Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts

Gustavo Grieco
Trail of Bits

Will Song
Trail of Bits

Artur Cygan
Trail of Bits

Josselin Feist
Trail of Bits

Alex Groce
Northern Arizona University

ABSTRACT

Ethereum smart contracts—autonomous programs that run on a blockchain—often control transactions of financial and intellectual property. Because of the critical role they play, smart contracts need complete, comprehensive, and effective test generation. This paper introduces an open-source smart contract fuzzer called Echidna that makes it easy to automatically generate tests to detect violations in assertions and custom properties. Echidna is easy to install and does not require a complex configuration or deployment of contracts to a local blockchain. It offers responsive feedback, captures many property violations, and its default settings are calibrated based on experimental data. To date, Echidna has been used in more than 10 large paid security audits, and feedback from those audits has driven the features and user experience of Echidna, both in terms of practical usability (e.g., smart contract frameworks like Truffle and Embark) and test generation strategies. Echidna aims to be good at finding real bugs in smart contracts, with minimal user effort and maximal speed.

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, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3395363.3404366

1 INTRODUCTION

Smart contracts for the Ethereum blockchain [5], usually written in the Solidity language [25], facilitate and verify high-value financial transactions, as well as track physical goods and intellectual property. Thus, it is essential that these programs be correct and secure, which is *not* always the case [4]. Recent work surveying and categorizing flaws in critical contracts [11] established that fuzzing using custom user-defined properties might detect up to 63% of the most severe and exploitable flaws in contracts. This suggests an important need for high-quality, easy-to-use fuzzing for smart contract developers and security auditors. Echidna [23] is an open-source Ethereum smart contract fuzzer. Rather than relying on a fixed set of pre-defined bug oracles to detect vulnerabilities during fuzzing campaigns, Echidna supports three types of properties: (1) user-defined properties (for property-based testing [7]), (2) assertion checking, and (3) gas use estimation. Currently, Echidna can test both Solidity and Vyper smart contracts, and supports most contract development frameworks, including Truffle and Embark. Echidna has been used by Trail of Bits for numerous code audits [11, 24]. The use of Echidna in internal audits is a key driver in three primary design goals for Echidna. Echidna must (1) be easy to use and configure; (2) produce good coverage of the contract or blockchain state; and (3) be *fast* and produce results quickly.

The third design goal is essential for supporting the first two design goals. Tools that are easy to run and produce quick results are more likely to be integrated by engineers during the development process. This is why most property-based testing tools have a small default run-time or number of tests. Speed also makes use of a tool in continuous integration (CI) (e.g., https://crytic.io) more practical. Finally, a fast fuzzer is more amenable to experimental techniques like mutation testing [21] or using a large set of benchmark contracts. The size of the statistical basis for decision-making and parameter choices explored is directly limited by the speed of the tool. Of course, Echidna supports lengthy testing campaigns as well: there is no upper bound on how long Echidna can run, and with coverage-based feedback there is a long-term improvement in test generation quality. Nonetheless, the goal of Echidna is to reveal issues to the user in less than 5 minutes.

Fuzzing smart contracts introduces some challenges that are unusual for fuzzer development. First, a large amount of engineering effort is required to represent the semantics of blockchain execution; this is a different challenge than executing instrumented native binaries. Second, since Ethereum smart contracts compute using transactions rather than arbitrary byte buffers, the core problem is one of transaction sequence generation, more akin to the problem of unit test generation [20] than traditional fuzzing. This makes it important to choose parameters such as the length of sequences [3] that are not normally as important in fuzzing or in single-value-generation as in Haskell's QuickCheck [7]. Finally, finding smart contract inputs that cause pathological execution times is not an exotic, unusual concern, as in traditional fuzzing [17]. Execution on the Ethereum blockchain requires use of gas, which has a price in cryptocurrency. Inefficiency can be costly, and malicious inputs can lock a contract by making all transactions require more gas than a transaction is allowed to use. Therefore producing quantitative output of maximum gas usage is an important fuzzer feature, alongside more traditional correctness checks. Echidna incorporates a worst-case gas estimator into a general-purpose fuzzer, rather than forcing users to add a special-purpose gas-estimation tool [2, 18] to their workflow.

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07.
https://doi.org/10.1145/3395363.3404366

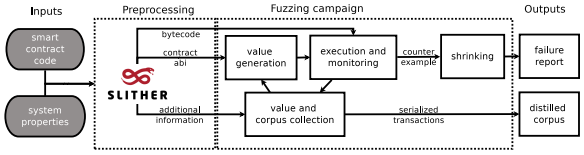


Figure 1: The Echidna architecture.

2 ARCHITECTURE AND DESIGN

2.1 Echidna Architecture

Figure 1 shows the Echidna architecture divided into two steps: pre-processing and the fuzzing campaign. Our tool starts with a set of provided contracts, plus properties integrated into one of the contracts. As a first step, Echidna leverages Slither [10], our smart contract static analysis framework, to compile the contracts and analyze them to identify useful constants and functions that handle Ether (ETH) directly. In the second step, the fuzzing campaign starts. This iterative process generates random transactions using the application binary interface (ABI) provided by the contract, important constants defined in the contract, and any previously collected sets of transactions from the corpus. When a property violation is detected, a counterexample is automatically minimized to report the smallest and simplest sequence of transactions that triggers the failure. Optionally, Echidna can output a set of transactions that maximize coverage over all the contracts.

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 (that checks detection of seeded faults, not just proper execution) to ensure that existing features are not degraded by improvements, and uses Haskell language features to maximize abstraction and applicability of code to new testing approaches.

2.2.1 Tuning Parameters. Echidna provides a large number of configurable parameters that control various aspects of testing. There are currently more than 30 settings, controlled by providing Echidna with a YAML configuration file. However, to avoid overwhelming users with complexity and to make the out-of-the-box experience as smooth as possible, these settings are assigned carefully chosen default values. Default settings with a significant impact on test generation are occasionally re-checked via mutation testing [12] or benchmark examples to maintain acceptable performance. This, like other maintenance, is required because other functionality changes may impact defaults. For example, changes in which functions are called (e.g., removing view/pure functions with no assertions) may necessitate using a different default sequence length. Parameter tuning can produce some surprises with major impact on users: e.g., the dictionary of mined constants was initially only used infrequently in transaction generation, but we found that mean coverage on benchmarks could be improved significantly by using constants a full 40% of the time.

3 USAGE

Before starting Echidna, the smart contract to test should have either explicit `echidna_` properties (public methods that return a Boolean have no arguments) or use Solidity's `assert` to express properties. For instance, Figure 2a shows a contract with a property that tests a simple invariant. After defining the properties to test, running Echidna is often as simple as installing it or using the provided Docker image and then typing:

```
$ echidna-test Contract.sol --contract TEST
```

An optional YAML configuration file overriding default settings can be provided using the `-config` option. Additionally, if a path to a directory is used instead of a file, Echidna will auto-detect the framework used (e.g. Truffle) and start the fuzzing campaign.

By default, Echidna uses a dashboard output similar to AFL's as shown in Figure 2b. However, a command line option or a config file can change this to output plaintext or JSON. The config file also controls various properties of test generation, such as the maximum length of generated transaction sequences, the frequency with which mined constants are used, whether coverage driven feedback is applied, whether maximum gas usage is computed, and any functions to blacklist from the fuzzing campaign.

4 EXPERIMENTAL EVALUATION

4.1 Setup

We compared Echidna's performance to the MythX platform [9], accessed via the `solfuzz` [19] interface, on a set of reachability targets. Our experiments are produced by insertion of `assert(false)` statements, on a set of benchmark contracts [1] produced for the VeriSmart safety-checker [22]. To our knowledge, MythX is the only comparable fuzzer that 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 [16], which does not support arbitrary assertions in Solidity code, is difficult to do objectively, as any differences are likely to be due to specification semantics, not exploration capability. MythX is a commercial SaaS platform for analyzing smart contracts. It offers a free tier of access (limited to 5 runs/month, however) and can easily run assertion checking on contracts via `solfuzz`, which provides an interface similiar to Echidna's. MythX analyzes the submitted contracts using the Mythril symbolic execution tool [8] and the Harvey fuzzer [26]. Harvey is a state-of-the-art closed-source tool, with a research paper describing its design and implementation in ICSE 2020 [26]. We also attempted to compare to the ChainFuzz [6] tool; unfortunately, it is not maintained, and failed to analyze contracts, producing an error reported in a GitHub issue submitted in April of 2019 (https://github.com/ChainSecurity/ChainFuzz/issues/2).

4.2 Datasets

VeriSmart. 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 and 2) neither tool reported any issues with the contract. This left us with 12 clean contracts 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.

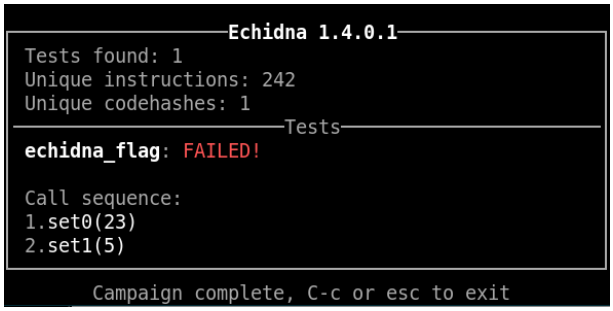
```
contract TEST {
  bool flag0; bool flag1;

  function set0(int val) public returns (bool) {
    if (val % 100 == 23)         { flag0 = true; } }

  function set1(int val) public returns (bool) {
    if (val % 10  == 5 && flag0) { flag1 = true; } }

  function echidna_flag() public returns (bool) {
    return(!flag1); }
}
```

(a) A contract with an echidna property.



(b) A screenshot of the UI with the result of a fuzzing campaign

Figure 2

Tether. For a larger, more realistic example, we modified the actual blockchain code for the TetherToken contract[1], and again inserted `assert(false)` targets to investigate reachability of the code. Tether is one of the most famous "stablecoins", a cryptocurrency pegged to a real-world currency, in this case the US dollar, and has a market cap of approximately 6 billion dollars. The contract has been involved in more than 23 million blockchain transactions.

4.3 Results

We then ran `solfuzz`'s default quick check and Echidna with a 2-minute timeout on 40 randomly selected 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 2 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 detected 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 (500 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 4 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. 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 repetition 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.

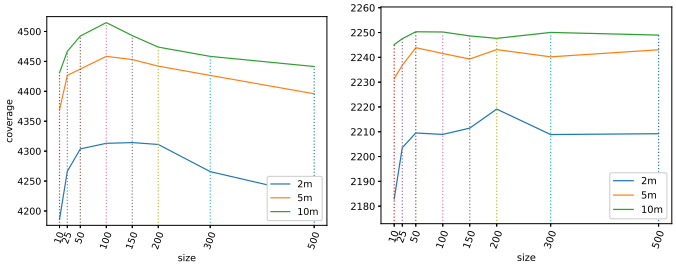
When using the Tether contract, we again randomly selected 40 targets, and ran two minutes of testing on each with `solfuzz` and Echidna. Echidna was able to reach 28 of the 40 targets, with mean and median runtimes of 24 and 15 seconds, respectively. The longest run required 103 seconds. On the other hand, solfuzz/MythX was unable to reach any of the targets using the default search. MythX/-solfuzz was able to all detect the targets Echidna detected using the standard search, and detected one additional target. The mean time required for detection, however, was almost 16 minutes. The additional target reached by `solfuzz` involves adding an address to a blacklist, then destroying the funds of that address. Because an address can also be removed from a blacklist, there is no simple coverage-feedback to encourage avoiding this, and there are many functions to test, Echidna has trouble generating such a sequence. However, using a prototype of a swarm testing [13] mode not yet included in the public release of Echidna, but briefly discussed in the conclusion below, we were able to produce such a sequence in less than five minutes. Even without swarm testing, we were able to detect the problem in between 10 and 12 minutes, using a branch (to be merged in the near future) that incorporates more information from Slither, and uses some novel mutation strategies. Of the 11 targets hit by neither tool, we manually determined that all but two are clearly actually unreachable.

As a separate set of experiments, we measured the average coverage obtained on the VeriSmart and Tether token contracts, with various settings for the length of transaction sequences, ranging from very short (length 10) to very long (length 500) for runs of 2, 5, and 10 minutes each. Figures 3a and 3b show that the current default value used by Echidna (100) is a reasonable compromise to maximize coverage in short fuzzing campaigns. Each run was repeated 10 times to reduce the variability of such short campaigns.

5 RELATED WORK

Echidna inherits concepts from property-based fuzzing, first popularized by the QuickCheck tool [7] and from coverage-driven

[1] https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code



(a) TetherToken (b) VeriSmart (avg)

Figure 3: Coverage obtained given short runs (2, 5 and 10 minutes) with different transaction sequence lengths.

```
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 4: Code for a difficult reachabilility target.

fuzzing, perhaps best known via the American Fuzzy Lop tool [27]. Other fuzzers for Ethereum smart contracts include Harvey [26], ContractFuzzer [16], and ChainFuzz [6]. We were unable to get ContractFuzzer to produce useful output within a four hour timeout, and ChainFuzz no longer appears to work. Harvey is closed-source, but is usable via the MythX [9] CI platform and the `solfuzz` [19] tool. Echidna uses information from the Slither static analysis tool [10] to improve the creation of Ethereum transactions.

6 CONCLUSIONS AND FUTURE WORK

Echidna is an effective, easy-to-use, and *fast* fuzzer for Ethereum blockchain smart contracts. Echidna provides a potent out-of-the-box fuzzing experience with little setup or preparation, but allows for considerable customization. Echidna supports assertion checking, custom property-checking, and estimation of maximum gas usage—a core feature set based on experience with security audits of contracts. The default test generation parameters of Echidna have been calibrated using real-world experience in commercial audits and via benchmark experiments and mutation analysis. In our experiments, Echidna outperformed a comparable fuzzer using sophisticated techniques: Echidna detected, in less than 2 minutes, many reachability targets that required 15 or more minutes with `solfuzz`, on both benchmark contracts and the real-world Tether token. Echidna is under heavy active development. Recently added or in-progress features include gas estimation, test corpus collection, integration of Slither static analysis information, and improved mutation for feedback-driven fuzzing. One future work will add a driver mode, similar to the swarm tool [14] for the SPIN model checker [15], to make better use of configuration diversity, including swarm testing [13], in order to fully exploit multicore machines. In particular, this mode will enable Echidna to produce even more accurate maximum gas usage estimates.

REFERENCES

[1] VeriSmart benchmark. https://github.com/kupl/VeriSmart-benchmarks.
[2] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. Gasol: Gas analysis and optimization for ethereum smart contracts, 2019.
[3] James H. Andrews, Alex Groce, Melissa Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *Automated Software Engineering*, pages 19–28, 2008.
[4] 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.
[5] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.
[6] Chain Security. https://github.com/ChainSecurity/ChainFuzz.
[7] 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.
[8] ConsenSys. Mythril: a security analysis tool for ethereum smart contracts. https://github.com/ConsenSys/mythril-classic, 2017.
[9] Consensys Diligence. https://mythx.io/.
[10] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019.
[11] 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.
[12] 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.
[13] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis*, pages 78–88, 2012.
[14] Gerard Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011.
[15] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
[16] 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.
[17] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.
[18] Fuchen Ma, Ying Fu, Meng Ren, Wanting Sun, Zhe Liu, Yu Jiang, Jun Sun, and Jiaguang Sun. Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability, 2019.
[19] Bernhard Mueller. https://github.com/b-mueller/solfuzz.
[20] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.
[21] 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.
[22] 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.
[23] Trail of Bits. Echidna: Ethereum fuzz testing framework. https://github.com/trailofbits/echidna, 2018.
[24] Trail of Bits. Trail of bits security reviews. https://github.com/trailofbits/publications#security-reviews, 2019.
[25] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf, 2014.
[26] Valentin Wüstholz and Maria Christakis. Targeted greybox fuzzing with static lookahead analysis. In *International Conference on Software Engineering*, 2020.
[27] Michal Zalewski. american fuzzy lop (2.35b). http://lcamtuf.coredump.cx/afl/. Accessed December 20, 2016.