Echidna: Effective and Usable Fuzzing for Smart Contracts

Gustavo Grieco
Will Song
Artur Cygan
Trail of Bits

Alex Groce
Northern Arizona University

ABSTRACT

Ethereum smart contracts are autonomous programs, operating on a blockchain, that often control significant financial or intellectual property resources. Bugs in smart contracts can be catastrophic, so effective test generation for smart contracts is critical for developers. This paper introduces an open source smart contract fuzzer called Echidna that makes it easy to automatically generate tests to detect assertion violations and check custom properties for smart contracts. Echidna is easy to use out of the box, requiring no complex configuration or deployment of contracts to a local blockchain, and is built to offer rapid feedback, with many property violations detected in less than two minutes. Echidna has been heavily tuned via empirical experiments to set default configuration parameters (e.g. how often to use mined constants, and how many transactions to include in generated sequences). Echidna has been used in more than 10 large-scale for-pay security audits, and feedback from those audits has strongly informed the evolution of Echidna, both in terms of practical usability with, e.g., smart contract frameworks such as truffle and embark, and in terms of test generation strategies.

CCS CONCEPTS

• **Software and its engineering → Dynamic analysis**; **Software testing and debugging**.

KEYWORDS

smart contracts, fuzzing, test generation

ACM Reference Format:
Gustavo Grieco, Will Song, Artur Cygan, and Alex Groce. 2020. Echidna: Effective and Usable Fuzzing for Smart Contracts. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Smart contracts for the Ethereum blockchain [3], usually written in the Solidity language [15], facilitate and verify high-value financial transactions, as well as tracking physical goods and intellectual property. It is essential that such largely autonomous programs be correct and secure. Unfortunately, such contracts are often *not* correct and secure [2]. Recent work surveying and categorizing flaws in critical contracts [7] established that fuzzing, using custom user-defined properties, might detect up to 63% of the most severe and exploitable flaws in contracts. It is therefore critical to make high-quality, easy-to-use fuzzing available to smart contract developers and security auditors.

Echidna [13] is an open source Ethereum smart contract fuzzer that supports user-defined properties (for property-based testing [4], as well as assertion checking and gas use estimation. Echidna supports the Solidity and Vyper smart contract languages and most contract development frameworks, including truffle and embark. Trail of Bits has used Echidna internally in numerous code audits [7, 14]. The use of Echidna in internal audits is a key driver in three primary design goals for Echidna. Echidna must be:

(1) **Effective:** If Echidna is unable to explore contract behavior in order to find faults, e.g., producing poor coverage of contract/blockchain state, then it is not a useful tool to apply in code audits. Both security auditors and other users' most important goal is to be able to find bugs.
(2) **Usable:** If Echidna is difficult to apply to smart contracts, auditors and other users will not find it effective, even if technically the tool performs well once it can be applied. Difficult-to-automate workflows, reliance on external tools (e.g., forcing users to set up a private blockchain and deploy contracts to it), unsupported language features, and complex configuration options without meaningful defaults: all of these limit the practical, if not theoretical, effectiveness of a tool. Echidna's design is focused on ease-of-use for practical bug-finding.
(3) **Fast:** Finally, it is worth noting as a separate category, though relevant to both effectiveness and usabliity, that Echidna is designed to provide *fast* feedback in most cases. While some subtle flaws inherently require a long analysis time, Echidna is tuned to detect violations quickly when this is possible.

The third design goal is, in our experience, essential for supporting the first two design goals. Quick turnaround makes it practical to apply a tool during development, and makes tuning user-defined properties easy; if every run requires even 10 minutes to produce useful results, incremental improvement of properties is painful and inefficient. This is why most property-based testing tools have a very small default runtime or number of tests. Speed also makes use of a tool in CI (https://crytic.io) more practical. Finally, effectiveness can be more easily improved if turnaround is usually quick: a fast fuzzer can be tuned using mutation testing [11] or experimental runs using a large set of configurations on a large set of benchmark contracts. The size of the statistical basis for decision-making and parameter choices explored is directly limited by the runtime of the tool. Of course, Echidna supports long runtimes: there is no upper bound on how long Echidna can execute, and with coverage-based

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

feedback there is a long-term improvement in test generation quality. Nonetheless, Echidna aims to find most problems it can detect in a matter of minutes, preferably fewer than five minutes.

2 ARCHITECTURE AND DESIGN

2.1 Echidna Architecture

2.2 Continuous Improvement

A key element of Echidna's design is to make continuous improvement of functionality sustainable. Echidna has an extensive test suite to ensure improvements do not degrade existing features, and uses Haskell language features to maximize abstraction and applicability of code to new testing approaches.

2.2.1 Tuning Parameters. Echidna provides flexibility to users by providing a large number of configurable parameters that control various aspects of testing, including some that have a major impact on test generation. As we write, there are more than 30 settings, controlled by providing Echidna with a `.yaml` configuration file. However, in order to avoid overwhelming users with complexity, and to make the out-of-the-box experience as good as possible, every such setting has a frequently-reviewed default setting. Default settings with a large impact on test generation have all been tuned using some combination of mutation testing (via the universalmutator tool [8]) on realistic contracts and evaluation over large benchmark sets. Rather than always tune to a single set of fixed benchmarks, we have thus far varied the benchmarks used, and confirmed results with different approaches, to avoid over-fitting to a single set of contracts or bugs. Empirical, statistical, tuning produced some surprises: e.g., the dictionary of mined constants was initially only used infrequently in transaction generation, but we found that coverage could often be improved significantly by using constants 40% of the time.

3 USAGE

4 EXPERIMENTAL EVALUATION

In order to evaluate Echidna, we compared its performance to the MythX platform [6], accessed via the `solfuzz` [10] interface, on a set of reachability targets, produced by insertion of `assert(false)` statements, on a set of benchmark contracts [1] produced for the VeriSmart safety-checker [12]. We compared to MythX as it was the only other currently-working fuzzer we are aware of that easily supports arbitrary reachability targets (via supporting assertion-checking). Comparing with a fuzzer that only supports a custom set of built-in detectors, such as ContractFuzzer [9], which does not support arbitrary assertions in Solidity code, is difficult to do in a fair way, as any differences are likely to be due to specification semantics, not exploration capability. ChainFuzz, which in theory offers this capability, unfortunately fails to run on our examples, or its own provided sample contract, terminating with the message: `panic: error unmarshalling transaction: json: cannot unmarshal string into Go struct field transaction.gas of type uint64`, when run inside the docker image it creates. The last commit was in April of 2019, and it is not clear if the tool is maintained, since the same problem that we encountered was submitted as a github issue in April of 2019: https://github.com/ChainSecurity/ChainFuzz/issues/2.

MythX is a commercial platform for analyzing smart contracts. It offers a free tier of access, and, via the `solfuzz` tool, this makes it possible to easily run assertion checking on contracts, with ease-of-use similar to Echidna. The primary differences to a user are that `solfuzz`/MythX does not work offline, and there is limit of 40 runs/month and 5 minutes of compute time for the free tier. The `solfuzz` tool runs a short-duration analysis, that in our experiments took about two minutes, with an upper bound of 5 minutes, that is available for non-paying users (it can also run 20 minute or 45 minute analyses for users with paid MythX accounts). MythX analyzes the submitted contracts using the Mythril symbolic execution tool [5] and the Harvey fuzzer [16]. Harvey is a state-of-the-art closed-source tool, with a research paper describing its design and implementation in ICSE 2020 [16].

To compare MythX and Echidna, we first analyzed the contracts in the VeriSmart benchmark [1] and identified all contracts such that 1) both tools ran on the contract[1] and 2) neither tool reported any issues with the contract. This left us with 12 clean contracts, over which to compare the tools' ability to explore behavior. We inserted `assert(false)` statements into each of these contracts, after every statement, resulting in 459 contracts representing reachability targets. We discarded 44 of these, as the assert was unconditionally executed in the contract's constructor, so no behavior exploration was required to reach it.

We then ran `solfuzz`'s quick check and Echidna with a two-minute timeout on a randomly selected set of 40 targets. Echidna was able to produce a transaction sequence reaching the assert for 19 of the 40 targets, and `solfuzz`/MythX generated a reaching sequence for 15 of the 40, all of which were also reached by Echidna. While the time to reach the assertion was usually close to two minutes with `solfuzz`, Echidna needed a maximum of only 52 seconds to hit the hardest target; the mean time required was 13.9 seconds, and the median time was only 6.9 seconds.

We manually examined the targets not deteced by either tool, and believe them all to represent unreachable targets, usually due to being inserted after an unavoidable `return` statement, or being inserted in the `SafeMath` contract, which redefines `assert`. Of the reachable targets, Echidna was able to produce sequences for 100%, and `solfuzz`/MythX for 78.9%. For Echidna, we repeated each experiment 10 more times, and Echidna always reached each target. Due to the limit on MythX runs, even under a developer license (200 executions/month), we were unable to statistically determine the stability of its results to the same degree, but can confirm that for two of the targets, a second run succeeded, and for two of the targets three additional runs still failed to reach the assert. Running `solfuzz` with the `--mode standard` argument (not available to free accounts) did detect all four, but it required at least 15 minutes of analysis time in each case.

Figure 1 shows the key part of the code for one of the four targets Echidna, but not `solfuzz`/MythX (even with additional runs), was able to reach within 2 minutes. The assert can only be executed when a call has been made to the `approve` function, allowing the

[1] MythX/`solfuzz` failed to compile some contracts for unknown reasons, and timed out without producing results on others, and Echidna does not currently easily support contracts with constructors that require arguments.

```
if (balances[_from] >= _amount
  && allowed[_from][msg.sender] >= _amount
  && _amount > 0
  && balances[_to] + _amount > balances[_to]){
    balances[_from] -= _amount;
    allowed[_from][msg.sender] -= _amount;

    assert(false);
```

Figure 1: Code for a difficult reachablility target.

sender of the `transferFrom` call to send an amount greater than or equal to `_amount`, and when the contract from which transfer is to be made has a token balance greater than `_amount`. Generating a sequence with the proper set of functions called and the proper relationships between variables is a difficult problem, but Echidna's heuristic use of small numeric values in arguments and heuristic repetiton of addresses in arguments and as message senders is able to navigate the set of constraints easily. A similar set of constraints over allowances and/or senders and function arguments is involved in two of the other four targets where Echidna performs better.

5 RELATED WORK

6 CONCLUSION

REFERENCES

[1] VeriSmart benchmark. https://github.com/kupl/VeriSmart-benchmarks.

[2] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts SoK. In *International Conference on Principles of Security and Trust*, pages 164–186, 2017.

[3] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[4] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.

[5] ConsenSys. Mythril: a security analysis tool for ethereum smart contracts. https://github.com/ConsenSys/mythril-classic, 2017.

[6] Consensys Diligence. https://mythx.io/.

[7] 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.

[8] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 25–28, New York, NY, USA, 2018. ACM.

[9] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, 2018.

[10] Bernhard Mueller. https://github.com/b-mueller/solfuzz.

[11] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[12] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VeriSmart: A highly precise safety verifier for ethereum smart contracts. In *IEEE Symposium on Security & Privacy*, 2020.

[13] Trail of Bits. Echidna: Ethereum fuzz testing framework. https://github.com/trailofbits/echidna, 2018.

[14] Trail of Bits. Trail of bits security reviews. https://github.com/trailofbits/publications#security-reviews, 2019.

[15] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf, 2014.

[16] Valentin Wüstholz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *International Conference on Software Engineering*, 2020.