A Comparative Analysis of Smart Contract Fuzzers' Effectiveness

Antonio Viggiano
agfviggiano@gmail.com

Abstract

This study presents a comparative analysis of randomized testing algorithms, commonly known as fuzzers, with a specific emphasis on their effectiveness in catching bugs in Solidity smart contracts. We employ the non-parametric Mann Whitney U-test to gauge performance, defined as the "time to break invariants per mutant", using altered versions of the widely-forked Uniswap v2 protocol. We conduct 30 tests, each with a maximum duration of 24 hours or 4,294,967,295 runs, and evaluate the speed at which the fuzzers Foundry and Echidna can breach any of the 22 protocol's invariant properties for each of the 12 mutants, created both with mutation testing tools and with manual bug injection methods. The research shows significant performance variabilities between runs for both Foundry and Echidna depending on the instances of mutated code. Our analysis indicates that Foundry was able to break invariants faster in 9 out of 12 tests, while Echidna in 1 out of 12 tests, and in the remaining 2 tests, the difference in performance between the two fuzzers was not statistically significant. The paper concludes by emphasizing the necessity for further research to incorporate additional fuzzers and real-world bugs, and paves ground for further developments of more precise and rigorous evaluations of fuzzer effectiveness.

CCS Concepts: • **Security and privacy → Software and application security**.

Keywords: blockchain, smart contracts, fuzz testing, mutation testing, application security

1 Introduction

A fuzz tester (or fuzzer) is a tool that randomly generates inputs in iterative fashion to test a target program. Research demonstrates that fuzzers can be surprisingly effective [1], and within the realm of blockchain, it has been established that custom user-defined properties in fuzzing can detect up to 63% of the most severe and exploitable flaws in contracts [2].

Several fuzz testing tools exist for Ethereum smart contracts, such as Echidna [3], Foundry [4], Medusa [5], and Harvey [6], yet few studies offer comparative analysis to assess their effectiveness. Recent efforts to automatically generate benchmarks for smart-contract fuzzers have emerged [7], with the limitation of measuring a fuzzer's performance in non-production code. Hence, different methodologies may still be valuable when comparing fuzzers against typical DeFi smart contracts.

In order to circumvent general issues found in existing experimental evaluations, which could potentially lead to inaccurate or misleading assessments, this paper seeks to adhere to the methodology and guidelines prescribed by Klees et al. [8], which will be presented in Section 2.

2 Methodology

Evaluating a fuzz testing algorithm A involves several steps: (a) selecting a baseline algorithm B for comparison; (b) choosing a representative set of target programs for testing; (c) determining how to measure A's performance against B's, ideally in terms of bugs found; (d) establishing algorithm parameters, such as the choice of seed files and the algorithm's run duration; and (e) conducting multiple runs for both A and B and statistically comparing their performance.

A study surveying 32 fuzzing papers by Klees et al. [8] uncovered problems in every evaluation reviewed, which led to the formation of guidelines designed to improve the experimental evaluations of fuzz testing algorithms and to enhance the robustness of reported results. In brief, they recommend comparing the median and standard deviation over many runs, using different seeds, using different test environments, fuzzing over long hours, plotting performance over time, measuring unique bugs by counting each single conceptual bugfix, directly measure the number of bugs found, and using a statistical test to compare performance.

In our study, we selected `echidna 2.2.0` (using `slither 0.9.3`) and `forge 0.2.0 (588ad27)` for comparison and tested the Uniswap v2 protocol [9], compiled with `solc 0.8.20`, with a set of 22 properties that should hold under stateful invariant tests. We gauged performance by measuring the time taken to find injected bugs in 12 mutant tests using both a mutation testing tool and some manually injected bugs. We selected 30 different seeds and allowed the fuzzers to run with a 24-hour timeout or $2^{32}-1 = 4294967295$ runs on a 1 vCPU general purpose `m3.medium` AWS EC2 instance on `Ubuntu 22.04.2 LTS` [10].

We conducted multiple runs and statistically compared their performance by analyzing the median time taken to find the bug on each mutant and the standard deviation between runs for different seed values and applying the Mann-Whitney U Test.

DeFi Security Summit 2023, July 15-16, 2023, Paris, France

It should be noted that slight alterations to the methodology prescribed in [8] were carried out due to operational constraints and particularities of Solidity smart contract fuzzers. In particular, Klees et al. suggests running fuzzers N times with the same configuration, each with a timeout T, and tallying the unique number of bugs found after the timeout. In our case, we performed N runs, each with T timeout and a different seed, and and measured the time to find the bug present in each mutant. Our approach is due to the practical difficulties of making the fuzzers continue running after the invariants break, and the complexities involved in analyzing which bug was found when several are injected simultaneously. By running the fuzzers against code containing one injected bug at a time, it was possible to analyse how the performance varies between each mutant and more easily derive conclusions from the results.

2.1 Choosing algorithms to compare

For our evaluation, we selected Echidna [3] and Foundry [4], two popular open source fuzz tools used by Ethereum smart contract developers, as evidenced by their GitHub stars — 6.2k for Foundry and 2.2k for Echidna respectively.

2.2 Choosing a representative set of target programs to test

The reference paper [8] advises testing fuzzers against many different programs representative of the target population, preferably over 100, in order to establish the general superiority of one fuzzing algorithm over another. Due to time constraints, however, we limited our analysis to just one program, Uniswap v2 [9]. We chose this protocol not only for being a fair representative of DeFi smart contracts in general, but also for its sheer relevance to the blockchain ecosystem: according to DeFiLlama [11], a major DeFi TVL aggregator, Uniswap v2 is the most forked protocol in crypto, with 437 forks valued over $2.6 billion at the time of writing, and the number one decentralized exchange (DEX) [12], with over $4.0b in Total Value Locked (TVL).

2.3 Choosing how to measure performance

When dealing with statistical measurements, high variance can render a difference in averages statistically insignificant. To mitigate this, it is recommended to employ a statistical test [13] that can provide an indication of whether the observed performance difference is due to a real effect rather than a mere product of chance.

For randomized testing algorithms like fuzzers, Arcuri and Briand [14] recommend using the Mann Whitney U-test to determine the stochastic ranking between two algorithms, A and B. The test assesses whether outcomes from A's data sample are more likely to exceed those in B's. The advantage of the Mann Whitney test is that it is non-parametric, meaning it does not make assumptions about the distribution of a randomized algorithm's performance. When comparing the median values of the two groups, if the Mann-Whitney U test yields a p-value less than 0.05, it indicates the distributions are significantly different and helps us infer which one is more efficient.

In our evaluation, we gauge performance based on the time required to break invariants per mutant. We chose this parameter because it provides a measure of efficiency and effectiveness, given that a superior fuzzer should exhibit both speed and precision in identifying and breaking invariants. Moreover, the choice of this criterion is rooted in the real-world application of fuzzers where time is often a critical factor, and the faster a fuzzer can identify and break invariants, the more efficient the debugging process becomes. Therefore, our analysis results are specific to the selected mutants and to the specific configurations and choice of algorithm parameters in our study.

2.4 Choosing algorithm parameters

Our analysis includes 30 seeds, 12 mutants, and a set of 22 properties. The seed values were chosen based on a combination of natural numbers and pseudorandom 32-byte unsigned integers. These values include `0`, `1`, `2`, `type(int64).max`, and `uint256(keccak256(abi.encodePacked(uint256(i))))`, where `i` varies from `0` to `25`.

The mutants were initially selected from a pool of 20, further narrowed down to 15, and eventually to 12. The selection process involved a combination of the mutation testing tools Slither [15] and Gambit [16], as well as the manual injection of bugs. Manually injected bugs were designed to simulate common errors made by developers, such as rounding errors (as shown below) or missing state variable updates.

```
diff --git a/UniswapV2Library.sol b/UniswapV2Library.sol
index 83cb0b6..3991c50 100644
--- a/UniswapV2Library.sol
+++ b/UniswapV2Library.sol
@@ -84,7 +84,7 @@ library UniswapV2Library {
         uint amountInWithFee = amountIn.mul(997);
         uint numerator = amountInWithFee.mul(reserveOut);
         uint denominator = reserveIn.mul(1000).add(amountInWithFee);
-        amountOut = numerator / denominator;
+        amountOut = (numerator / denominator).add(1);
     }
```

After the initial generation, we discarded 5 mutants crafted by `slither-mutate` due to their tendency to inject an excessive number of bugs by default, which would render the evaluation of results more challenging. In a second step, 3 other mutants were discarded due to a failure in producing any crashes within the 24-hour timeout or $2^{32} - 1 = 4294967295$ runs for any of the fuzzers. Out of the 12 final patch files, 3 were generated by Gambit and 9 manually.

The choice of the test duration was set to one day following the suggestion of [8], with the limitation of a maximum number of runs due to Foundry using Rust's `u32` (unsigned 32-bit integer) type for the `runs` parameter, below Echidna's usage of Haskell's `Int` (signed 64-bit integer) for the `testLimit` parameter.

Regarding the selection of invariants, we performed a comprehensive analysis of the Uniswap v2 Core protocol properties, which are detailed in Table 1.

Besides the choice of test parameters, we conducted the invariant tests following a stateful invariant testing approach. In stateful fuzzing, the ending state of one fuzzing iteration serves as the initial state for the subsequent iteration, which can be particularly sensitive to initial conditions. In our case, we set the initial token balance of the `User.sol` handler smart contract during the first transaction, and this balance is not replenished in subsequent transactions.

The fuzzer parameters were chosen to closely align the behavior of the two tools within the limitations of the test environment. Specifically, Echidna's `testMode` was set to assertion, `shrinkLimit` was set to zero, and `stopOnFail` was enabled. In addition, Foundry's `depth` was set to 25 to match Echidna's `seqLen`, and `fail_on_revert` was disabled.

```
# config.yaml
testMode: assertion
corpusDir: corpus
seqLen: 25
testLimit: 4294967295
shrinkLimit: 0
stopOnFail: true
```

```
# foundry.toml
[invariant]
runs = 4294967295
depth = 25
fail_on_revert = false
call_override = false
dictionary_weight = 80
include_storage = true
include_push_bytes = true
```

To ensure maximum code coverage, we studied the maintenance of invariants in both successful and failed calls. This enabled the test contract to identify particular values contributing to test failure, as can be seen in properties P-08, P-09, P-16, P-17, and P-22 from Table 1.

2.5 Comparing algorithm performances

To execute the benchmark, we crafted a collection of Terraform [17] and Packer [18] configuration files and bash scripts for the deployment of an infrastructure consisting of numerous AWS EC2 instances. These instances operated the benchmark and collected results, which are available for review at [19].

In order to analyse the results, a Python script was created to plot results and perform statistical tests to assess fuzzer performance.

3 Results

The summarized results are presented in Figure 1. This chart offers a logarithmic representation of the "Time to break invariants" for each Mutant, measured in seconds. To better understand the dispersion and central tendency of the results, we have utilized a box-and-whisker plot, overlaid with a scatterplot of individual runs. The box in the box-and-whisker plot, often referred to as the "box" part of the plot, is drawn from the first quartile (Q1) to the third quartile (Q3) of the data, with a line at the median (the second quartile, Q2). This visualization technique effectively shows the interquartile range (the range between Q1 and Q3), representing the middle 50% of the data. The line within the box depicts the median value, providing a picture of the central tendency of the data. The scatterplot overlay helps in visualizing the individual data points, thus providing additional insight into the data distribution beyond the summary statistics presented in the boxplot. At the bottom of each test, the p-value from the Mann-Whitney U Test is annotated, offering further statistical significance context.

The analysis indicates that different mutants produce significantly different outcomes in terms of performance variability for each fuzzer. For both Echidna and Foundry, different runs can have a drastic three orders of magnitude difference in performance depending on the code under test. Some tests may finish within minutes, while others can stretch for several hours.

With regard to the difference in performance, the results show that, out of 12 mutants, Foundry is able to identify bugs quicker in 9 of them (mutants 01, 03, 06, 07, 08, 09, 10, 11, and 12), and Echidna in 1 of them (mutant 05), while in the other 2 (mutants 02, and 04), there were no display statistically significant difference between the two fuzzers.

It should be noted that although both programs were configured to fuzz for up to $2^{32} - 1 = 4294967295$ runs, Foundry produced some false negatives, in which it finished successfully after around 8 hours but did not break any of the invariants. We chose not to discard these results in the same way as timeouts produced by Echidna were counted for their elapsed time in the statistics.

4 Conclusion and future work

This paper delivers a comparative examination of randomized testing algorithms, primarily focusing on their capacity to detect bugs in Solidity smart contracts. We measured their efficacy via "time to break invariants per mutant", employing modified versions of the extensively-forked Uniswap v2 protocol.

We determined that performance can vary greatly across different runs due to initial conditions or seed values, highlighting the importance of numerous, extensive testing runs.

Regarding the difference in time to break invariants, Foundry demonstrates superior performance for 9 out of 12 tests,

Property	Description
P-01	Adding liquidity increases k
P-02	Adding liquidity increases the total supply of LP tokens
P-03	Adding liquidity increases reserves of both tokens
P-04	Adding liquidity increases the user's LP balance
P-05	Adding liquidity decreases the user's token balances
P-06	Adding liquidity does not decrease the `feeTo` LP balance
P-07	Adding liquidity for the first time should mint LP tokens equals to the square root of the product of the token amounts minus a minimum liquidity constant
P-08	Adding liquidity should not change anything if it fails
P-09	Adding liquidity should not fail if the provided amounts are within the valid range of `uint112`, would mint positive liquidity and are above the minimum initial liquidity check when minting for the first time
P-10	Removing liquidity decreases k
P-11	Removing liquidity decreases the total supply of LP tokens if fee is off
P-12	Removing liquidity decreases reserves of both tokens
P-13	Removing liquidity decreases the user's LP balance
P-14	Removing liquidity increases the user's token balances
P-15	Removing liquidity does not decrease the `feeTo` LP balance
P-16	Removing liquidity should not change anything if it fails
P-17	Removing liquidity should not fail if the returned amounts to the user are greater than zero
P-18	Swapping does not decrease k
P-19	Swapping increases the sender's tokenOut balance
P-20	Swapping decreases the sender's tokenIn balance by swapAmountIn
P-21	Swapping does not decrease the `feeTo` LP balance
P-22	Swapping should not fail if there's enough liquidity, if the output would be positive and if the input would not overflow the valid range of `uint112`

Table 1. Uniswap v2 properties

while Echidna performs better in 1 out of 12 tests, and in the other 2 tests, the difference was not statistically significant. Due to the constraints of the test setup, several features of both tools were not assessed, including aspects like coverage guidance, shrinking, and multi-core support. Also, the configuration parameters were not necessarily fine-tuned for each specific fuzzer, such as the sequence length or corpus generation. Consequently, to ensure thorough testing, we recommend the utilization of multiple fuzzers.

Areas for future research could involve expanding the number of tested protocols, in order to enhance the study's generalizability, and increasing the number of evaluated fuzzers. Conducting tests in varying environments, testing different fuzzer configurations, assessing fuzzers against real-world issues rather than artificial bugs, analyzing performance progression over time, and creating a robust, independently defined benchmark suite would further contribute to the research.

While our research offers significant insights into the analysis of fuzzers' effectiveness in finding bugs in typical DeFi smart contracts, its limitations preclude a comprehensive conclusion. Nevertheless, we hope that this study will inspire and direct future researchers toward continuous improvement in the tooling and benchmarks used in this space.

References

[1] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[2] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In *International Conference on Financial Cryptography and Data Security*, 2020.

[3] Trail of Bits. Echidna: Ethereum fuzz testing framework, 2018. URL https://github.com/crytic/echidna.

[4] Foundry: a blazing fast, portable and modular toolkit for ethereum application development written in rust, 2022. URL https://github.com/foundry-rs/foundry.

[5] Trail of Bits. Parallelized, coverage-guided, mutational solidity smart contract fuzzing, powered by go-ethereum, 2023. URL https://github.com/crytic/medusa.

[6] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. 2019.

[7] Valentin Wüstholz. Benchmarking smart-contract fuzzers, 4 2023. URL https://consensys.net/diligence/blog/2023/04/benchmarking-smart-contract-fuzzers/.

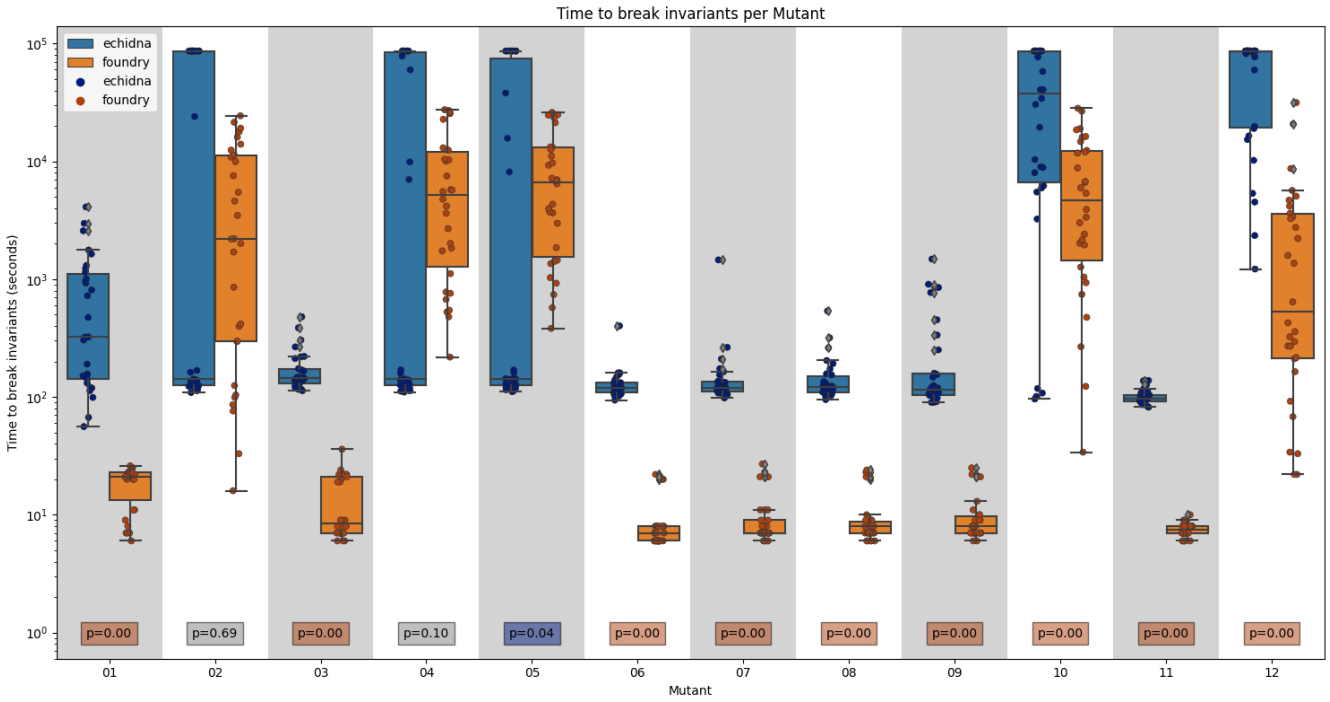

Figure 1. Time to break invariants per Mutant

[8] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. *CoRR*, abs/1808.09700, 2018. URL http://arxiv.org/abs/1808.09700.

[9] Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core, 3 2020. URL https://uniswap.org/whitepaper.pdf. hayden@uniswap.org, noah@uniswap.org, dan@paradigm.xyz.

[10] Amazon Web Services. Amazon ec2 previous generation instances pricing | aws. URL https://aws.amazon.com/ec2/previous-generation/.

[11] Forks - defillama, . URL https://defillama.com/forks.

[12] Dexes tvl rankings - defillama, . URL https://defillama.com/protocols/Dexes.

[13] R. Lyman Ott and Micheal T. Longnecker. *Introduction to Statistical Methods and Data Analysis (with CD-ROM)*. 2006.

[14] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2011.

[15] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither Analyzer, June 2023. URL https://github.com/crytic/slither.

[16] Certora. Gambit, 2023. URL https://github.com/Certora/gambit.

[17] HashiCorp. Terraform by hashicorp, . URL https://www.terraform.io/.

[18] HashiCorp. Packer by hashicorp, . URL https://www.packer.io/.

[19] Antonio Viggiano. Evaluating fuzzer effectiveness, 2023. URL https://github.com/aviggiano/fuzzer-evaluation.