstholz et al. propose HARVEY [41], a fuzzer that makes use of a novel method for predicting new inputs based on instruction-granularity cost metrics. CONFUZZIUS, however, exploits light symbolic execution on the execution traces when the population fitness does not increase (see Section IV-D). 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. CONFUZZIUS relies instead on the read-after-write principle of data dependencies to guide the crossover operator to create meaningful transaction sequences efficiently (see Section IV-B). ILF [14] is a smart contract fuzzer based on imitation learning. It introduces a learning phase prior to the fuzzing phase. ILF consists of a neural network that is trained on transactions obtained by running a symbolic execution expert over a broad set of contracts. Instead, CONFUZZIUS does not have the overhead of the learning phase and uses on-demand constraint solving while actively fuzzing the target.

Smart Contract Symbolic Execution. Apart from fuzzing,

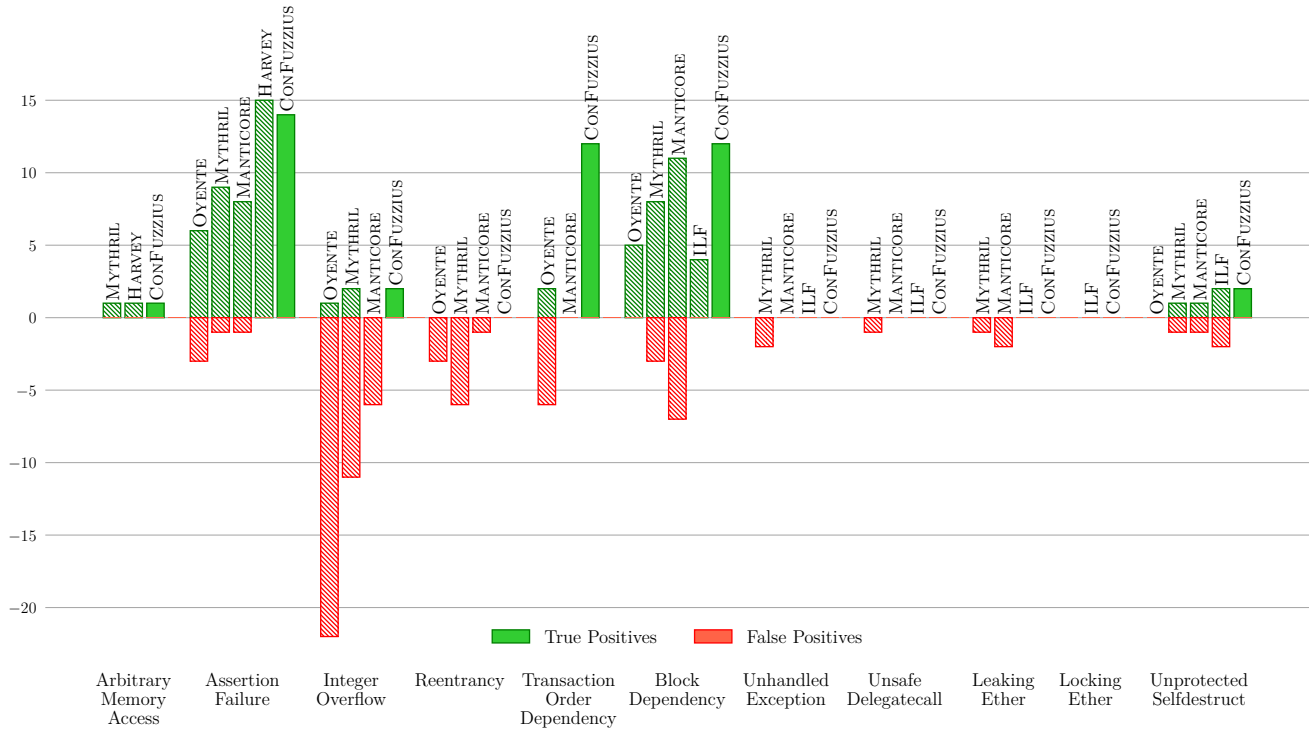

Fig. 7. Comparison of detected vulnerabilities between ConFuzzius and other security tools.

several other automated tools for assessing the security of smart contract were proposed. Luu et al. present Oyente [22], the first symbolic execution tool for Ethereum smart contracts. Oyente can automatically detect vulnerability patterns, such as transaction order dependency and reentrancy. Nikolic et al. present Maian [27], a tool that employs inter-procedural symbolic analysis as well as concrete validation to find and validate vulnerabilities on trace properties of Ethereum smart contracts. Mueller et al. present Mythril [26], a security analysis tool for Ethereum smart contracts. It uses concolic analysis, taint analysis and control-flow checking to detect a variety of security vulnerabilities. Torres et al. propose Osiris [36], a tool based on Oyente that focuses on detecting integer related bugs, such as integer overflows. It combines symbolic execution and taint analysis in order to reduce the number of false positives.

Smart Contract Static Analysis. Besides, symbolic execution, other works based on static analysis were proposed. Kalra et al. propose Zeus [18], a framework for automated verification of smart contracts using abstract interpretation and model checking, accepting user-provided policies. Zeus inserts policy predicates as assert statements in the source code, then translates the source code into an intermediate LLVM representation, and finally invokes its verifier to determine assertion violations. Tsankov et al. present Securify [37], a tool that uses static analysis based on a contract's dependency graph to extract semantic information about the program bytecode and then check for violations of safety patterns. To remain flexible, the tool permits new patterns to be specified via a designated domain-specific language. Finally, Kolluri et al. present EthRacer [20], a tool that, similar to our work, uses a hybrid approach. However, the authors employ the opposite of our strategy, by primarily using concolic 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 is more efficient than EthRacer's strategy, because it is not completely random but rather based on read-after-write data dependencies between transactions, yielding faster and more efficient combinations of transaction order dependencies.

VII. Conclusion

We presented ConFuzzius, a novel hybrid fuzzer that solves the three main challenges of smart contract testing. The key idea is to model not only transactions but also the execution environment, and combine the characteristics of evolutionary fuzzing with data dependency analysis and on-demand constraint solving to generate meaningful transaction sequences that get past complex conditions. We run ConFuzzius against a benchmark of real-world smart contracts [41], and showed that it detects significantly more vulnerabilities and achieves more code coverage than state-of-the-art fuzzers and symbolic execution tools for smart contracts. In future work, we plan to extend the evaluation to a large scale blockchain analysis and improve code coverage by dealing with transaction sequences that require consistent input values.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] ConsenSys Diligence, "EVM Analyzer Benchmark Suite," August 2019. [Online]. Available: https://github.com/ConsenSys/evm-analyzer-benchmark-suite

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

[9] 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.

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

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

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

[13] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 48, 2017.

[14] 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

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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

[23] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

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

[25] 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.

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

[27] 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.

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

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

[30] 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.

[31] D. G. Saari and V. R. Merlin, "The copeland method," *Economic Theory*, vol. 8, no. 1, pp. 51–76, 1996.

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

[33] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 9–19.

[34] 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.

[35] 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.

[36] 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

[37] 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.

[38] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[39] G. Wood, "Solidity 0.6.3 documentation," February 2020, https://solidity.readthedocs.io/en/v0.6.3/.

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

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

APPENDIX

A. Vulnerability Detectors

We elaborate on the implementation details of our 11 vulnerability detectors below.

Arbitrary Memory Access. We make use if our symbolic taint flow analysis to detect arbitrary memory access. We check if the execution trace contains an `SSTORE` instruction where its two arguments, storage index and storage value, are directly addressable via function arguments. This is achieved by checking if symbolic values originating from `CALLDATALOAD` flow into both arguments of `SSTORE`.

Assertion Failure. We detect an assertion failure by checking

TABLE IV
INSTRUCTION COVERAGE COMPARISON BETWEEN CONFUZZIUS AND ILF.

BID	ILF	CONFUZZIUS	Ratio	p-value	$A_{\text{ILF,CONFUZZIUS}}$	$A_{\text{CONFUZZIUS,ILF}}$	Total
1	**3154**	3085.0	0.80	<0.001	0.79	0.21	8899
2	4139.0	**4936.0**	1.23	<0.001	0.00	1.00	5212
3	3759.0	**4562.0**	1.31	<0.001	0.00	1.00	4767
4	4121.0	**5052.5**	1.34	<0.001	0.00	1.00	5335
5	3859.0	**4635.5**	1.32	<0.001	0.00	1.00	4890
6	1936.0	**2330.0**	1.20	<0.001	0.00	1.00	2631
7	1587.0	1587.0	1.04	<0.001	0.50	0.50	1658
8	2273.0	**2561.0**	1.16	<0.001	0.00	1.00	2934
9	2905.0	**3627.0**	1.05	<0.001	0.00	1.00	4109
10	5793.0	**6728.0**	0.91	<0.001	0.00	1.00	7659
11	7577.5	**8856.0**	1.02	<0.001	0.00	1.00	10493
12	3854.0	**4263.0**	0.83	<0.001	0.00	1.00	5237
13	2573.0	**3394.0**	0.75	<0.001	0.00	1.00	3720
14	**4292.0**	3763.0	0.81	<0.001	1.00	0.00	6735
15	**5278.0**	5086.5	1.00	<0.001	0.92	0.08	6208
16	498.0	**505.0**	1.02	<0.001	0.00	1.00	752
17	3062.0	**3072.0**	1.12	<0.001	0.17	0.83	3077
18	2368.0	**2419.0**	0.83	<0.001	0.00	1.00	2715
19	2608.0	**3325.0**	1.27	<0.001	0.00	1.00	6185
20	3043.0	**3157.0**	1.05	<0.001	0.00	1.00	3243
21	447.0	447.0	1.03	<0.001	0.50	0.50	448
22	1302.0	**1309.0**	1.03	<0.001	0.00	1.00	1317
23	2028.5	**2109.0**	1.01	<0.001	0.00	1.00	2379
24	693.0	**939.0**	1.25	<0.001	0.00	1.00	946
25	910.0	**1347.0**	1.13	<0.001	0.00	1.00	1351
26	4449.0	**4486.0**	2.79	<0.001	0.14	0.86	5034
27	6893.0	**7752.0**	1.41	<0.001	0.00	1.00	8492
Median	3163.0	**3531.0**	1.14				4312

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. 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 used to send funds. We follow the approach by Torres et al. [36] 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 ourselves. Afterwards, we check if our result is equivalent to the result that has been pushed onto the stack. If they are not equivalent, then 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.

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 2300 and where the amount of funds to be transferred depends on an SLOAD instruction. We then report a reentrancy if we find 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 the value 1 as a result after the call to the stack. 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 it does not flow 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 non-attacker 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 or has never been passed as a parameter in a function by a 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 and its sender is an attacker.

B. HARVEY *Benchmark*

TABLE V
OVERVIEW OF THE HARVEY BENCHMARK.

BIDs	Name	Func.	LoSC	Description
1	ENS	24	1205	ENS domain name auction
2-3	CMSW	49	503	ConsenSys multisig wallet
4-5	GMSW	49	704	Gnosis multisig wallet
6	BAT	23	191	BAT token (advertising)
7	CT	12	200	ConsenSys token library
8	ERCF	19	747	ERC Fund (investment fund)
9	FBT	34	385	FirstBlood token (e-sports)
10-13	HPN	173	3065	Havven payment network
14	MR	25	1053	MicroRaiden payment service
15	MT	38	437	MOD token (supply-chain)
16	PC	7	69	Payment channel
17-18	RNTS	49	749	Request Network token sale
19	DAO	23	783	The DAO organization
20	VT	18	242	Valid token (personal data)
21	USCC1	4	57	USCC17 entry
22	USCC2	14	89	USCC17 (honorable mention)
23	USCC3	21	535	USCC17 (3rd place)
24	USCC4	7	164	USCC17 (1st place)
25	USCC5	10	188	USCC17 (2nd place)
26	PW	19	549	Parity multisig wallet
27	BNK	44	649	Bankera token
Total		**662**	**12564**	

C. Copeland Method

TABLE VI
COPELAND WINNERS FOR EACH PAIRWISE COMPARISON.

Comparison			Result			Winner
OYENTE	vs.	MYTHRIL	7	vs.	20	MYTHRIL
OYENTE	vs.	MANTICORE	17	vs.	10	OYENTE
OYENTE	vs.	CONFUZZIUS	2	vs.	25	CONFUZZIUS
MYTHRIL	vs.	MANTICORE	22	vs.	1	MYTHRIL
MYTHRIL	vs.	CONFUZZIUS	13	vs.	14	CONFUZZIUS
MANTICORE	vs.	CONFUZZIUS	2	vs.	25	CONFUZZIUS

D. Copeland Ranking

TABLE VII
RANKING BETWEEN CONFUZZIUS AND SYMBOLIC EXECUTORS.

Toolname	Victories	Defeats	Result	Ranking
OYENTE	1	2	-1	3rd
MYTHRIL	2	1	1	2nd
MANTICORE	0	3	-3	4th
CONFUZZIUS	**3**	**0**	**3**	**1st**

E. Number of Transactions per Vulnerability

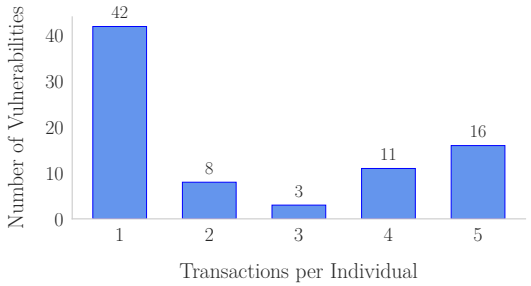

Fig. 8.

F. Instruction Coverage Between CONFUZZIUS *and* ILF