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, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

2 DESIGN AND ARCHITECTURE

3 USAGE

4 EXPERIMENTAL EVALUATION

In order to evaluate Echidna, we compared its performance to the MythX platform [3], accessed via the `solfuzz` [5] 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 [6]. 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 [4], 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 [2] and the Harvey fuzzer [7]. Harvey is a state-of-the-art closed-source tool, with a research paper describing its design and implementation in ICSE 2020 [7].

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

[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.

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

```
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.

sequence for 15 of the 40, all of which were also reached by Echidna. 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%. Due to the limit on MythX runs, we were unable to statistically determine the stability of its results. For Echidna, we repeated each experiment 30 times, and Echidna always reached each target.

Figure 1 shows the key part of the code for one of the four targets Echidna, but not `solfuzz`/MythX, was able to reach. The assert can only be executed when a call has been made to the `approve` function, allowing the 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] ConsenSys. Mythril: a security analysis tool for ethereum smart contracts. https://github.com/ConsenSys/mythril-classic, 2017.
[3] Consensys Diligence. https://mythx.io/.
[4] 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.
[5] Bernhard Mueller. https://github.com/b-mueller/solfuzz.
[6] 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.
[7] Valentin Wüstholz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *International Conference on Software Engineering*, 2020.