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 Ethereum Virtual Machine (EVM) compatible 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, 5 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]. 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 [14], 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 internally auditors at Trail of Bits. Echidna is, to our understanding, the most popular and best supported fuzzer for Ethereum smart contracts, with almost an order of magnitude more GitHub stars than any other open-source EVM fuzzer.

Echidna is essentially a single-threaded tool that does not make effective use of multiple 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). The tool is already being used internally in security audits by Trail of Bits.

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 specifies 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,

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

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, 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 configurations (YAML files) up to the number of cores the tool is allowed to use.
(2) Spawning Echidna processes 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 tool allows users to configure the time allotted to the initial scan and to each generation, and initialize testing from an existing corpus. A parade run can also be resumed, with the `--resume` option, taking up where the fuzzing left off, a feature that often proves useful in fuzzers such as afl and libFuzzer.

The core non-bookkeeping element of the tool is the construction of the 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, 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 [12] and is a core element of the testing for FoundationDB [16].

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 (the probability of omission is 50%, by default, but can be configured) 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].

Because users may know that *some* calls are essential to the functionality being testing, *echidna-parade* supports an argument, `--always`, specifying function signatures that should *never* be omitted from configurations.

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 Systems Under Test (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.

```
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 dictFreq 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

	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

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. This feature makes it possible for users, with some effort, to diversify *any* Echidna configuration option, with any desired probability distribution. In particular, we expect that some users will want to use this to, e.g., run some fuzzing in configurations where the sender of transactions is the contract owner/creator, and some in configurations where unauthorized users originate calls into the contract. These kinds of diversification are inherently contract-specific, unlike the generic forms of diversity *echidna-parade* automatically provides.

4 EXPERIMENTAL EVALUATION

We first compared the performance of Echidna and *echidna-parade* on the example contract included in the *echidna-parade* repo to demonstrate the value of swarm testing. We configured both tools to use 10 minutes of CPU time (so *echidna-parade* derived no advantage in testing cycles from using multiple cores, and in fact paid a substantial overhead due to having to restart the fuzzer multiple times). There are three detectable assertion violations in the example. One of the violations is trivial to detect, but detecting it will also abort the trace and prevent detecting the other issues. The other two violations require constructing very large arrays via repeated calls. We configured *echidna-parade* to always include the functions with the hard-to-find violations, and three functions known to be relevant to finding the vulnerabilities. User knowledge of such "key" functions is common in real analysis of contracts, but cannot be expressed to Echidna without using *echidna-parade*.

Over 10 runs, *echidna-parade* detected a mean of 2.44 of the violations, with a median of two detected issues. It detected the hardest-to-trigger violation in 7 of the runs. Echidna alone only detected a mean of 1.6 issues, with a median detection of 1.5 issues. Moreover, Echidna only detected the hardest-to-produce problem twice. The difference between the two approaches was statistically significant by Mann-Whitney U-test with $p \leq 0.005$.

For a more substantial, real-world 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 liquidation contracts from 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] https://tinyurl.com/dazdu66c. 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, ensuring guards on prohibited functionality are enforced. 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. In fact, even using *echidna-parade* there were coverage targets that were only covered once in total experiment runtime of over 36 CPU days! Clearly for effective and efficient testing of this contract, use of *echidna-parade* is required.

5 RELATED WORK

Smart contract fuzzing has been a popular topic in recent literature. To our knowledge, ContractFuzzer [11] was the first such tool described in the literature. Harvey [15], sFuzz [13], and other tools, in addition to Echidna [6], the basis for our work, have subsequently appeared, and more are in development.

The foundations for the approach to diversification taken by *echidna-parade* are three primary lines of research. The first of these is the idea of swarm testing [9], which aims to improve testing by omitting some features uniformly throughout a test, in part because a feature (e.g. API call) may *suppress* some behaviors [8]. Second, there is literature showing that a single test *length* may not be ideal for all SUTs or behaviors of an SUT [1, 2]. Finally, there is the swarm verification approach to model checking [10], a precursor to swarm testing, which diversifies search strategy parameters themselves. The only other tool we are aware of that adds swarm testing as a layer atop an existing fuzzer is the DeepState tool [5], which was a direct inspiration for *echidna-parade*.

6 CONCLUSION

Smart contract fuzzing is a challenging problem, a particularly high-stakes instance of the general API-sequence test generation problem. As our experiments above show, high stakes contracts can be hard to explore, even with multi-hour fuzzing runs. The *echidna-parade* tool addresses this problem in two ways. First, it adds easy multicore support to the essentially single-threaded Echidna smart contract fuzzer. Second, *echidna-parade* adds additional value by automatically diversifying the fuzzing performed by Echidna, which makes some bugs much easier to find. The tool is open source and is already being used in security audits at Trail of Bits.

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] Peter Goodman and Alex Groce. DeepState: Symbolic unit testing for C and C++. In *NDSS Workshop on Binary Analysis Research*, 2018.

[6] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: Effective, usable, and fast fuzzing for smart contracts. In *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] Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *International Conference on Automated Software Engineering*, pages 259–269, 2018.

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

[13] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. SFuzz: An efficient adaptive fuzzer for Solidity smart contracts. In *International Conference on Software Engineering*, page 778–788, 2020.

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

[15] Valentin Wüstholz and Maria Christakis. Harvey: A greybox fuzzer for smart contracts. In *Foundations of Software Engineering*, pages 1398–1409, 2020.

[16] 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

(1) First, briefly note the importance of Ethereum smart contracts, go to coindesk and etherscan pages to show the current ETH market cap, and then the massive number of transactions happening each day and associated ETH dollar value, which doesn't even cover token values where much of the action is.
(2) Next show the Echidna tool page, and introduce it is as a widely used smart contract fuzzer.
(3) Show a simple smart contract, such as the justlen.sol code, and then a complex one, such as part of the DSS code used in the evaluation. Discuss the basics of Solidity, run echidna on a trivial example from the Echidna distribution, to show what a property failure and transaction sequence looks like.
(4) Discuss the challenge of fuzzing complex contracts, just like any other hard fuzzing problem. Discuss that Echidna is single-core, but can communicate via corpuses, like afl but without mid-run reading of new corpus info at present. Discuss how many hours of Echidna will not hit all interesting targets in code like DSS. Briefly describe use in both contract development (show pages of real contracts with Echidna tests) and in audits (mention some real bugs found in public Trail of Bits audits using Echidna).
(5) Now show how to install echidna-parade, github repo.
(6) Describe how the tool has the same interface as echidna. Run on the simple example we used Echidna on.
(7) Look at directory contents generated by echidna-parade.
(8) Show the experimental results from the paper, discuss the advantages testing something like DSS.
(9) Step through the three sources of diversity, pointing to the run on the simple example to explain swarm omission of API calls, and the differing test lengths and parameter choices.
(10) Run on the justlen.sol example, which is quick enough to show during the demonstration. Note the experimental results for it.