echidna-parade: A Tool for Diverse Multicore Smart Contract Fuzzing

Alex Groce
Northern Arizona University
United States

Gustavo Grieco
Trail of Bits
United States

ABSTRACT

Echidna is a widely used fuzzer for Etherum blockchain smart contracts that generates *transaction sequences* of calls to smart contracts. While Echidna is an essentially single-threaded tool, it is possible for multiple Echidna processes to communicate by use of a shared transaction sequence corpus. Echidna provides a very large variety of configuration options, since each smart contract may be best-tested by a non-default configuration, and different faults or coverage targets within a single contract may also have differing ideal configurations. This paper presents *echidna-parade*, a tool that provides pushbutton multicore fuzzing using Echidna as an underlying fuzzing engine, and automatically provides sophisticated diversification of configurations. Even without using multiple cores, echidna-parade can improve the effectiveness of fuzzing with Echidna, due to the advantages provided by multiple types of test configuration diversity. Using *echidna-parade* with multiple cores can produce significantly better results than Echidna, in less time.

CCS CONCEPTS

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

KEYWORDS

fuzzing, smart contracts, test diversity, swarm testing, test length

ACM Reference Format:
Alex Groce and Gustavo Grieco. 2021. *echidna-parade*: A Tool for Diverse Multicore Smart Contract Fuzzing. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

An echidna is a spiny monotreme; Echidna is a widely used open source fuzzer for Ethereum smart contracts [6, 12]. The collective noun for echidnae is *"parade"*; echidna-parade is a tool for configuring and running multiple Echidna instances to improve the effectiveness of smart contract fuzzing.

Smart contracts for the Ethereum blockchain [4], most often written in Solidity [13], a JavaScript-like language, support high-value financial transactions, as well as tracking valuable IP and even physical goods. It is essential that autonomous financial programs be reliable and protected against attack. Unfortunately, smart contracts are often *neither* correct nor secure [3]. A survey categorizing flaws in critical contracts [7] estimated that fuzzing, using custom user-defined properties, might detect more than 60% of the most severe, exploitable flaws in contracts, and that many of these cannot easily be detected using purely static analysis. Highly effective fuzzing is therefore essential to smart contract developers and security auditors, and the Echidna tool is used by both contract developers and auditors at Trail of Bits.

Echidna is essentially a single-threaded tool that does not make effective use of mulitple cores. However, multiple Echidna independent Echidna processes can be run at the same time. An Echidna process will, upon termination, produce a corpus of transaction sequences needed to cover all reached contract locations and transaction dispositions (whether the transaction succeeded or caused a revert of the EVM) as a set of files, and an Echidna process can take as input a set of such transaction sequences to seed fuzzing.

The *echidna-parade* tool is an open source utility that orchestrates multiple Echidna processes using this mechanism, to enable both multicore fuzzing and more effective single-core fuzzing, by diversifying the configuration of Echidna, in order to cover hard-to-reach code, and discover subtle flaws. It is available via pypi (`pip install echidna-parade`) or on GitHub (https://github.com/agroce/echidna-parade).

2 BASIC USAGE

Using *echidna-parade* is intended to be a push-button process, requiring no additional expertise for users familiar with the Echidna tool. For instance, if a user used the command line:

```
> echidna-test contract.sol --config config.yaml --contract TEST
```

to test a smart contract, then testing the same contract with echidna-parade, using all available CPUs for one hour, would require only a slight modification:

```
> echidna-parade contract.sol --config config.yaml --contract TEST
```

In both cases, a contract called `TEST` in a file called `contract.sol` will be tested, based on a configuration file, `config.yaml`. The configuration file specifices the details of the fuzzing campaign to be run, for Echidna, and specifies a baseline for generating a set of varying configurations, for *echidna-parade*. The other major obvious difference is that Echidna will run using a single core, while, if called this way, without an `--ncores` command line argument, *echidna-parade* will make use of as many cores as the tool finds available. Figure 1 shows output from a typical *echidna-parade* run from our experiments below.

Part of the need for *echidna-parade* arises from the complexities of the configuration file. Echidna currently supports 34 different configuration options in this file, over 20 of which have complex,

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

```
Starting echidna-parade with
config=Config(files=['/Users/adg326/dss/crytic-export/flattening/DogTest.sol'],
  name='2hour.8.parade.experiment.29', resume=None, contract='DogTest',
  config=<_io.TextIOWrapper name='plain.yaml' mode='r' encoding='UTF-8'>, bases=None, ncores=8,
  corpus_dir='/Users/adg326/dss/corpus.2hour.8.parade.experiment.29', timeout=7200,
  gen_time=300, initial_time=300, seed=None, minseqLen=10,
  maxseqLen=300, PdefaultLen=0.5, PdefaultDict=0.5,
  prob=0.5, always=[])
Results will be written to: /Users/adg326/dss/2hour.8.parade.experiment.29
Identified 29 public and external functions: Dog.rely(address), Dog.deny(address), Dog.file(bytes32,address), Dog.file(bytes32,uint256),
  Dog.file(bytes32,bytes32,uint256), Dog.file(bytes32,bytes32,address), Dog.chop(bytes32), Dog.bark(bytes32,address,address), Dog.digs(bytes32,uint256),
  Dog.cage(), Vat.rely(address), Vat.deny(address), Vat.hope(address), Vat.nope(address), Vat.init(bytes32), Vat.file(bytes32,uint256), Vat.file(bytes32,bytes32,uint256),
  Vat.cage(), Vat.slip(bytes32,address,int256), Vat.flux(bytes32,address,address,uint256), Vat.move(address,address,uint256),
  Vat.frob(bytes32,address,address,address,int256,int256), Vat.fork(bytes32,address,address,int256,int256), Vat.grab(bytes32,address,address,address,int256,int256),
  Vat.heal(uint256), Vat.suck(address,address,uint256), Vat.fold(bytes32,address,int256),VowMock.fess(uint256), ClipperMock.kick(uint256,uint256,address,address)

RUNNING INITIAL CORPUS GENERATION
- LAUNCHING echidna-test in 2hour.8.parade.experiment.29/initial blacklisting [  ] with seqLen 100 dictFr
eq 0.4 and mutConsts  [1, 1, 1, 1]

SWARM GENERATION #1: ELAPSED TIME 303.87 SECONDS / 7200
- LAUNCHING echidna-test in 2hour.8.parade.experiment.29/gen.1.0 blacklisting [ Dog.rely(address), Dog.deny(address), Dog.file(bytes32,address), Dog.file(bytes32,uint256),
  Dog.file(bytes32,bytes32,address), Vat.rely(address), Vat.deny(address), Vat.nope(address), Vat.file(bytes32,uint256), Vat.slip(bytes32,address,int256),
  Vat.move(address,address,uint256), Vat.frob(bytes32,address,address,address,int256,int256), Vat.heal(uint256) ] with seqLen 100 dictFreq 0.4 and mutConsts  [0, 3, 1000, 3]
- LAUNCHING echidna-test in 2hour.8.parade.experiment.29/gen.1.1 blacklisting [ Dog.deny(address), Dog.file(bytes32,address), Dog.file(bytes32,bytes32,uint256),
  Dog.file(bytes32,bytes32,address), Dog.cage(), Vat.hope(address), Vat.file(bytes32,bytes32,uint256), Vat.flux(bytes32,address,address,uint256),
  Vat.frob(bytes32,address,address,address,int256,int256), Vat.fork(bytes32,address,address,int256,int256), Vat.grab(bytes32,address,address,address,int256,int256),
  Vat.suck(address,address,uint256), Vat.fold(bytes32,address,int256), VowMock.fess(uint256), ClipperMock.kick(uint256,uint256,address,address) ] with seqLen 100
  dictFreq 0.88 and mutConsts  [2, 0, 2, 2000]
...
COLLECTING NEW COVERAGE: 2hour.8.parade.experiment.29/gen.1.0/corpus/coverage/-7507126444194881135.txt
COLLECTING NEW COVERAGE: 2hour.8.parade.experiment.29/gen.1.0/corpus/coverage/-6907082692337979773.txt
COLLECTING NEW COVERAGE: 2hour.8.parade.experiment.29/gen.1.2/corpus/coverage/-4795647765542071453.txt
COLLECTING NEW COVERAGE: 2hour.8.parade.experiment.29/gen.1.4/corpus/coverage/3547233554766032648.txt
COLLECTING NEW COVERAGE: 2hour.8.parade.experiment.29/gen.1.4/corpus/coverage/-7663735074641916226.txt
...
SWARM GENERATION #2: ELAPSED TIME 612.63 SECONDS / 7200
- LAUNCHING echidna-test in 2hour.8.parade.experiment.29/gen.2.0
...
DONE!
RUNNING FINAL COVERAGE PASS...
- LAUNCHING echidna-test in 2hour.8.parade.experiment.29/coverage blacklisting [  ] with seqLen 100 dictFreq 0.4 and mutConsts  [1, 1, 1, 1]
COVERAGE PASS TOOK 62.12 SECONDS
NO FAILURES
```

Figure 1: Running *echidna-parade* on the DSS Code

non-boolean values. Some of these parameters relate to the details of the contract under test, e.g. special Ethereum addresses with meaning to the contract, or simply determine basic properties of a run, such as the time allowed for testing, format of the output, or whether to try to find transactions with high gas prices or check assertions in addition to user-defined properties. However, there are configurations for transaction sequence maximum length, which functions to test, and control of low-level test generation strategies that have a large impact on test effectiveness, and are either hard to tune or, for that matter, impossible to "tune" in that different values are essential to covering different aspects of the code or exposing different bugs.

The *echidna-parade* tool stores all results from all the separate Echidna runs it generates in separate directories, so that users can inspect failed properties and violated assertion. The tool collects all failures and provides a summary at the end of the run, including information on where exact failure traces can be found. Users can also use these detailed results to determine if increases in coverage or bugs were correlated with inclusion or omission of certain functions.

3 ARCHITECTURE AND DIVERSIFICATION STRATEGIES

The basic architecture of *echidna-parade* is simple. It uses the Slither static analysis tool to scan the contract under test and extract needed information, then examines any custom Echidna configuration options provided by the user (in the form of a YAML file). After this initial scan, it runs an initial run of Echidna using default or user-configuration parameters, to form a starting corpus of transaction sequences (API call tests, essentially). After this initial run, each *generation* involves:

(1) Generating a set of novel configuations (YAML files) up to the number of cores the tool is allowed to use.
(2) Spawning Echidna processess for each of these configurations. Each process is seeded with the set of all coverage-enhancing transaction sequences found by any run thus far.
(3) Collecting any transaction sequences from these runs that produce new coverage, and adding them to the corpus.
(4) Reporting new corpus sequences discovered and/or new property failures to the user.

The core non-bookkeeping element of the tool is the generation of novel configurations that provide search diversity and improve testing. Understanding these sources of diversity is key to understanding the rationale for the *echidna-parade* tool.

3.1 Swarm Testing (API Call Omission)

Swarm testing [9] is a method for improving automated test generation that relies on identifying *features* of tests, and disabling some features in each test. For instance, if features are API calls,

	Echidna (5 hours)	Parade (1 core, 5 hours)	Parade (8 cores, 1 hour)	Parade (8 cores, 2 hours)
Mean Fully Covered	88.63	90.33	91.33	94.3
Median Fully Covered	86.0	86.0	86.0	98.0
Std. Dev Fully Covered	7.17	7.36	7.78	7.26
Mean Non-Revert Covered	136.2	138.23	139.7	143.37
Median Non-Revert Covered	133.0	133.0	133.0	148.0
Std. Dev Non-Revert Covered	9.03	9.36	9.95	9.14

Table 1: DSS Experiment Results

and we are testing a stack with `push`, `pop`, `top`, and `clear` calls, a non-swarm random test of any significant length will normally contain multiple calls to all of these functions. In swarm testing, for each test some of the calls (with probability usually equal to 0.5 for each call) will be disabled, but different calls will be disable for each generated test. This produces less variance between calls within a test, but much more variance *between tests*. Practically, in the stack example, it will enable the size of the stack to grow much larger than it ever would have any chance of doing in non-swarm testing, due to some tests omitting `pop` and/or `clear` calls. Swarm testing is widely used in compiler testing [5, 11] and is a core element of the testing for FoundationDB, the back-end database for Apple and Snowflake cloud services [14].

echidna-parade uses the Slither static analysis tool to extract the set of public functions from tested smart contracts, and automatically configures Echidna to omit some of these functions (about 50%, by default) during each run. We believe this provides the most important form of variation in fuzzing. In particular, given that some bugs and/or coverage targets are *triggered* by some function calls but *suppressed* by other calls, it is important to perform testing with as many different call sets as possible [8].

3.2 Test Length Variance

Another important form of variation is the length of test sequences [1, 2]. It is known that there is no single best choice for the length of API call sequences in test generation; different SUTs and even different bugs and/or coverage targets in the same SUT may "prefer" different test lengths. For example, a line of code relating to a resource limit (a check for an array being full, for example) may require a long test to have any chance of execution. Another line of code may only execute if another value has not been initialized by a different API function that can be called. Running a larger number of shorter tests will enable executing this line more often, since once the code is initialized, the line can no longer be covered in a test. *echidna-parade* therefore varies the maximum sequence length for each Echidna run as well, in a user-configurable way (with a bias towards the default, tuned value).

3.3 Mutation and Search Variance

Finally, as Holzmann et al. showed, in hard search problems in model checking, it is extremely useful to simply vary the underlying search methods, given the lack of an optimal solution [10]. The equivalent for Echidna is to vary the sequence mutation strategies used in the coverage-driven GA search and the frequency with which dictionary constants mined from source code are used. *echidna-parade* therefore also varies these parameters over the full set of valid values, but with some bias towards the default values.

3.4 User-Controlled Variance

In addition to these automatic variations, a user can also provide a set of "overlay" YAML Echidna configuration files to *echidna-parade*, which will select one of these at random to use in each configuration, replacing that configuration's choices for any settings included in the chosen YAML file.

4 EXPERIMENTAL EVALUATION

We compared Echidna without any variance, using a fixed configuration, to a 5 hour single-core *echidna-parade* run using the default variance, a 1 hour *echidna-parade* run using 8 cores, and a 2 hour *echidna-parade* run using 8 cores. For our evaluation, we used the Multi Collateral Dai contract code (https://github.com/makerdao/dss), developed as part of the MakerDAO decentralized autonomous organization's Dai stablecoin (market cap as we write of approximately $3.7 billion). The code is a variant of the repo code used during a security audit [7]. Results in Table 1 show results for 30 runs of each of these approaches. The first set of results compares fully-covered lines of code. A line of code in a solidity contract is fully-covered if it has been executed both in a context where the function in which it resides runs to completion *and* a context where the function *reverts*. In the Ethereum blockchain, a revert causes a call to a contract to terminate and roll back any state changes. Testing revert behavior can be essential for detecting certain bugs. The second comparison considers only non-reverting coverage; while covering reverts is important, it is usually most important for testing to run all code in a context where the effects propagate to produce state changes. Despite the large overhead of continually starting up a new echidna process, writing new coverage results to disk after each run, and re-executing all corpus transaction sequences, even a single core parade produces better results over 5 hours. Making use of multiple cores, which is not supported inside Echidna itself, a 1 or 2 hour *echidna-parade* run can produce much better results. The 2 hour, 8-core parade run produces statistically significantly better results for all measures, by the Mann-Whitney U test, with p-values below 0.001. All other results, due to the notably high variance of testing, are not significant ($0.11 \leq p \leq 0.24$). Some coverage targets were *only* reached during a parade run; we are unsure if these can be covered by non-diversified Echidna, in any reasonable amount of time. Clearly for time efficient testing of the contract, use of *echidna-parade* is required.

5 RELATED WORK

6 CONCLUSION

REFERENCES

[1] 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.

[2] Andrea Arcuri. A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage. *IEEE Trans. Software Eng.*, 38(3):497–519, 2012.

[3] 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.

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

[5] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using clp (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 482–493. IEEE, 2015.

[6] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: Effective, usable, and fast fuzzing for smart contracts. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 557–560, New York, NY, USA, 2020.

[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, Chaoqiang Zhang, Mohammad Amin Alipour, Eric Eide, Yang Chen, and John Regehr. Help, help, I'm being suppressed! the significance of suppressors in software testing. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 390–399. IEEE, 2013.

[9] 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.

[10] Gerard Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011.

[11] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014.

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

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

[14] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J Beamon, Rusty Sears, John Leach, et al. Foundationdb: A distributed unbundled transactional key value store. In *ACM SIGMOD*, 2021.

A PROPOSED DEMONSTRATION