HARVEY: A Greybox Fuzzer for Smart Contracts

Valentin Wüstholz
ConsenSys, Germany
valentin.wustholz@consensys.net

Maria Christakis
MPI-SWS, Germany
maria@mpi-sws.org

ABSTRACT

We present HARVEY, an industrial greybox fuzzer for smart contracts, which are programs managing accounts on a blockchain.

Greybox fuzzing is a lightweight test-generation approach that effectively detects bugs and security vulnerabilities. However, greybox fuzzers randomly mutate program inputs to exercise new paths; this makes it challenging to cover code that is guarded by narrow checks. Moreover, most real-world smart contracts transition through many different states during their lifetime, e.g., for every bid in an auction. To explore these states and thereby detect deep vulnerabilities, a greybox fuzzer would need to generate sequences of contract transactions, e.g., by creating bids from multiple users, while keeping the search space and test suite tractable.

In this paper, we explain how HARVEY alleviates both challenges with two key techniques. First, HARVEY extends standard greybox fuzzing with a method for predicting new inputs that are more likely to cover new paths or reveal vulnerabilities in smart contracts. Second, it fuzzes transaction sequences in a targeted and demand-driven way. We have evaluated our approach on 27 real-world contracts. Our experiments show that our techniques significantly increase HARVEY's effectiveness in achieving high coverage and detecting vulnerabilities, in most cases orders-of-magnitude faster.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

automated testing, greybox fuzzing, smart contracts

ACM Reference Format:
Valentin Wüstholz and Maria Christakis. 2020. HARVEY: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3417064

1 INTRODUCTION

Smart contracts are programs that manage crypto-currency accounts on a blockchain. Reliability of these programs is critical since bugs may jeopardize digital assets. Automatic test generation has shown to be effective in finding many types of bugs, thereby improving software quality. In fact, there exists a wide variety of test-generation tools, ranging from random testing [29, 30, 59], over greybox fuzzing [5, 9], to dynamic symbolic execution [20, 37].

Random testing [29, 30, 59] and blackbox fuzzing [7, 11] generate random inputs to a program, run the program with these inputs, and check for bugs. Despite the practicality of these techniques, their effectiveness, that is, their ability to explore new paths, is limited. The search space of valid program inputs is typically huge, and a random exploration can only exercise a small fraction of (mostly shallow) paths.

At the other end of the spectrum, dynamic symbolic execution [20, 37] and whitebox fuzzing [19, 34, 38] repeatedly run a program, both concretely and symbolically. At runtime, they collect symbolic constraints on program inputs from branch statements along the execution path. These constraints are then appropriately modified and a constraint solver is used to generate new inputs, thereby steering execution toward another path. Although these techniques are very effective in covering new paths, they cannot be as efficient and scalable as other test-generation techniques that do not spend any time on program analysis and constraint solving. Moreover, dynamic symbolic execution struggles with functionality commonly found in smart contracts, such as hash functions. Even more importantly, it struggles with input-dependent loops, which are present in any smart contract due to the possibly unbounded number of transactions that may invoke it. This is why existing symbolic-execution engines targeting smart contracts bound the number of explored transactions [6, 57].

Greybox fuzzing [5, 9] lies in the middle of the spectrum between performance and effectiveness in discovering new paths. It does not require program analysis or constraint solving, but it relies on a lightweight program instrumentation that allows the fuzzer to tell when an input exercises a new path. In other words, the instrumentation is useful in computing a unique identifier for each explored path in the program under test. American Fuzzy Lop (AFL) [9] is a prominent example of a state-of-the-art greybox fuzzer that has detected numerous bugs and security vulnerabilities [1].

In this paper, we present HARVEY, an industrial greybox fuzzer for smart contracts. HARVEY has been under development since Sept. 2017. It is used at ConsenSys[1] both for smart-contract audits and as part of MythX, an automated contract analysis service[2]. It has analyzed more than 3.3M submitted contracts from March 2019 to April 2020 and has found hundreds of thousands of issues.

Here, we describe HARVEY's design and architecture and focus on how to alleviate two key challenges we encountered when fuzzing real-world contracts. Although the challenges are not exclusive to our specific application domain, our techniques are shown to be particularly effective for smart contracts, thereby providing useful insights for other contract analyzers.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417064

[1] https://consensys.net
[2] https://mythx.io

Challenge #1. Despite the fact that greybox fuzzing strikes a good balance between performance and effectiveness, the inputs are still randomly mutated, for instance, by flipping arbitrary bits. As a result, many generated inputs exercise the same program paths. To address this problem, there have emerged techniques that direct greybox fuzzing toward low-frequency paths [17], vulnerable paths [64], deep paths [70], or specific sets of program locations [16]. Such techniques have mostly focused on which seed inputs to prioritize and which parts of these inputs to mutate.

Challenge #2. Smart contracts may transition through many different states during their lifetime, for instance, for every bet in a gambling game. The same holds for any stateful system that is invoked repeatedly, such as a web service. Therefore, detecting vulnerabilities in such programs often requires generating and fuzzing sequences of invocations that explore the possible states. For instance, to test a smart contract that implements a gambling game, a fuzzer would need to automatically create sequences of bets from multiple players. However, since the number of possible sequences grows exponentially with the sequence length, it is difficult to efficiently detect the few sequences that reveal a bug.

Our approach and general insights. To alleviate the first challenge, we developed a technique that systematically predicts new inputs for the program under test with the goal of increasing performance and effectiveness of greybox fuzzing. In contrast to existing work in greybox fuzzing, our approach suggests concrete input values based on information from previous executions, instead of performing arbitrary mutations.

Inputs are predicted in a way that aims to direct greybox fuzzing toward *optimal* executions, for instance, defined as executions that flip a branch condition in order to increase coverage. Our technique is parametric in what constitutes an optimal execution, and in particular, in what properties such an execution needs to satisfy.

More specifically, each program execution is associated with zero or more *cost metrics*, which are computed automatically. A cost metric captures how close the execution is to satisfying a given property at a given program location. Executions that minimize a cost metric are considered optimal with respect to that metric. For example, a cost metric could be defined at each arithmetic operation in the program such that it is minimized (i.e., becomes zero) when an execution triggers an arithmetic overflow. Our technique uses the costs that are computed with cost metrics along executions of the program to *iteratively* predict inputs leading to optimal executions. Our experiments show that Harvey is extremely successful in predicting inputs that flip a branch condition even in a single iteration (average success rate of 99%).

Although this input-prediction technique is very effective in practice, it is not sufficient for thoroughly testing a smart contract and its state space. As a result, Harvey generates, executes, and fuzzes sequences of transactions, which invoke the contract's functions. Each of these transactions can have side effects on the contract's state, which may affect the execution of subsequent invocations. To alleviate the second challenge of exploring the search space of all possible sequences, we devised a technique for demand-driven sequence fuzzing, which avoids generating transaction sequences when they cannot further increase coverage. Our experiments show that 80% of bugs in real smart contracts require generating more than one transaction to be found. This highlights the need for techniques like ours that are able to effectively prune the space of transaction sequences.

We evaluate Harvey on 27 benchmarks from related work [84] and a large industrial benchmark. The presented techniques boost its effectiveness in achieving high coverage (by up to 3x) and detecting vulnerabilities, in most cases orders-of-magnitude faster.

Contributions. We make the following contributions:

- We present Harvey, a greybox fuzzer for smart contracts that is being used industrially.
- We describe our architecture and two key techniques for alleviating the important challenges outlined above.
- We evaluate our fuzzer on real-world benchmarks and demonstrate the effectiveness of the underlying techniques.

2 BACKGROUND

In this section, we give background on standard greybox fuzzing and smart contracts.

2.1 Greybox Fuzzing

Alg. 1 shows how greybox fuzzing works. (The grey boxes should be ignored for now.) The fuzzer takes as input the program under test *prog* and a set of seeds *S*. It starts by running the program with the seeds, and during each program execution, the instrumentation is able to capture the path that is currently being explored and associate it with a unique identifier *PID* (line 1). Note that the *PIDs* data structure is a key-value store from a *PID* to an input that exercises the path associated with *PID*. Next, an input is selected for mutation (line 3), and it is assigned an "energy" value that denotes how many times it should be fuzzed (line 5).

The input is mutated (line 12), and the program is run with the new input (line 13). If the program follows a path that has not been previously explored, the new input is added to the test suite (lines 14–15). The above process is repeated until an exploration bound is reached (line 2). The fuzzer returns a test suite containing one test for each explored path.

2.2 Smart Contracts

Ethereum [3, 4] is one of the most popular blockchain-based [63, 72, 73], distributed-computing platforms [14]. It supports two kinds of accounts, user and contract accounts, both of which store a balance and publicly reside on the blockchain.

In contrast to a user account, a contract account is managed through code that is associated with it. The contract code captures agreements between users and other contracts, for example, to encode the rules of an auction. A contract account also has persistent state where the code may store data, such as auction bids.

Contract accounts, their code, and persistent state are called *smart contracts*. Programmers may write the code in several languages, like Solidity or Vyper, all of which compile to the Ethereum Virtual Machine (EVM) [83] bytecode.

To interact with a contract, users issue *transactions* that call its functions, for instance, to bid in an auction, and are required to pay a fee for transactions to be executed. This fee is called *gas* and is roughly proportional to how much code is run.

```
1  function baz(int256 a, int256 b, int256 c)
2           returns (int256) {
3    int256 d = b + c;
4    minimize(d < 1 ? 1 - d : 0);
5    minimize(d < 1 ? 0 : d);
6    if (d < 1) {
7      minimize(b < 3 ? 3 - b : 0);
8      minimize(b < 3 ? 0 : b - 2);
9      if (b < 3) {
10       return 1;
11     }
12     minimize(a == 42 ? 1 : 0);
13     minimize(a == 42 ? 0 : |a - 42|);
14     if (a == 42) {
15       return 2;
16     }
17     return 3;
18   } else {
19     minimize(c < 42 ? 42 - c : 0);
20     minimize(c < 42 ? 0 : c - 41);
21     if (c < 42) {
22       return 4;
23     }
24     return 5;
25   }
26 }
```

Figure 1: Example for fuzzing with input prediction.

3 OVERVIEW

We now give an overview of our approach focusing on the challenges we aim to alleviate.

3.1 Challenge #1: Random Input Mutations

Fig. 1 shows a constructed smart-contract function `baz` (in Solidity) that takes as input three (256-bit) integers `a`, `b`, and `c` and returns an integer. There are five paths in this function, all of which are feasible. Each path is denoted by a unique return value. (The grey boxes should be ignored for now.)

When running AFL, a state-of-the-art greybox fuzzer, on (a C version of) function `baz`, only four out of five paths are explored within 12h. During this time, greybox fuzzing constructs a test suite of four inputs, each of which exploring a different path. The path with return value 2 remains unexplored even after the fuzzer generates about 311M different inputs. All but four of these inputs are discarded as they exercise a path in `baz` that has already been covered by a previous test.

The path with return value 2 is not covered because greybox fuzzers randomly mutate program inputs (line 12 of Alg. 1). It is generally challenging for fuzzers to generate inputs that satisfy "narrow checks", that is, checks that only become true for very few input values (e.g., line 14 of Fig. 1). In this case, the probability that the fuzzer will generate value 42 for input `a` is 1 out of 2^{256} for 256-bit integers. Even worse, to cover the path with return value 2 (line 15), the sum of inputs `b` and `c` also needs to be less than 1 (line 6) and `b` must be greater than or equal to 3 (line 9). As a result, several techniques have been proposed to guide greybox fuzzing to satisfy such narrow checks, e.g., selective whitebox fuzzing [71].

Fuzzing with input prediction. In contrast, our technique for input prediction does not require any analysis or constraint solving. It does, however, require additional instrumentation of the program to collect more information about its structure than standard greybox fuzzing, thus making fuzzing a lighter shade of grey. This information captures the distance from an optimal execution at various program points and is then used to predict inputs that guide exploration toward optimal executions.

Our fuzzer takes as input a program *prog* and seeds S. It also requires a partial function f_{cost} that maps execution states to cost metrics. When execution of *prog* reaches a state s, the fuzzer evaluates the cost metric $f_{cost}(s)$. For example, the grey boxes in Fig. 1 define a function f_{cost} for `baz`. Each `minimize` statement specifies a cost metric at the execution state where it is evaluated. Note that f_{cost} constitutes a runtime instrumentation of *prog*—we use `minimize` statements only for illustration. A compile-time instrumentation would increase gas usage of the contract and potentially lead to false positives when detecting out-of-gas errors. On the contrary, a runtime instrumentation is implemented in the EVM, and thus, it does not affect contract semantics.

The cost metrics of Fig. 1 define optimal executions as those that flip a branch condition and are inspired by Korel's branch distance [49]. Specifically, consider an execution along which variable `d` evaluates to 0. This execution takes the then-branch of the first if-statement, and the cost metric defined by the `minimize` statement on line 4 evaluates to 1. This means that the distance of the current execution from an execution that exercises the (implicit) else-branch of the if-statement is 1. Now, consider a second execution that also takes this then-branch (`d` evaluates to −1). In this case, the cost metric on line 4 evaluates to 2, which indicates a greater distance from an execution that exercises the else-branch.

Based on this information, our input-prediction technique is able to suggest new inputs that make the execution of `baz` take the else-branch of the first if-statement and minimize the cost metric on line 4 (i.e., the cost becomes zero). For instance, assume that the predicted inputs cause `d` to evaluate to 7. Although the cost metric on line 4 is now minimized, the cost metric on line 5 evaluates to 7, which is the distance of the current execution from an execution that takes the then-branch.

Similarly, the `minimize` statements on lines 7–8, 12–13, and 19–20 of Fig. 1 define cost metrics that are minimized when an execution flips a branch condition in a subsequent if-statement. This instrumentation aims to maximize path coverage, and for this reason, an execution can never minimize all cost metrics. In fact, the fuzzer has achieved full path coverage when the generated tests cover all feasible combinations of branches in the program; that is, when they minimize all possible combinations of cost metrics.

The fuzzer does not exclusively rely on prediction to generate program inputs, for instance, when there are not enough executions from which to make a good prediction. In the above example, the inputs for the first two executions (where `d` is 0 and −1) are generated by the fuzzer without prediction. Prediction can only approximate correlations between inputs and their corresponding costs; therefore, it is possible that certain predicted inputs do not lead to optimal executions. In such cases, it is also up to standard fuzzing to generate inputs that cover any remaining paths.

For the example of Fig. 1, HARVEY explores all five paths within 8s and after generating only 15'545 different inputs.

```
1  contract Foo {
2    int256 private x;
3    int256 private y;
4
5    constructor () public {
6      x = 0;
7      y = 0;
8    }
9
10   function Bar() public returns (int256) {
11     if (x == 42) {
12       assert(false);
13       return 1;
14     }
15     return 0;
16   }
17
18   function SetY(int256 ny) public { y = ny; }
19
20   function IncX() public { x++; }
21
22   function CopyY() public { x = y; }
23 }
```

Figure 2: Example for demand-driven sequence fuzzing.

3.2 Challenge #2: State Space Exploration

Fig. 2 shows a simple contract `Foo`. The constructor on line 5 initializes variables `x` and `y`, which are stored in the persistent state of the contract. In function `Bar`, the failing assertion (line 12) denotes a bug. An assertion violation causes a transaction to be aborted, and as a result, users lose their gas. Triggering the bug requires a sequence of at least three transactions, invoking functions `SetY(42)`, `CopyY()`, and `Bar()`. (Note that a transaction may directly invoke up to one function.) The assertion violation may also be triggered by calling `IncX` 42 times before invoking `Bar`.

There are three ways to test this contract with a standard greybox fuzzer. First, each function could be fuzzed separately without considering the persistent variables of the contract as fuzzable inputs. For example, `Bar` would be executed only once—it has zero fuzzable inputs. No matter the initial values of `x` and `y`, the fuzzer would only explore one path in `Bar`.

Second, each function could be fuzzed separately while considering the persistent variables as fuzzable inputs. The fuzzer would then try to explore both paths in `Bar` by generating values for `x` and `y`. However, the probability of generating value 42 for `x` is tiny, as discussed earlier. More importantly, this approach might result in false positives when the persistent state generated by the fuzzer is not reachable with any sequence of transactions. For example, the contract would never fail if `SetY` ensured that `y` is never set to 42 and `IncX` only incremented `x` up to 41.

Third, the fuzzer could try to explore all paths in all possible sequences of transactions up to a bounded length. This, however, means that a path would span all transactions (instead of a single function). For example, a transaction invoking `Bar` and a sequence of two transactions invoking `CopyY` and `Bar` would exercise two different paths in the contract, even though from the perspective of `Bar` this is not the case. With this approach, the number of possible sequences grows exponentially in their length, and so does the number of tests in the test suite. The larger the test suite, the more difficult it becomes to find a test that, when fuzzed, leads to the assertion in `Foo`, especially within a certain time limit.

We propose a technique for demand-driven sequence fuzzing that alleviates these limitations. First, it discovers that the only branch in `Foo` that requires more than a single transaction to be covered is the one leading to the assertion in `Bar`. Consequently, Harvey only generates transaction sequences whose last transaction invokes `Bar`. Second, our technique aims to increase path coverage only of the function that is invoked by this last transaction. In other words, the goal of any previous transactions is to set up the state, and path identifiers are computed only for the last transaction. Therefore, reaching the assertion in `Bar` by first calling `SetY(42)` and `CopyY()` or by invoking `IncX()` 42 times both result in covering the same path of the contract.

Harvey triggers the above assertion violation in about 29s and after generating 48'117 inputs.

4 FUZZING WITH INPUT PREDICTION

In this section, we present the technical details of how we extend greybox fuzzing with input prediction.

4.1 Algorithm

The grey boxes in Alg. 1 indicate the key differences. In addition to the program under test *prog* and a set of seeds S, Alg. 1 takes as input a partial function f_{cost} that, as explained earlier, maps execution states to cost metrics. The fuzzer first runs the program with the seeds, and during each program execution, it evaluates the cost metric $f_{cost}(s)$ for every encountered execution state s in the domain of f_{cost} (line 1). Like in standard greybox fuzzing, each explored path is associated with a unique identifier *PID*. Note, however, that the *PIDs* data structure now maps a *PID* both to an input that exercises the corresponding path as well as to a cost vector, which records all costs computed during execution of the program with this input. Next, an input is selected for mutation (line 3) and assigned an energy value (line 5).

The input is mutated (line 12), and the program is run with the new input (line 13). We assume that the new input differs from the original input (which was selected for mutation on line 3) by the value of a single input parameter—an assumption that typically holds for mutation-based greybox fuzzers. As usual, if the program follows a path that has not been explored, the new input is added to the test suite (lines 14–15).

On line 17, the original and the new input are passed to the prediction component of the fuzzer along with their cost vectors. This component inspects *input* and *input′* to determine the input parameter by which they differ. Based on the cost vectors, it then suggests a new value for this input parameter such that one of the cost metrics is minimized. In case a new input is predicted, the program is tested with this input, otherwise the original input is mutated (lines 8–13). The former happens even if the energy of the original input has run out (line 7) to ensure that we do not waste predicted inputs.

The above process is repeated until an exploration bound is reached (line 2), and the fuzzer returns a test suite containing one test for each program path that has been explored.

Algorithm 1: Greybox fuzzing with input prediction.

Input: Program *prog*, Seeds *S*, Cost function f_{cost}

```
1  PIDs ← RunSeeds(S, prog, f_cost)
2  while ¬Interrupted() do
3    input, cost ← PickInput(PIDs)
4    energy ← 0
5    maxEnergy ← AssignEnergy(input)
6    predictedInput ← nil
7    while energy < maxEnergy ∨ predictedInput ≠ nil do
8      if predictedInput ≠ nil then
9        input′ ← predictedInput
10       predictedInput ← nil
11     else
12       input′ ← FuzzInput(input)
13     PID′, cost′ ← Run(input′, prog, f_cost)
14     if IsNew(PID′, PIDs) then
15       PIDs ← Add(PID′, input′, cost′, PIDs)
16     if energy < maxEnergy then
17       predictedInput ← Predict(input, cost, input′, cost′)
18     energy ← energy + 1
```

Output: Test suite Inputs(*PIDs*)

Example. In Tab. 1, we run our algorithm on the example of Fig. 1 step by step. The first column of the table shows an identifier for every generated test, and the second column shows the path that each test exercises identified by the return value of the program. The highlighted boxes in this column denote paths that are covered for the first time, which means that the corresponding tests are added to the test suite (lines 14–15 of Alg. 1). The third column shows the test identifier from which the value of variable *input* is selected (line 3 of Alg. 1). Note that, according to the algorithm, *input* is selected from tests in the test suite.

The fourth column shows a new input for the program under test; this input is either a seed or the value of variable *input′* in the algorithm, which is obtained with input prediction (line 9) or fuzzing (line 12). Each highlighted box in this column denotes a predicted value. The fifth column shows the cost vector that is computed when running the program with the new input of the fourth column. Note that we show only non-zero costs and that the subscript of each cost denotes the line number of the corresponding `minimize` statement in Fig. 1. The sixth column shows which costs (if any) are used to predict a new input, and the last column shows the current energy value of the algorithm's *input* (lines 4 and 18). For simplicity, we consider *maxEnergy* of Alg. 1 (line 5) to always have value 2 in this example. Our implementation, however, incorporates an existing energy schedule [17].

We assume that the seeds *S* contain only the random input (a = −1, b = 0, c = −5) (test #1 in Tab. 1). This input is then fuzzed to produce (a = −1, b = −3, c = −5) (test #2), that is, to produce a new value for input parameter b. Our algorithm randomly selects the costs computed with metric C_7 to predict a new value for b. (We explain how new values are predicted in the next subsection.) As a result, test #3 exercises a new path of the program (the one with return value 3). From the cost vectors of tests #1 and #3, only the costs computed with metric C_4 may be used to predict another value for b; costs C_7 and C_8 are already zero in one of the two tests, while metric C_{13} is not reached in test #1. Even though the energy of the original input (from test #1) has run out, the algorithm still runs the program with the input predicted from the C_4 costs (line 7). This results in covering the path with return value 4.

Table 1: Running Alg. 1 on the example of Fig. 1.

Test	Path	Test Input	New Input a	b	c	Costs	Pred. Cost	Energy
1	1	–	−1	0	−5	$C_4 = 6$ $C_7 = 3$	–	–
2	1	1	−1	−3	−5	$C_4 = 9$ $C_7 = 6$	C_7	0
3	3	1	−1	3	−5	$C_4 = 3$ $C_8 = 1$ $C_{13} = 43$	C_4	1
4	4	1	−1	6	−5	$C_5 = 1$ $C_{19} = 47$	–	2
5	3	3	7	3	−5	$C_4 = 3$ $C_8 = 1$ $C_{13} = 35$	C_{13}	0
6	2	3	42	3	−5	$C_4 = 3$ $C_8 = 1$ $C_{12} = 1$	–	1
7	4	4	−1	6	0	$C_5 = 6$ $C_{19} = 42$	C_{19}	0
8	5	4	−1	6	42	$C_5 = 48$ $C_{20} = 1$	–	1

Next, we select an input from tests #1, #3, or #4 of the test suite. Let's assume that the fuzzer picks the input from test #3 and mutates the value of input parameter a. Note that the cost vectors of tests #3 and #5 differ only with respect to the C_{13} costs, which are therefore used to predict a new input for a. The new input exercises a new path of the program (the one with return value 2). At this point, the cost vectors of tests #3 and #6 cannot be used for prediction because the costs are either the same (C_4 and C_8) or they are already zero in one of the two tests (C_{12} and C_{13}). Since no input is predicted and the energy of the original input (from test #3) has run out, our algorithm selects another input from the test suite.

This time, let's assume that the fuzzer picks the input from test #4 and mutates the value of input parameter c. From the cost vectors of tests #4 and #7, it randomly selects the C_{19} costs for predicting a new value for c. The predicted input exercises the fifth path of the program, thus achieving full path coverage of function baz by generating only 8 tests.

Note that our algorithm makes several non-systematic choices, which may be random or based on heuristics, such as when function PickInput picks an input from the test suite, when FuzzInput selects which input parameter to fuzz, or when Predict decides which costs to use for prediction. For illustrating how the algorithm works, we made "good" choices such that all paths are exercised with a small number of tests. In practice, the fuzzer achieved full path coverage of baz with 15'545 tests, instead of 8 (see Sect. 3.1).

4.2 Input Prediction

Our algorithm passes to the prediction component the input vectors *input* and *input′* and the corresponding cost vectors *cost* and *cost′* (line 17 of Alg. 1). The input vectors differ by the value of a single input parameter, say i_0 and i_1. Now, let us assume that the prediction component selects a cost metric to minimize and that the costs that

have been evaluated using this metric appear as c_0 and c_1 in the cost vectors, i.e., c_0 is associated with i_0, and c_1 with i_1.

For example, consider tests #3 and #5 in Tab. 1. The input vectors differ by the value of input parameter `a`, so $i_0 = -1$ (value of `a` in test #3) and $i_1 = 7$ (value of `a` in test #5). The prediction component chooses to make a prediction based on C_{13} since the cost vectors of tests #3 and #5 differ only with respect to this cost metric, so $c_0 = 43$ (value of C_{13} in test #3) and $c_1 = 35$ (value of C_{13} in test #5).

Using the two data points (i_0, c_0) and (i_1, c_1), the goal is to find a value i such that the corresponding cost is zero. In other words, our technique aims to find a root of the unknown, but computable, function that relates input parameter `a` to cost metric C_{13}. While there is a wide range of root-finding algorithms, Harvey uses the *Secant method*. Like other methods, such as Newton's, the Secant method tries to find a root by performing successive approximations.

Its basic approximation step considers the two data points as x-y-coordinates on a plane. Our technique then fits a straight line $c(i) = m * i + k$ through the points, where m is the slope of the line and k is a constant. To predict the new input value, it determines the x-coordinate i where the line intersects with the x-axis (i.e., where the cost is zero).

From the points $(-1, 43)$ and $(7, 35)$ defined by tests #3 and #5, we compute the line to be $c(i) = -i + 42$. Now, for the cost to be zero, the value of parameter `a` must be 42. Indeed, when `a` becomes 42 in test #6, cost metric C_{13} is minimized.

This basic approximation step is precise if the target cost metric is indeed linear (or piece-wise linear) with respect to the input parameter for which the prediction is made. If not, the approximation may fail to minimize the cost metric. In such cases, Harvey applies the basic step iteratively (as the Secant method). Our experiments show that one iteration is typically sufficient in practice.

4.3 Cost Metrics

We now describe the different cost metrics that our fuzzer aims to minimize: (1) ones that are minimized when execution flips a branch condition, and (2) ones that are minimized when execution is able to modify arbitrary memory locations.

Branch conditions. We have already discussed cost metrics that are minimized when execution flips a branch condition in the example of Fig. 1. Here, we show how the cost metrics are automatically derived from the program under test.

For the comparison operators == (*eq*), < (*lt*), and <= (*le*), we define the following cost functions:

$$C_{eq}(l,r) = \begin{cases} 1, & l = r \\ 0, & l \neq r \end{cases} \qquad C_{\overline{eq}}(l,r) = \begin{cases} 0, & l = r \\ |l - r|, & l \neq r \end{cases}$$

$$C_{lt}(l,r) = \begin{cases} r - l, & l < r \\ 0, & l \geq r \end{cases} \qquad C_{\overline{lt}}(l,r) = \begin{cases} 0, & l < r \\ l - r + 1, & l \geq r \end{cases}$$

$$C_{le}(l,r) = \begin{cases} r - l + 1, & l \leq r \\ 0, & l > r \end{cases} \qquad C_{\overline{le}}(l,r) = \begin{cases} 0, & l \leq r \\ l - r, & l > r \end{cases}$$

C_{eq} is non-zero when a branch condition l == r holds; it defines the cost metric for making this condition false. On the other hand, $C_{\overline{eq}}$ defines the cost metric for making the same branch condition true. The arguments l and r denote the left and right operands of the operator. The notation is similar for all other functions.

Based on these cost functions, our instrumentation evaluates two cost metrics before every branch condition in the program under test. The metrics depend on the comparison operator used in the branch condition. The cost functions for other comparison operators, i.e., != (*ne*), > (*gt*), and >= (*ge*), are easily derived from the functions above, and our tool supports them. Note that our implementation works on the bytecode, where logical operators, like `&&` or `||`, are expressed as branch conditions. We, thus, do not define cost functions for such operators, but they are also straightforward.

```
1  contract Wallet {
2    address private owner;
3    uint[] private bonusCodes;
4
5    ...
6
7    function PopCode() public {
8      require(0 <= bonusCodes.length);
9      bonusCodes.length--;
10   }
11
12   function SetCodeAt(uint idx, uint c) public {
13     require(idx < bonusCodes.length);
14     minimize(|&(bonusCodes[idx]) - 0xffcaffee|);
15     bonusCodes[idx] = c;
16   }
17
18   function Destroy() public {
19     require(msg.sender == owner);
20     selfdestruct(msg.sender);
21   }
22 }
```

Figure 3: Example of a memory-access vulnerability.

Observe that the inputs of the above cost functions are the operands of comparison operators, and not program inputs. This makes the cost functions *precise*, that is, when a cost is minimized, the corresponding branch is definitely flipped. Approximation can only be introduced when computing the correlation between a program input and a cost (Sect. 4.2).

Memory accesses. To illustrate the flexibility of our cost metrics, we now show another instantiation that targets a vulnerability specific to smart contracts. Consider the example in Fig. 3. (The grey box should be ignored for now.) It is a simplified version of code submitted to the Underhanded Solidity Coding Contest (USCC) in 2017 [10]. The USCC is a contest to write seemingly harmless Solidity code that, however, disguises unexpected vulnerabilities.

Fig. 3 implements a wallet that has an owner and stores an array of bonus codes (lines 2–3). Contract functions (not all shown) allow bonus codes to be pushed, popped, or updated. The last function (line 18) must be called only by the owner and causes the wallet to self-destruct after transferring all assets to the owner.

The vulnerability in this code is caused by the precondition on line 8, which should require the array length to be greater than zero (not equal) before popping an element. When the array is empty, the statement on line 9 causes the (unsigned) array length to underflow; this effectively disables the bound-checks of the array, allowing elements to be stored anywhere in the persistent state of the contract. Therefore, by setting a bonus code at a specific index in the array, an attacker could overwrite the address of the owner to their own address. Then, by destroying the wallet, the attacker would transfer all assets to their account. In a more optimistic

scenario, the owner could be accidentally set to an invalid address, in which case the wallet assets would become inaccessible.

To detect such vulnerabilities, a greybox fuzzer can, for every assignment to the persistent state of a contract, pick an arbitrary address and compare it to the target address of the assignment. When these two addresses happen to be the same, it is very likely that the assignment may also target other arbitrary addresses, perhaps as a result of an exploit. A fuzzer without input prediction, however, is only able to detect these vulnerabilities by chance, and chances are extremely low that the target address of an assignment matches an arbitrarily selected address, especially given that these are 32 bytes long. In fact, when disabling Harvey's input prediction, the vulnerability in the code of Fig. 3 is not detected within 12h.

To direct the fuzzer toward executions that could reveal such vulnerabilities, we define the following cost function:

$$C_{st}(lhsAddr, addr) = |lhsAddr - addr|$$

Here, *lhsAddr* denotes the address of the left-hand side of an assignment to persistent state (that is, excluding assignments to local variables) and *addr* an arbitrary address. C_{st} is non-zero when *lhsAddr* and *addr* are different, and therefore, optimal executions are those where the assignment writes to the arbitrary address.

Our instrumentation evaluates the corresponding cost metric before every assignment to persistent state. An example is shown on line 14 of Fig. 3. (We use the `&` operator to denote the address of `bonusCodes[idx]`.) Harvey with input prediction detects the vulnerability in Fig. 3 within 25s and after generating 43'950 inputs.

Detecting such vulnerabilities based on whether an assignment could target an arbitrary address might generate false positives when the address is indeed an intended target of the assignment. However, the probability of this occurring in practice is extremely low (again due to the address length). So far, we have not encountered any false positives.

In general, defining other cost functions is straightforward as long as there is an expressible measure for the distance between a current execution and an optimal one.

5 DEMAND-DRIVEN SEQUENCE FUZZING

Recall from Sect. 3.2 that Harvey uses demand-driven sequence fuzzing to set up the persistent state for testing the last transaction in the sequence. The goal is to explore new paths in the function that this transaction invokes, and thus, detect more bugs. Directly fuzzing the state, for instance, variables `x` and `y` of Fig. 2, might lead to false positives. Nonetheless, Harvey uses this aggressive approach when fuzzing transaction sequences to determine whether a different persistent state can increase path coverage.

The key idea is to generate longer transaction sequences on demand. This is achieved by fuzzing a transaction sequence in two modes: *regular*, which does not directly fuzz the persistent state, and *aggressive*, which is enabled with a small probability p_a (0.125 in our implementation) and may fuzz the persistent state directly. If Harvey is able to increase coverage of the last transaction in the sequence using the aggressive mode, the corresponding input is discarded (because it might lead to false positives), but longer sequences are generated when running in regular mode later.

For instance, when fuzzing a transaction that invokes `Bar` from Fig. 2, Harvey temporarily considers `x` and `y` as fuzzable inputs of the function. If this aggressive fuzzing does not discover any more paths, Harvey does not generate additional transactions before the invocation of `Bar`. If, however, the aggressive fuzzing discovers new paths, our tool generates and fuzzes transaction sequences whose last transaction calls `Bar`. That is, longer transaction sequences are only generated when they might be able to set up the state before the last transaction such that its coverage is increased.

For our example, Harvey generates the sequence `SetY(42)`, `CopyY()`, and `Bar()` that reaches the assertion in about 29s. At this point, the fuzzer stops exploring longer sequences for contract `Foo` because aggressively fuzzing the state cannot further increase the already achieved coverage.

We make two important observations. First, Harvey is so quick in finding the right argument for `SetY` due to input prediction. Second, demand-driven sequence fuzzing relies on path identifiers to span no more than a single transaction. Otherwise, aggressive fuzzing would not be able to determine if longer sequences may increase coverage of the contract.

Mutation operations. To generate and fuzz sequences of transactions, Harvey applies three mutation operations to a given transaction t: (1) fuzz transaction t, which fuzzes the inputs of its invocation, (2) insert a new transaction before t, and (3) replace the transactions before t with another sequence.

Harvey uses two pools for efficiently generating new transactions or sequences, respectively. These pools store transactions or sequences that are found both to increase coverage of the contract under test and to modify the persistent state in a way that has not been explored before. Harvey selects new transactions or sequences from these pools when applying the second and third mutation operations.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate Harvey on real-world smart contracts.

6.1 Benchmark Selection

Merely fuzzing millions of contracts provides little insight about Harvey's effectiveness. Instead, we perform a thorough evaluation on 27 benchmarks from related work [84] and a large benchmark from a client engagement (see RQ4). The corresponding paper [84] provides an overview of the 27 benchmarks, the projects from which they originate, and outlines how the benchmarks were selected.

Generally, the benchmarks were selected by following published guidelines on evaluating fuzzers [48]. The authors did not simply scrape contracts from the blockchain since most are created with no quality control and many contain duplicates—contracts without assets or users are essentially dead code. Moreover, good-quality contracts typically have dependencies (e.g., on libraries or other contracts) that would likely not be scraped with them.

In terms of size, most contracts are a few hundred lines of code (up to ~3'000 lines) [84]. Nonetheless, they are complex programs, each occupying at least a couple of auditors for weeks. More importantly, their size does not necessarily represent how difficult it is for a fuzzer to test all paths. For instance, Fig. 1 is very small, but AFL fails to cover all paths within 12h.

6.2 Experimental Setup

We ran different configurations of Harvey and compared the achieved coverage and required time to detect a bug.

Our evaluation focuses on detecting two types of bugs. First, we detect crashes due to assertion violations (SWC-110 according to the Smart Contract Weakness Classification [8]); in addition to user-provided checks, these include checked errors, such as division by zero or out-of-bounds array access, inserted by the compiler. At best, these bugs cause a transaction to be aborted and waste gas fees. In the worst case, they prevent legitimate transactions from succeeding, putting assets at risk. For instance, a user may not be able to claim an auctioned item due to an out-of-bounds error in the code that iterates over an array of bidders to determine the winner. Second, we detect memory-access errors (SWC-124 [8]) that may allow an attacker to modify the persistent state of a contract (Fig. 3). In practice, Harvey covers a wide range of test oracles[3], such as reentrancy and overflows, and also supports custom oracles.

For bug de-duplication, it uses a simple approach (much more conservative than AFL): two bugs of the same type are duplicates if they occur at the same program location.

For each configuration, we performed 24 runs, each with independent random seeds, an all-zero seed input, and a time limit of 1h; we report medians unless stated otherwise. In addition, we performed Wilcoxon-Mann-Whitney U tests to determine if differences in medians are statistically significant based on reported p-values.

We used an Intel® Xeon® CPU @ 2.67GHz 24-core machine with 48GB running Debian Linux 9.11.

6.3 Results

We evaluate Harvey's underlying techniques through four research questions. The first two focus on input prediction, the third on demand-driven sequence fuzzing, and the fourth assesses Harvey's effectiveness in testing custom properties in an industrial setting.

RQ1: Effectiveness of input prediction. To evaluate input prediction, we compare with a baseline (configuration A), which only disables prediction. The first column of Tab. 2 identifies the benchmark, the second the bug, and the third the bug type according to the SWC (110 stands for assertion violations and 124 for memory-access errors).

The fourth and fifth columns show the median time (in secs) after which each unique bug was found by configurations A and B within the time limit—B differs from A only by enabling input prediction. *Configuration B finds 39 out of 46 bugs significantly faster than A.* We report the speed-up factor in the sixth column and the significance level, i.e., p-value, in the seventh (we use $p < 0.05$). As shown in the table, *configuration B is faster than A by a factor of up to 872 (median 5.65).* The last two columns compute the Vargha-Delaney A12 effect sizes [79]. Intuitively, these show the probability of configuration A being faster than B and vice versa. Note that, to compute the median time, we conservatively counted 3'600s for a given run even if the bug was not found. However, *on average, B detects 9 more bugs.*

Tab. 3 compares A and B with respect to instruction coverage. *For 23 out of 27 benchmarks, B achieves significantly higher coverage.* The results for path coverage are very similar.

[3]SWC-101, 104, 107, 110, 113, 123, 124, 127

Table 2: Comparing time-to-bug between configuration A (w/o input prediction) and B (w/ input prediction).

BID	Bug ID	SWC ID	T_A	T_B	$\frac{T_A}{T_B}$	p	$A12_A$	$A12_B$
2	b6e44d03	SWC-110	10.41	**0.43**	23.96	< 0.001	0.05	0.95
2	413fb2d5	SWC-110	43.30	**8.43**	5.14	< 0.001	0.15	0.85
3	e8238b35	SWC-110	12.79	**0.52**	24.42	< 0.001	0.06	0.94
3	1f3f0ef2	SWC-110	25.19	**6.25**	4.03	< 0.001	0.19	0.81
4	8886bf98	SWC-110	28.15	**7.61**	3.70	< 0.001	0.16	0.84
4	48dacbdf	SWC-110	16.63	**0.99**	16.81	< 0.001	0.02	0.98
5	38ba300c	SWC-110	14.25	**0.61**	23.35	< 0.001	0.03	0.97
5	2df51bba	SWC-110	24.16	**6.01**	4.02	< 0.001	0.15	0.85
8	2b1c0cec	SWC-110	3600.00	**306.56**	11.74	< 0.001	0.00	1.00
13	f3e720de	SWC-110	11.48	**5.70**	2.01	0.003	0.25	0.75
13	c1f84a45	SWC-110	18.09	**5.42**	3.34	< 0.001	0.09	0.91
13	9b1b09cf	SWC-110	24.59	**4.75**	5.18	< 0.001	0.08	0.92
13	f3cedeff	SWC-110	13.73	**4.88**	2.82	< 0.001	0.07	0.93
15	a6b732ec	SWC-110	3600.00	**7.28**	494.22	< 0.001	0.00	1.00
15	f8de6e5	SWC-110	3600.00	**559.44**	6.44	< 0.001	0.00	1.00
15	5e9050f	SWC-110	3600.00	3600.00	1.00	1.000	0.50	0.50
17	2247827e	SWC-110	3600.00	**548.00**	6.57	< 0.001	0.02	0.98
18	ac098c9e	SWC-110	24.31	**4.99**	4.87	< 0.001	0.09	0.91
18	36163f3f	SWC-110	**2003.39**	3600.00	0.56	0.006	0.70	0.30
19	effbd6b	SWC-110	29.16	**6.94**	4.20	< 0.001	0.15	0.85
19	5a3b5bb	SWC-110	11.29	**0.53**	21.45	< 0.001	0.07	0.93
19	4106286f	SWC-110	23.82	**5.07**	4.70	< 0.001	0.09	0.91
19	2375ebdd	SWC-110	16.70	**5.76**	2.90	< 0.001	0.15	0.85
19	bc4c025e	SWC-110	25.41	**2.37**	10.71	< 0.001	0.06	0.94
19	95379a77	SWC-110	30.45	**4.98**	6.12	< 0.001	0.11	0.89
19	a2f1b5aa	SWC-110	12.83	**1.24**	10.31	< 0.001	0.04	0.96
22	58e7ec73	SWC-110	6.30	**1.02**	6.15	< 0.001	0.18	0.82
22	aabc1e28	SWC-124	3600.00	**1503.38**	2.39	< 0.001	0.15	0.85
23	59a59c1e	SWC-110	3600.00	**392.09**	9.18	< 0.001	0.00	1.00
23	6b2d3b36	SWC-110	3600.00	**392.62**	9.17	< 0.001	0.00	1.00
24	8fddbdb2	SWC-110	1.85	4.15	0.44	0.585	0.45	0.55
24	20c005cb	SWC-110	77.47	**7.43**	10.43	< 0.001	0.15	0.85
24	3fdc5722	SWC-110	2.52	4.04	0.62	0.464	0.44	0.56
24	358907d3	SWC-110	5.25	5.03	1.04	0.228	0.40	0.60
24	f9967d0a	SWC-110	1.74	4.04	0.43	0.585	0.55	0.45
24	329bb319	SWC-110	260.64	**13.78**	18.91	< 0.001	0.12	0.88
24	3716d2b7	SWC-110	662.44	**9.86**	67.21	< 0.001	0.04	0.96
24	9cb62e3e	SWC-110	3600.00	**4.13**	872.15	< 0.001	0.00	1.00
26	7ef333fe	SWC-110	28.58	**7.03**	4.06	< 0.001	0.06	0.94
27	a4d82705	SWC-110	77.52	**5.31**	14.59	< 0.001	0.07	0.93
27	94fb0959	SWC-110	62.20	**13.62**	4.57	< 0.001	0.09	0.91
27	79788ab0	SWC-110	141.01	**43.80**	3.22	0.001	0.22	0.78
27	f546c070	SWC-110	1645.96	**168.74**	9.75	< 0.001	0.12	0.88
27	874af19f	SWC-110	3600.00	**15.16**	237.45	< 0.001	0.00	1.00
27	1620e17f	SWC-110	3600.00	**459.52**	7.83	< 0.001	0.00	1.00
27	479b458f	SWC-110	3600.00	3600.00	1.00	0.001	0.29	0.71
Median			28.37	**6.13**	5.65			

Input prediction is very effective in both detecting bugs faster and achieving higher coverage.

RQ2: Effectiveness of iterative input prediction. Configuration C differs from B in that it does not iteratively apply the basic approximation step of the Secant method in case it fails to minimize a cost metric. For artificial examples with non-linear branch conditions (e.g., `a^4 + a^2 == 228901770`), we were able to show that this configuration is less efficient than B in finding bugs. However, for our benchmarks, there were no significant time differences between B and C for detecting 45 of 46 bugs. Similarly, in only 2 benchmarks configuration B achieved significantly higher instruction coverage. However, in both cases the differences were relatively small (i.e., less than 10 instructions).

During our experiments with C, *we measured the success rate of one-shot cost minimization to range between 95% and 100% (median*

Table 3: Comparing instruction coverage for configurations A (w/o input prediction) and B (w/ input prediction).

BID	C_A	C_B	$\frac{C_B}{C_A}$	p	$A12_A$	$A12_B$
1	3868.00	3868.00	1.00	0.061	0.36	0.64
2	3064.00	**4004.00**	1.31	< 0.001	0.00	1.00
3	2575.00	**3487.00**	1.35	< 0.001	0.02	0.98
4	2791.00	**3753.00**	1.34	< 0.001	0.00	1.00
5	2567.00	**3501.00**	1.36	< 0.001	0.00	1.00
6	1832.00	**1943.00**	1.06	< 0.001	0.00	1.00
7	1524.00	1524.00	1.00	< 0.001	0.50	0.50
8	1051.00	**2247.50**	2.14	< 0.001	0.00	1.00
9	2694.00	**3378.00**	1.25	< 0.001	0.00	1.00
10	6805.00	**7470.00**	1.10	< 0.001	0.00	1.00
11	7295.00	**8588.00**	1.18	< 0.001	0.00	1.00
12	2816.00	**5013.00**	1.78	< 0.001	0.00	1.00
13	1585.00	**4510.00**	2.85	< 0.001	0.00	1.00
14	3772.50	**4493.00**	1.19	< 0.001	0.00	1.00
15	3488.00	**4720.50**	1.35	< 0.001	0.00	1.00
16	496.00	496.00	1.00	< 0.001	0.50	0.50
17	1832.00	**2757.00**	1.50	< 0.001	0.00	1.00
18	2767.00	**3481.00**	1.26	< 0.001	0.00	1.00
19	2418.00	**2611.00**	1.08	< 0.001	0.00	1.00
20	1635.00	**3028.50**	1.85	< 0.001	0.00	1.00
21	434.00	**440.00**	1.01	< 0.001	0.00	1.00
22	919.00	**1274.00**	1.39	< 0.001	0.00	1.00
23	1344.00	**2095.00**	1.56	< 0.001	0.00	1.00
24	687.00	**754.00**	1.10	< 0.001	0.15	0.85
25	1082.00	**1192.00**	1.10	< 0.001	0.00	1.00
26	1606.00	1606.00	1.00	< 0.001	0.50	0.50
27	4354.00	**5763.00**	1.32	< 0.001	0.00	1.00
Median	2418.00	**3378.00**	1.26			

100%). This suggests that complex branch conditions are not very common in real-world smart contracts.

> Even one iteration of the Secant method is successful in predicting inputs. This suggests that the vast majority of branch conditions in our diverse benchmarks are linear (or piece-wise linear) with respect to the program inputs.

RQ3: Effectiveness of demand-driven sequence fuzzing. To evaluate this research question, we compare configuration A with D, which differs from A by disabling demand-driven sequence fuzzing. In particular, D tries to eagerly explore all paths in all possible transaction sequences, where paths span all transactions. Tab. 4 shows a comparison between A and D with respect to time-to-bug for bugs that were found by at least one configuration. As shown in the table, *A is significantly faster than D in detecting 24 out of 35 bugs, with a speed-up of up to 100x.* Note that 22 of those bugs require more than a single transaction to be detected. Interestingly, configuration A is also significantly faster for 2 bugs that require only a single transaction; we suspect that this is caused by the large number of irrelevant inputs that configuration D adds to the test suite. We observe a similar trend for instruction coverage: *configuration A achieves significantly higher coverage for 15 out of 27 benchmarks.*

In total, 28 out of 35 bugs require more than one transaction to be found. This suggests that real contracts need to be tested with sequences of transactions, and consequently, there is much to be gained from pruning techniques like ours. Our experiments with D also confirm that, when paths span all transactions, the test suite becomes orders-of-magnitude larger.

Table 4: Comparing time-to-bug between configuration A (w/ demand-driven sequence fuzzing) and D (w/o demand-driven sequence fuzzing).

BID	Bug ID	SWC ID	T_A	T_D	$\frac{T_A}{T_D}$	p	$A12_A$	$A12_D$
2	b6e44d03	SWC-110	**10.41**	63.64	0.16	< 0.001	0.86	0.14
2	413fb2d5	SWC-110	**43.30**	2877.12	0.02	< 0.001	0.81	0.19
3	e8238b35	SWC-110	**12.79**	35.65	0.36	0.024	0.69	0.31
3	1f3f0ef2	SWC-110	**25.19**	79.05	0.32	0.047	0.67	0.33
4	8886bf98	SWC-110	**28.15**	456.12	0.06	< 0.001	0.80	0.20
4	48dacbdf	SWC-110	**16.63**	68.36	0.24	0.001	0.78	0.22
5	38ba300c	SWC-110	**14.25**	49.89	0.29	0.006	0.73	0.27
5	2df51bba	SWC-110	**24.16**	124.37	0.19	0.025	0.69	0.31
13	f3e720de	SWC-110	11.48	12.82	0.90	0.628	0.54	0.46
13	c1f84a45	SWC-110	18.09	11.28	1.60	0.106	0.36	0.64
13	9b1b09cf	SWC-110	24.59	25.36	0.97	0.829	0.48	0.52
13	f3cedeff	SWC-110	13.73	11.39	1.21	0.503	0.44	0.56
15	5e9050f	SWC-110	3600.00	3600.00	1.00	0.021	0.60	0.40
18	ac098c9e	SWC-110	24.31	18.14	1.34	0.781	0.48	0.52
18	36163f3f	SWC-110	**2003.39**	3600.00	0.56	< 0.001	0.79	0.21
19	effbd6b	SWC-110	**29.16**	174.63	0.17	0.002	0.76	0.24
19	5a3b5bb	SWC-110	**11.29**	135.14	0.08	< 0.001	0.91	0.09
19	4106286f	SWC-110	23.82	57.13	0.42	0.056	0.66	0.34
19	2375ebdd	SWC-110	16.70	58.44	0.29	0.170	0.62	0.38
19	bc4c025e	SWC-110	25.41	33.70	0.75	0.190	0.61	0.39
19	95379a77	SWC-110	**30.45**	284.46	0.11	0.001	0.79	0.21
19	a2f1b5aa	SWC-110	**12.83**	24.99	0.51	0.028	0.69	0.31
22	58e7ec73	SWC-110	6.30	6.13	1.03	0.490	0.56	0.44
24	8fddbdb2	SWC-110	**1.85**	8.96	0.21	0.001	0.77	0.23
24	20c005cb	SWC-110	**77.47**	1935.29	0.04	< 0.001	0.80	0.20
24	3fdc5722	SWC-110	**2.52**	5.94	0.42	0.028	0.69	0.31
24	358907d3	SWC-110	5.25	6.12	0.86	0.813	0.52	0.48
24	f9967d0a	SWC-110	**1.74**	4.21	0.41	0.004	0.74	0.26
24	329bb319	SWC-110	**260.64**	3600.00	0.07	< 0.001	0.93	0.07
24	3716d2b7	SWC-110	**662.44**	3600.00	0.18	< 0.001	0.88	0.12
26	7ef333fe	SWC-110	**28.58**	3600.00	0.01	< 0.001	1.00	0.00
27	a4d82705	SWC-110	**77.52**	3600.00	0.02	< 0.001	1.00	0.00
27	94fb0959	SWC-110	**62.20**	3600.00	0.02	< 0.001	1.00	0.00
27	79788ab0	SWC-110	**141.01**	3600.00	0.04	< 0.001	0.97	0.03
27	f546c070	SWC-110	**1645.96**	3600.00	0.46	< 0.001	0.94	0.06
Median			**24.31**	68.36	0.29			

> Most bugs require invoking multiple transactions to be revealed. Demand-driven sequence fuzzing is effective in pruning the search space of transaction sequences; as a result, it detects bugs and achieves coverage faster.

RQ4: Effectiveness in industrial setting. We now describe our experience with using HARVEY during a client engagement. The first version of the client's smart contract system is deployed on the Ethereum blockchain and consists of over 6'000 lines of Solidity (excl. libraries). Their test environment features a single main contract and more than 50 additional contracts (incl. several tokens) with which it may interact. In collaboration with the client, we developed 16 functional properties and instrumented the contracts with corresponding checks. We ran 12h fuzzing campaigns (for 24 random seeds) with their existing test deployment as the initial state. We set up HARVEY to generate sequences of transactions that invoke the main contract as well as three additional contracts.

Fig. 4 shows instruction coverage over time for configurations A, B, and D. B is clearly the best and covers up to 48'376 instructions and 9'118 paths (requiring up to 8 transactions). B also detects 12 property violations (requiring up to 6 transactions), which is 2x and 12x more violations than A and D. Manual inspection of the violations revealed subtle omissions in the properties, not the code.

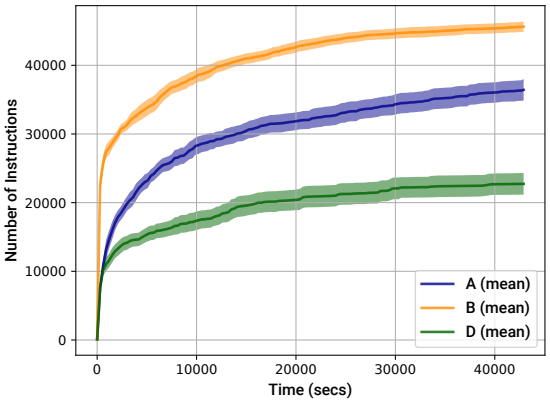

Figure 4: Instruction coverage over time for configurations A, B, and D. Each plot shows the mean coverage (dark) and the 95% confidence interval (lighter).

6.4 Threats to Validity

External validity. Our results may not generalize to all smart contracts or program types [67]. However, we evaluated our technique on a diverse set of contracts from a wide range of domains. We, thus, believe that this selection significantly helps to ensure generalizability. To further improve external validity, we use publicly available benchmarks from related work [84].

Internal validity. Another potential issue has to do with whether systematic errors are introduced in the setup [67]. When comparing configurations, we always used the same seed inputs in order to avoid bias in the exploration.

Construct validity. Construct validity ensures that the evaluation measures what it claims. We compare several configurations of Harvey, and thus, ensure that any improvements are exclusively due to techniques enabled in a given configuration.

7 RELATED WORK

Harvey is an industrial greybox fuzzer for smart contracts. It incorporates two key techniques, input prediction and demand-driven sequence fuzzing, that improve its effectiveness.

Greybox fuzzing. There are several techniques that aim to direct greybox fuzzing toward certain parts of the search space, such as low-frequency paths [17], vulnerable paths [64], deep paths [70], or specific sets of program locations [16]. There are also techniques that boost fuzzing by smartly selecting and mutating inputs [23, 65, 82], or by searching for new inputs using iterative optimization algorithms, such as gradient descent [25], with the goal of increasing branch coverage.

In general, input prediction could be used in combination with these techniques. In comparison, our approach predicts concrete input values based on two previous executions. To achieve this, we rely on additional, but still lightweight, instrumentation.

Whitebox fuzzing. Whitebox fuzzing is implemented in many tools, like EXE [21], jCUTE [66], Pex [76], BitBlaze [68], Apollo [13], S2E [27], and Mayhem [22], and comes in different flavors, such as probabilistic [36] or model based [62].

As discussed earlier, our input-prediction technique does not rely on any program analysis or constraint solving, and our instrumentation is more lightweight.

Hybrid fuzzing. Hybrid fuzzers combine fuzzing with other techniques to join their benefits. For example, Dowser [42] uses static analysis to identify code regions with potential buffer overflows. Similarly, BuzzFuzz [34] uses taint tracking to discover which input bytes flow to "attack points". Hybrid Fuzz Testing [60] first runs symbolic execution to find inputs that lead to "frontier nodes" and then applies fuzzing on these inputs. Driller [71], on the other hand, starts with fuzzing and uses symbolic execution when it needs help in generating inputs that satisfy complex checks.

In contrast, input prediction extends greybox fuzzing without relying on static analysis or whitebox fuzzing. Harvey could, however, benefit from hybrid-fuzzing approaches.

Optimization in testing. Miller and Spooner [56] were the first to use optimization methods in generating test data, and in particular, floating-point inputs. It was not until 1990 that these ideas were extended by Korel [49]. Such optimization methods have recently been picked up again [54], enhanced, and implemented in testing tools, such as FloPSy [52], CORAL [69], EvoSuite [32], AUSTIN [51], CoverMe [33], Angora [25], and Eclipser [28]. Most of these tools use fitness functions to determine the distance from a target and attempt to minimize them. The minimization procedure is typically iterative, e.g., by using hill climbing, simulated annealing, or genetic algorithms [47, 55, 61]. In contrast, for linear cost relations, the Secant method finds roots in a single step.

Our prediction technique is inspired by these approaches but is applied in the context of greybox fuzzing. When failing to minimize a cost metric, Harvey falls back on standard greybox fuzzing.

Method-call sequence generation. For testing object-oriented programs, it is often necessary to generate complex input objects using sequences of method calls. There are many approaches [35, 44, 59, 74, 75, 77, 85, 86] that automatically generate such sequences using techniques such as dynamic inference, static analysis, or evolutionary testing. In contrast, demand-driven sequence fuzzing only relies on greybox fuzzing and targets smart contracts.

Program analysis for smart contracts. There exist various applications of program analysis to smart contracts, such as symbolic execution, static analysis, and verification [12, 15, 18, 24, 26, 31, 39–41, 46, 50, 53, 57, 58, 78, 80, 81]. The work most closely related to ours is the blackbox fuzzer ContractFuzzer [45], the property-based testing tool Echidna [2], and the imitation-learning fuzzer ILF [43]. In contrast to these, our technique applies greybox fuzzing. From these, Echidna is the only tool deployed in industry[4], but its fuzzing technique significantly differs from Harvey, for instance, in requiring an API description of contracts under test.

8 CONCLUSION

We presented Harvey, an industrial greybox fuzzer for smart contracts. During its development, we encountered two key challenges that we alleviate with input prediction and demand-driven sequence fuzzing. Our experiments show that both techniques significantly improve Harvey's effectiveness on both existing benchmarks and a large, industrial benchmark.

[4] https://www.trailofbits.com/

REFERENCES

[1] [n.d.]. The AFL Vulnerability Trophy Case. http://lcamtuf.coredump.cx/afl/#bugs.
[2] [n.d.]. Echidna. https://github.com/trailofbits/echidna.
[3] [n.d.]. Ethereum. https://github.com/ethereum.
[4] [n.d.]. Ethereum White Paper. https://github.com/ethereum/wiki/wiki/White-Paper.
[5] [n.d.]. LibFuzzer—A Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.
[6] [n.d.]. Mythril. https://github.com/ConsenSys/mythril-classic.
[7] [n.d.]. Peach Fuzzer Platform. https://www.peach.tech/products/peach-fuzzer/peach-platform.
[8] [n.d.]. Smart Contract Weakness Classification. https://swcregistry.io.
[9] [n.d.]. Technical "Whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.
[10] [n.d.]. Underhanded Solidity Coding Contest. http://u.solidity.cc.
[11] [n.d.]. zzuf—Multi-Purpose Fuzzer. http://caca.zoy.org/wiki/zzuf.
[12] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP*. ACM, 66–77.
[13] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. 2010. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *TSE* 36 (2010), 474–494. Issue 4.
[14] Massimo Bartoletti and Livio Pompianu. 2017. An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. In *FC (LNCS, Vol. 10323)*. Springer, 494–509.
[15] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *PLAS*. ACM, 91–96.
[16] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *CCS*. ACM, 2329–2344.
[17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *CCS*. ACM, 1032–1043.
[18] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *CoRR* abs/1809.03981 (2018).
[19] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*. USENIX, 209–224.
[20] Cristian Cadar and Dawson R. Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN (LNCS, Vol. 3639)*. Springer, 2–23.
[21] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *CCS*. ACM, 322–335.
[22] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *SP*. IEEE Computer Society, 380–394.
[23] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *SP*. IEEE Computer Society, 725–741.
[24] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Yaron Velner. 2018. Quantitative Analysis of Smart Contracts. In *ESOP (LNCS, Vol. 10801)*. Springer, 739–767.
[25] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *SP*. IEEE Computer Society, 711–725.
[26] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-Optimized Smart Contracts Devour your Money. In *SANER*. IEEE Computer Society, 442–446.
[27] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *ASPLOS*. ACM, 265–278.
[28] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-Box Concolic Testing on Binary Code. In *ICSE*. IEEE Computer Society/ACM, 736–747.
[29] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP*. ACM, 268–279.
[30] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *SPE* 34 (2004), 1025–1050. Issue 11.
[31] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *WETSEB*. IEEE Computer Society/ACM, 8–15.
[32] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *ESEC/FSE*. ACM, 416–419.
[33] Zhoulai Fu and Zhendong Su. 2017. Achieving High Coverage for Floating-Point Code via Unconstrained Programming. In *PLDI*. ACM, 306–319.
[34] Vijay Ganesh, Tim Leek, and Martin C. Rinard. 2009. Taint-Based Directed Whitebox Fuzzing. In *ICSE*. IEEE Computer Society, 474–484.
[35] Pranav Garg, Franjo Ivančić, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. 2013. Feedback-Directed Unit Test Generation for C/C++ Using Concolic Execution. In *ICSE*. IEEE Computer Society/ACM, 132–141.
[36] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *ISSTA*. ACM, 166–176.
[37] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI*. ACM, 213–223.
[38] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*. The Internet Society, 151–166.
[39] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *PACMPL* 2 (2018), 116:1–116:27. Issue OOPSLA.
[40] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *PACMPL* 2 (2018), 48:1–48:28. Issue POPL.
[41] Ákos Hajdu and Dejan Jovanovic. 2019. solc-verify: A Modular Verifier for Solidity Smart Contracts. In *VSTTE (LNCS, Vol. 12031)*. Springer, 161–179.
[42] István Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Security*. USENIX, 49–64.
[43] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *CCS*. ACM, 531–548.
[44] Kobi Inkumsah and Tao Xie. 2007. Evacon: A Framework for Integrating Evolutionary and Concolic Testing for Object-Oriented Programs. In *ASE*. ACM, 425–428.
[45] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *ASE*. ACM, 259–269.
[46] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *NDSS*. The Internet Society.
[47] Scott Kirkpatrick, C. Daniel Gelatt Jr., and Mario P. Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220 (1983), 671–680. Issue 4598.
[48] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *CCS*. ACM, 2123–2138.
[49] Bogdan Korel. 1990. Automated Software Test Data Generation. *TSE* 16 (1990), 870–879. Issue 8.
[50] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Security*. USENIX, 1317–1333.
[51] Kiran Lakhotia, Mark Harman, and Hamilton Gross. 2013. AUSTIN: An Open Source Tool for Search Based Software Testing of C Programs. *IST* 55 (2013), 112–125. Issue 1.
[52] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. 2010. FloPSy—Search-Based Floating Point Constraint Solving for Symbolic Execution. In *ICTSS (LNCS, Vol. 6435)*. Springer, 142–157.
[53] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. ACM, 254–269.
[54] Phil McMinn. 2004. Search-Based Software Test Data Generation: A survey. *Softw. Test., Verif. Reliab.* 14 (2004), 105–156. Issue 2.
[55] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. 1953. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics* 21 (1953), 1087–1092. Issue 6.
[56] Webb Miller and David L. Spooner. 1976. Automatic Generation of Floating-Point Test Data. *TSE* 2 (1976), 223–226. Issue 3.
[57] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. *CoRR* abs/1907.03890 (2019).
[58] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. (2018), 653–663.
[59] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *ICSE*. IEEE Computer Society, 75–84.
[60] Brian S. Pak. 2012. Master's thesis "Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution". School of Computer Science, Carnegie Mellon University, USA.
[61] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. 1999. Test-Data Generation Using Genetic Algorithms. *Softw. Test., Verif. Reliab.* 9 (1999), 263–282. Issue 4.
[62] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2016. Model-Based Whitebox Fuzzing for Program Binaries. In *ASE*. ACM, 543–553.
[63] Siraj Raval. 2016. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. O'Reilly Media.
[64] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-Aware Evolutionary Fuzzing. In *NDSS*. The Internet Society, 1–14.
[65] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Security*. USENIX, 861–875.

[66] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV (LNCS, Vol. 4144)*. Springer, 419–423.
[67] Janet Siegmund, Norbert Siegmund, and Sven Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *ICSE*. IEEE Computer Society, 9–19.
[68] Dawn Xiaodong Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *ICISS (LNCS, Vol. 5352)*. Springer, 1–25.
[69] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Pasareanu. 2011. CORAL: Solving Complex Constraints for Symbolic PathFinder. In *NFM (LNCS, Vol. 6617)*. Springer, 359–374.
[70] Sherri Sparks, Shawn Embleton, Ryan Cunningham, and Cliff Changchun Zou. 2007. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. In *ACSAC*. IEEE Computer Society, 477–486.
[71] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*. The Internet Society.
[72] Melanie Swan. 2015. *Blockchain: Blueprint for a New Economy*. O'Reilly Media.
[73] Don Tapscott and Alex Tapscott. 2016. *Blockchain Revolution: How the Technology Behind Bitcoin is Changing Money, Business, and the World*. Penguin.
[74] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code. In *ESEC/FSE*. ACM, 193–202.
[75] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. 2011. Synthesizing Method Sequences for High-Coverage Testing. In *OOPSLA*. ACM, 189–206.
[76] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex—White Box Test Generation for .NET. In *TAP (LNCS, Vol. 4966)*. Springer, 134–153.
[77] Paolo Tonella. 2004. Evolutionary Testing of Classes. In *ISSTA*. ACM, 119–128.
[78] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS*. ACM, 67–82.
[79] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *JEBS* 25 (2000), 101–132. Issue 2.
[80] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. *PACMPL* 3 (2019), 189:1–189:29. Issue OOPSLA.
[81] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2019. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *VSTTE (LNCS, Vol. 12031)*. Springer, 87–106.
[82] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling Black-Box Mutational Fuzzing. In *CCS*. ACM, 511–522.
[83] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. http://gavwood.com/paper.pdf.
[84] Valentin Wüstholz and Maria Christakis. 2020. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *ICSE*. To appear.
[85] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. 2005. Symstra: A Framework for Generating Object-Oriented Unit Tests Using Symbolic Execution. In *TACAS (LNCS, Vol. 3440)*. Springer, 365–381.
[86] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *ISSTA*. ACM, 353–363.