Olympia: Fuzzer Benchmarking for Solidity

Jana Chadt
e1528032@student.tuwien.ac.at
TU Wien, Austria

Christoph Hochrainer
christoph.hochrainer@tuwien.ac.at
TU Wien, Austria

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

Maria Christakis
maria.christakis@tuwien.ac.at
TU Wien, Austria

Abstract

Over the last few years, smart-contract hacks have resulted in the loss of billions of assets. To efficiently identify such vulnerabilities, academic and industrial researchers have developed several popular smart-contract fuzzers. However, it has been challenging to objectively compare their bug-finding effectiveness. In this paper, we present Olympia, the first benchmark-generation tool that is designed for smart-contract, rather than general-purpose, fuzzers. We have used Olympia to evaluate the effectiveness of four well known, open-source fuzzers for Solidity smart contracts.

Screencast: https://www.youtube.com/watch?v=DdADa2vcicA

Implementation: https://github.com/Rigorous-Software-Engineering/olympia

CCS Concepts

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

Keywords

fuzzing, benchmarking, Solidity

ACM Reference Format:

Jana Chadt, Christoph Hochrainer, Valentin Wüstholz, and Maria Christakis. 2024. Olympia: Fuzzer Benchmarking for Solidity. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3691620.3695352

1 Introduction

Fuzzing [14, 16, 23] has become a popular approach to find bugs in many different systems, ranging from network protocols [18] to reinforcement-learning agents [9] and smart contracts. Smart-contract hacks have resulted in the loss of billions of assets [5]. For this reason, several popular fuzzers, such as Echidna [1, 10], Foundry [2], and Harvey [21, 22], have been developed over the last few years.

So far, it has been challenging to objectively compare the bug-finding effectiveness of these fuzzers. This makes it difficult for users to quickly identify one or more fuzzers for effectively testing smart contracts. It is also difficult for fuzzer developers to evaluate their fuzzing techniques against the state of the art. While there are several fuzzer-benchmarking tools for general-purpose programming languages, such as FuzzBench [15], Fuzzle [12], and Magma [11], there are no similar tools for benchmarking smart-contract fuzzers.

In this paper, we introduce the Olympia benchmarking tool to easily, quickly, and reliably generate benchmarks for smart-contract fuzzers. Each generated smart contract has a known bug, and different fuzzers can be ranked by how quickly they are able to generate an input that triggers the bug. Olympia reuses some parts of Fuzzle, an existing benchmarking tool for general-purpose fuzzers, and builds on our experience with preliminary work [20].

Instead of generating C programs, like Fuzzle does, Olympia generates Solidity smart contracts. Solidity [4] is the most popular programming language for smart contracts that run on the Ethereum virtual machine (EVM). Due to differences in the execution environment between C and Solidity (e.g., related to the use of gas metering for bounding the running time of individual transactions), we encountered several challenges when adapting Fuzzle's benchmark-generation component to emit Solidity code that is suitable for smart-contract fuzzers. We describe our approach in addressing some of these challenges in Sect. 3.

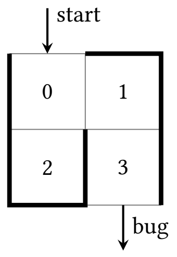
Contributions. Overall, we make the following contributions:

- We present Olympia, the first benchmark-generation tool for smart-contract fuzzers, which we make open source.
- We evaluate Olympia by comparing the bug-finding effectiveness of four popular open-source fuzzers for Solidity smart contracts.

2 Background on Fuzzle

Fuzzle is designed to benchmark fuzzers based on the insight that finding a bug in a program resembles finding the exit in a maze. It, therefore, generates buggy programs from randomly generated mazes, such as the one shown in Fig. 1a. Each maze cell has a unique identifier, and in the figure, cell 0 is the maze entry, and cell 3 exits to a bug. Despite the small maze size, there are still infinitely many paths to the bug, e.g., 0 – 2 – 0 – 1 – 3 – 1 – 3 – bug.

As a next step, Fuzzle uses this maze to generate a program template, shown in Fig. 1b. A program template is a C program with holes (depicted by the boxes in the figure). Each maze cell is represented by a function definition, e.g., `func_0` represents cell 0, etc. A path through the maze is represented by a sequence of function calls, e.g., moving from cell 0 to 1 is represented by the call to `func_1` on line 6. The bug is reached and execution is aborted when `func_bug` is called on line 13. Otherwise, execution terminates



(a) Maze

```
1  void func_bug(signed char *inp, int idx, int l) {
2    abort();
3  }
4  void func_0(signed char *inp, int idx, int l) {
5            // manipulate input
6    if (      ) func_1(inp, idx, l); // go to 1
7    else if (      ) func_2(inp, idx, l); // go to 2
8    else printf("no further steps possible");
9  }
10 // func_1 and func_2 definitions are analogous
11 void func_3(signed char *inp, int idx, int l) {
12           // manipulate input
13   if (      ) func_bug(inp, idx, l); // exit
14   else if (      ) func_1(inp, idx, l); // go to 1
15   else printf("no further steps possible");
16 }
17 // main is omitted
```

(b) Program

Figure 1: A maze and program generated by Fuzzle.

normally when all input is consumed. Function `main`, omitted in the figure, simply calls the maze-entry function, `func_0`, with the fuzzer-generated input (`inp`), an index pointing to the non-consumed part of the input (`idx`), and the input length (`l`).

Finally, Fuzzle fills the holes in the program template as follows. (1) The conditions along valid paths through the maze are generated to be satisfiable using input-range or equality checks. The conditions along a path to the bug may be additionally generated using path constraints, obtained with symbolic execution [7], to known CVEs—the idea is to make it as hard to reach the bug in the maze as it is to detect a known CVE in the program where the CVE was found. (2) Boxes like the ones on lines 5 and 12 check the input length, assign input bytes to variables that are later used in conditions, and update the input index.

3 Overview of Olympia

Olympia reuses the maze-generation component of Fuzzle, and from the generated mazes, produces smart contracts. In the following, we discuss some of the challenges that we address when translating mazes to Solidity contracts.

Solidity contract. From the maze of Fig. 1a, Olympia generates the (partial) Solidity contract shown in Fig. 2. As for C, each function definition represents a maze cell, and the bug is reached when `func_bug` (line 5) is called. The definition of `func_bug`, however, simply sets the persistent `bug` flag to true instead of aborting execution. We explain how this flag is used by the Solidity fuzzers later in this section.

```
1  contract Maze {
2    bool public bug = false;
3    bool private stop = false;
4    int64 next_cell = 0;
5    function func_bug(int8[] memory inp) internal {
6      bug = true;
7    }
8    function func_0(int8[] memory inp) internal {
9              // manipulate input
10     if (      ) next_cell = 1; // go to 1
11     else if (      ) next_cell = 2; // go to 2
12     else stop = true; // no further steps possible
13   }
14   // func_1 and func_2 definitions are analogous
15   function func_3(int8[] memory inp) internal {
16             // manipulate input
17     if (      ) next_cell = -1; // exit
18     else if (      ) next_cell = 1; // go to 1
19     else stop = true; // no further steps possible
20   }
21   function step(int8[] calldata inp) external {
22     require(!stop && !bug,
23       "no further steps possible");
24     if (next_cell == -1) {
25       func_bug(inp);
26       return;
27     }
28     if (next_cell == 0) {
29       func_0(inp);
30       return;
31     }
32     // func_1 and func_2 calls are analogous
33     if (next_cell == 3) {
34       func_3(inp);
35       return;
36     }
37   }
38 }
```

Figure 2: Solidity contract generated by Olympia from the maze of Fig. 1a.

In contrast to C, where `main` is the program entry point, in Solidity, any visible function may be called in any order. To avoid that fuzzers start exploring the maze from any cell, our translation ensures a single entry point. Specifically, we define function `step` (line 21) as the only visible function of the contract. It uses persistent variable `next_cell` to keep track of the next maze cell to enter. Initially, `next_cell` is 0 (line 4), and consequently, `func_0` is called first (line 29). When `next_cell` has the special value -1, `func_bug` is called (line 25). Each cell function, e.g., `func_0` on line 8 and `func_3` on line 15, updates `next_cell` accordingly.

Note that, to test the C program, fuzzers generate input for `main`, which initiates an entire maze exploration from the maze entry until either the exit is reached, all input is consumed, or no further steps are possible. On the other hand, Solidity fuzzers typically generate sequences of transactions (i.e., calls) to a contract under test. Moreover, the running time of each transaction is bounded using gas metering. For these reasons, Olympia generates a `step` function that, in contrast to `main`, performs a single step from one cell to the next. The Solidity fuzzers then need to generate sequences of transactions, each calling `step`, to move through the maze. The function requires the fuzzers to generate at least as much input as is necessary to perform each step.

```
1 contract TestMaze is Test {
2   Maze m;
3   function setUp() external { m = new Maze(); }
4   function invariant_no_bug() external {
5     if (m.bug()) { fail(); }
6   }
7 }
```

Figure 3: The Foundry test contract generated by Olympia.

Last, to fill the holes in the contract, Olympia uses the same mechanisms as Fuzzle.

Fuzzer oracles. Recall that `func_bug` in the generated Solidity code sets the flag `bug` to true instead of aborting execution, as the C code does. We use this flag to signal to Solidity fuzzers that a bug has been found; how exactly `bug` is used depends on each fuzzer.

For example, Echidna is a property-based testing tool and requires that users specify properties under test in functions starting with the prefix `echidna_`. Echidna reports any such properties that are violated during testing. Medusa and ItyFuzz are also compatible with properties defined for Echidna. To benchmark these fuzzers with Olympia, we define the following property:

```
function echidna_no_bug() external
                          returns (bool) {
  return !bug;
}
```

The above function appears at the end of the generated contract `Maze` (Fig. 2).

In contrast, Foundry requires creating a test contract, such as `TestMaze` shown in Fig. 3 (inheriting from the Foundry base contract `Test`). The `setUp` function runs before any tests are executed. In our case (line 3), it deploys the `Maze` contract of Fig. 2. Foundry also allows expressing invariants, such as the one on line 4, which should hold during a fuzzing campaign of the deployed contract(s). Our invariant calls the Foundry-specific function `fail`, which fails a test, when the `bug` flag of `Maze` is true. Note that we define `bug` as `public` (line 2 of Fig. 2) to be able to use it in Foundry test contracts—on line 5 of Fig. 3, we call its (automatically generated) getter function `bug()`.

Olympia generates both Echidna- and Foundry-targeted benchmarks, which are then distributed accordingly to the fuzzers.

Type casts. Recall that Fuzzle and Olympia may generate conditions along a path to a bug using path constraints derived from known CVEs. These constraints are SMT formulas and may manipulate bit-vectors by applying signed or zero extensions.

More specifically, a zero extension adds leading zeros to a given bit-vector until the required size is reached. A signed extension, on the other hand, interprets the given bit-vector as a signed integer using two's complement; if the first bit is zero, the integer is positive, otherwise, it is negative. In the former case, a signed extension adds leading zeros, and in the latter, it adds leading ones. For example, Fuzzle translates to C a zero extension of an 8-bit vector by 24 bits as `(uint32_t)x`, where `x` is the variable representing the bit-vector. A signed extension is translated to `(int32_t)x`.

Solidity, however, does not allow changing the signedness of a variable (i.e., from unsigned to signed and vice versa) while, at the same time, changing the type size. In particular, for the above example, if `x` is an `int8` variable, we cannot translate the zero extension to `uint32(x)`—this changes both the signedness and size of `x`. We, instead, need to apply an intermediate cast that changes the signedness of the variable before changing the type size. Concretely, the zero extension is translated to `uint32(uint8(x))`, whereas the signed extension is translated to `int32(x)` since `x` is already signed. Conversely, if `x` were a `uint8`, the zero extension would be translated to `uint32(x)` and the signed extension to `int32(int8(x))`.

4 Experimental Evaluation

For our experiments, we use Olympia to generate 50 Solidity benchmarks for each of four different maze dimensions (5x5, 10x10, 15x15, and 20x20); we, therefore, generate a total of 200 benchmarks. To produce these benchmarks, all other Olympia parameters, which it inherits from Fuzzle, for instance regarding condition-generation strategies or maze-generation algorithms, are chosen randomly.

We benchmark four popular Solidity fuzzers, that is, Foundry [2], Medusa [3], Echidna [1, 10], and ItyFuzz [19]. To each fuzzer, we give a 60-minute timeout per benchmark.

We performed all experiments on a machine with two AMD EPYC 9474F CPUs @ 3.60GHz and 1.5TB of memory, running Debian GNU/Linux 12 (bookworm).

Results. Fig. 4 shows the results of our experiments. More specifically, the bar chart in Fig. 4a shows how many mazes each of the four fuzzers was able to solve within the time limit—that is, in how many generated contracts each fuzzer detected a bug. Unsurprisingly, the fuzzers perform better for the smaller mazes and less well for the larger ones. Overall, out of the 200 mazes, ItyFuzz solves 147, Echidna 83, Medusa 58, and Foundry 31. ItyFuzz significantly outperforms all other fuzzers on all maze dimensions.

The cactus plot in Fig. 4b shows the number of solved mazes versus the time (in minutes) it took to solve each maze for all fuzzers. ItyFuzz solves the mazes between 140.0msecs and 3.7mins, Echidna between 6.1secs and 58.3mins, Medusa between 13.1secs and 59.2mins, and Foundry between 12.5secs and 20mins.

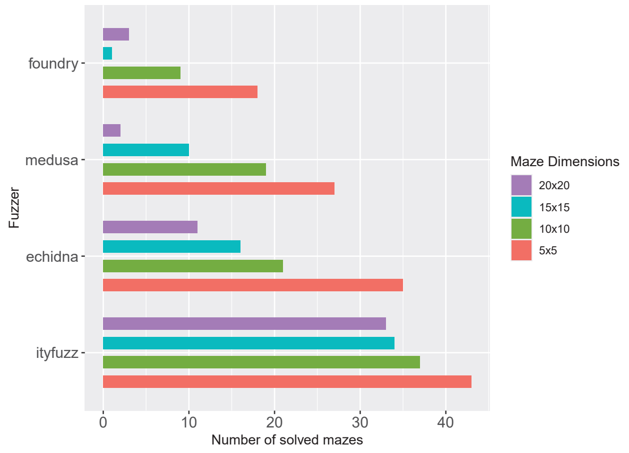
Fig. 4c shows the intersection of mazes solved by the fuzzers; ItyFuzz solves all mazes that the other fuzzers solve.

Olympia provides a fast, easy, and reliable platform for evaluating fuzzers for Solidity smart contracts. Results such as those discussed above can help users make more informed choices regarding which fuzzers to run for more effective bug finding; they can also help fuzzer developers improve their techniques by comparing against the state of the art.

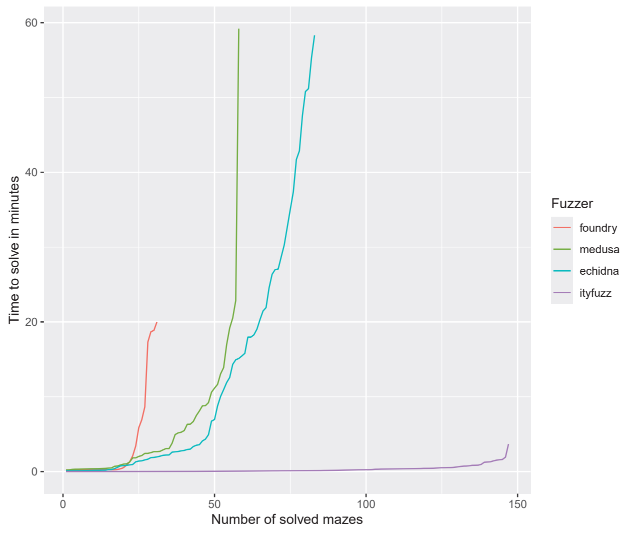
5 Related Work

Here, we briefly present work related to fuzzer benchmarking. On one hand, there are synthetic benchmarks, e.g., LAVA [8] and Fuzzle [12]. On the other hand, there are benchmarks based on real programs, such as FuzzBench [15], Magma [11], and UNIFUZZ [13]. There are also benchmarks created for a competition of testing tools, namely Test-COMP [6].

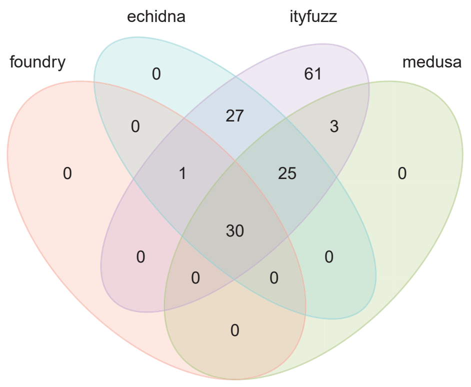
In contrast to benchmarks generated by Olympia, all of these target general-purpose fuzzers. Although synthetic, Fuzzle and Olympia's benchmarks may approximate the difficulty of detecting real bugs by incorporating CVEs. Moreover, unlike fixed sets of real-world benchmarks, Olympia generates random benchmarks based



(a) Number of solved mazes per benchmarked fuzzer.



(b) Time (in minutes) to solve each maze per benchmarked fuzzer.



(c) Intersection of mazes solved by the fuzzers.

Figure 4: Benchmarking Foundry, Medusa, Echidna, and ItyFuzz using Olympia.

on several inputs, e.g., maze dimensions, condition-generation strategies, maze-generation algorithms, etc., that it inherits from Fuzzle. This helps in preventing fuzzer over-fitting to fixed benchmark sets [17].

6 Conclusion

In this paper, we have introduced Olympia, the first benchmark-generation tool for smart-contract fuzzers. Olympia automatically generates buggy smart contracts that can be used to objectively compare the bug-finding effectiveness of different fuzzers. Olympia is open source, and we have used it to evaluate the effectiveness of four popular fuzzers for Solidity contracts.

Acknowledgments. This work was supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22007].

References

[1] [n. d.]. Echidna. https://github.com/trailofbits/echidna.
[2] [n. d.]. Foundry. https://github.com/foundry-rs/foundry.
[3] [n. d.]. Medusa. https://github.com/crytic/medusa.
[4] [n. d.]. Solidity. https://github.com/ethereum/solidity.
[5] 2024. Funds Stolen from Crypto Platforms Fall More Than 50% in 2023, but Hacking Remains a Significant Threat as Number of Incidents Rises. https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2024.
[6] Dirk Beyer. 2019. Competition on Software Testing (Test-COMP). https://test-comp.sosy-lab.org.
[7] 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.
[8] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *S&P*. IEEE Computer Society, 110–121.
[9] Hasan Ferit Eniser, Timo P. Gros, Valentin Wüstholz, Jörg Hoffmann, and Maria Christakis. 2022. Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing. In *ISSTA*. ACM, 52–63.
[10] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In *ISSTA*. ACM, 557–560.
[11] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Meas. Anal. Comput. Syst.* 4 (2020), 49:1–49:29. Issue 3.
[12] Haeun Lee, Soomin Kim, and Sang Kil Cha. 2022. Fuzzle: Making a Puzzle for Fuzzers. In *ASE*. ACM, 45:1–45:12.
[13] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *Security*. USENIX, 2777–2794.
[14] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *TSE* 47 (2021), 2312–2331. Issue 11.
[15] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *ESEC/FSE*. ACM, 1393–1403.
[16] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *CACM* 33 (1990), 32–44. Issue 12.
[17] Jiradet Ounjai, Valentin Wüstholz, and Maria Christakis. 2023. Green Fuzzer Benchmarking. In *ISSTA*. ACM, 1396–1406.
[18] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *ICST*. IEEE Computer Society, 460–465.
[19] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *ISSTA*. ACM, 322–333.
[20] Valentin Wüstholz. [n. d.]. Daedaluzz. https://github.com/Consensys/daedaluzz.
[21] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts. In *ESEC/FSE*. ACM, 1398–1409.
[22] Valentin Wüstholz and Maria Christakis. 2020. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *ICSE*. ACM, 789–800.
[23] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org.