DeepState: Unit Testing Unbound

Alex Groce
Northern Arizona University
United States

Peter Goodman
Gustavo Grieco
Trail of Bits
United States

Alan Cao
New York University
United States

ABSTRACT

Almost every software developer knows how to write a unit test; very few software developers know how to use fuzzing or symboilic execution tools. DeepState provides a way to write parameterized/generalized unit tests by adding data generation and nondeterministic choice constructs to a Google Test-like API. DeepState can then use modern fuzzers such as afl or libFuzzer, or the Manticore symbolic execution tool, to construct concrete tests. DeepState makes it easy to apply mulitple fuzzers, including in a cooperative ensemble, to a testing problem, and to use fuzzers for full-fledged property-based testing with complex input validity and correctness constraints. DeepState also adds swarm testing and smart test reduction and normalization capabilities to fuzzers without the need to modify the back-end tool.

CCS CONCEPTS

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

KEYWORDS

parameterized unit tests, fuzzing, symbolic execution, test reduction

ACM Reference Format:
Alex Groce, Peter Goodman, Gustavo Grieco, and Alan Cao. 2021. DeepState: Unit Testing Unbound. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Automated test generation using sophisticated methods, including fuzzers and symbolic execution, has the potential to greatly improve the reliability, safety, and security of software systems. Unfortunately, few real-world developers have any experience using such tools, and the widely differing ways test harnesses must be written [7] and tools must be run limit knowledge transfer from one tool to another. DeepState [6] provides a single front-end that allows a developer to *write a single parameterized unit test* [16] and then use a variety of fuzzers and/or symbolic execution tools to generate test cases in a single, shared, binary format. DeepState has been used in internal security audits at Trail of Bits, and to detect bugs in real-world code (https://github.com/Blosc/c-blosc2/issues/95, https://github.com/laurynas-biveinis/unodb, https://github.com/RoaringBitmap/CRoaring); DeepState is being used in various non-public testing efforts, as well.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

```
#include <deepstate/DeepState.hpp>
using namespace deepstate;
#include  "runlen.h"
// This can be much higher for fuzzing, but should be small for
// symbolic execution to scale
#define MAX_STR_LEN 6
TEST(Runlength, EncodeDecode) {
  char* original = DeepState_CStrUpToLen(MAX_STR_LEN,
                                          "abcdef0123456789");
  char* encoded = encode(original);
  ASSERT_LE(strlen(encoded), strlen(original)*2) <<
   "Encoding is > length*2!";
  char* roundtrip = decode(encoded);
  ASSERT_EQ(strncmp(roundtrip, original, MAX_STR_LEN), 0) <<
    "ORIGINAL: '" << original << "', ENCODED: '" << encoded <<
    "', ROUNDTRIP: '" << roundtrip << "'";
}
```

Figure 1: A Simple DeepState Test Harness

2 A SIMPLE EXAMPLE

As Figure 1, taken from an example in the DeepState repository, but omitting the actual (buggy) code for a runlength encoding shows, DeepState test harnesses look very much like the very widely used GoogleTest (https://github.com/google/googletest) framework. The great difference is the use of DeepState API calls to replace fixed, concrete values with tool-generated values, and the use of more general, property-based-testing style, specifications to account for the variability of test data. Here, the "work" is done by `DeepState_CStrUpToLen`, which takes a maximum string length and (optionally) an alphabet to use, and generates (and manages memory for) string inputs. DeepState similarly provides APIs for all common C/C++ base types, including generation of values in ranges (e.g., `DeepState_IntInRange`, `DeepState_DoubleInRange`, etc.). Once such a harness is written, test generation is easy. Simply compiling the test and linking with the DeepState binary, via a command such as `clang++ -o runlen Runlen.cpp -ldeepstate` allows a user to use symbolic execution via Manticore [14], the Eclipser fuzzer [3], afl in QEMU mode, or DeepState's built-in dumb (but very fast) fuzzer:

```
> deepstate-manticore ./runlen --output_test_dir manticore_symex
...
> deepstate-eclipser ./runlen --output_test_dir eclipser_fuzzing
...
> ./runlen --fuzz --output_test_dir bruteforce_fuzzing
...
```

Fuzzing with afl or another fuzzer requiring compile-time instrumentation is only slightly more difficult:

```
> deepstate-afl --compile_test ./Runlen.cpp --out_test_name runlen
...
> deepstate-afl ./runlen.afl --fuzzer_out --output_test_dir afl_fuzzing
```

```
OneOf(
      [&] {MakeNewPath(path);
        r = tfs_mkdir(sb, path);
        LOG(TRACE) << "tfs_mkdir(sb, " << path << ") = " << r;
      },
      [&] {MakeNewPath(path);
        r = tfs_rmdir(sb, path);
        LOG(TRACE) << "tfs_rmdir(sb, " << path << ") = " << r;
      },
      [&] {r = tfs_ls(sb);
        LOG(TRACE) << "tfs_ls(sb) = " << r;
      },
...
```

Figure 2: OneOf in Action

A major feature of the DeepState API not shown in the first example is the `OneOf` construct, which takes as parameter a vector, array, string, or an arbitrary number of C++ lambdas. In the first three cases, it returns a nondetermnistic element of the sequence; for lambdas, it selects and executes one of the code choices. This makes the basic "choose and run one option" structure of API call sequence testing trivial to implement. Figure 2 shows use of OneOf in a harness for testing a file system developed at U. Toronto [15]. For an extended example of `OneOf` see the TestFS harness (https://github.com/agroce/testfs) or a DeepState harness for differential testing of Google's leveldb and Facebook's rocksdb https://github.com/agroce/testleveldb. Behind the scenes, the alphabet parameter to `DeepState_CstrUpToLen` also operates as a `OneOf`, which means that swarm testing (see below) and other `OneOf` semantics automatically are used when producing restricted-alphabet strings.

3 SUPPORTED FUZZERS AND BACK ENDS

DeepState includes full-featured front-ends for libFuzzer, afl [18], libFuzzer, Eclipser [3], Angora [1], and Honggfuzz (https://github.com/google/honggfuzz), as well as a built-in, extremely fast dumb fuzzer for quick discovery of some bugs. It also, using the same interface, allows users to generate tests via symbolic execution using Manticore [14], as noted above.

In addition, DeepState can interface with any fuzzer that allows fuzzing via either file inputs or stdin, which means, in practice, the majority of fuzzers that are being developed now. If a fuzzer is similar in interface to a fully-supported fuzzer (e.g., afl, the basis for many fuzzer variants), then writing and submitting a first-class interface is usually trivial.

4 MORE THAN A FRONT-END

4.1 Swarm Testing

Swarm testing [10, 11] is a low-cost, high-impact approach to improving test generation, widely used in compiler testing [5, 12] and as a foundation for the test approach for FoundationDB, the back-end database for Apple and Snowflake cloud services [19]. Heretofore, no fuzzers to our knowledge have included support for swarm testing, which operates by constructing tests that universally omit some options in repeated choices (e.g. among API calls or options). The swarm testing concept aligns perfectly with DeepState's `OneOf` construct, and DeepState allows the use of swarm testing with any fuzzer, simply by compiling the DeepState harness with the swarm option enabled. The impact can be extremely large. For example, finding the simple stack overflow in https://github.com/agroce/deepstate-stack requires less than one second using afl, libFuzzer, or even DeepState's built-in dumb fuzzer, using swarm testing, but an average of over an hour to detect with afl when not using swarm testing, and (we estimate) trillions of years with the brute-force fuzzer not using swarm testing. A core goal of DeepState is to enable the exploration and use of test generation strategies that do not need to be implemented in individual fuzzers, but operate on a more semantic level.

Fuzzer	Runlen Mean Crash Time	TweetNaCl Bug Mean Crash Time
Ensembler	3.29s	4m13s
afl	7.01s	3m 39s
libFuzzer	4.05s	4m 19s
Angora	7.06s	10m 47s
Eclipser	2.74s	12m 45s

Table 1: Ensemble fuzzing experiments

4.2 State-of-the-Art Test Reduction

4.3 Ensemble Fuzzing

Ensemble fuzzing [2] extends the idea of ensemble learning, where multiple machine learning methods are applied to a problem, given the unpredictable diversity of performance of methods, to the fuzzing problem. DeepState supports automatic cooperative fuzzing, using tests generated by one fuzzer to seed another. The best known other implementation of ensemble fuzzing (http://wingtecher.com/Enfuzz requires uploading source code to a web site, supports a more limited set of fuzzers than DeepState (lacking Eclipser and Angora), and requires use of the very limited API of a standard libFuzzer char buffer interface to fuzzing. DeepState in contrast lets users write parameterized unit tests as usual, but then use any fuzzers DeepState provides for cooperative fuzzing.

4.3.1 Experimental Evaluation. Table 1 shows, for simple examples, some of the benefits of the ensembler. The first example, Runlen, is the DeepState example included in the distribution and discussed above. The second is a real-world bignum vulnerability http://seb.dbzteam.org/blog/2014/04/28/tweetnacl_arithmetic_bug.html#fn:2. In addition to the benefits described in the academic literature, where cooperative fuzzing can find bugs no single fuzzer finds well, an ensemble also mediates fuzzer variability, often (as here) ensuring that any faults are found about as quickly as by the *best* of the individual fuzzers. Note that Eclipser and Angora work well on the simple buggy run length encoder, but struggle with the more complex example. Is it a good idea to use Eclipser? Ensemble fuzzing lets a developer who just wants to find bugs avoid thinking about such problems.

5 RELATED WORK

DeepState lies at the intersection of practical work on unit testing frameworks (e.g., JUnit or GoogleTest) and test generation research. In particular, DeepState inherits from two approaches. Parameterized unit testing [16, 17] argues that "closed" unit tests should be opened by providing a way to specify that some values are to be filled in by a tool. DeepState extends this idea to allow the generation to be done via either fuzzers or symbolic execution tools, or simply by stored test cases. Second, DeepState is fundamentally a property-based testing [4, 13] tool, and parameterized unit test specifications work much like those in any property-based testing tool for an imperative language. The idea of a universal tool for automated test generation where the generation technology is a detail, not the focus, arises from efforts to test the Curiosity Mars Rover code at NASA/JPL [7–9].

6 CONCLUSIONS

DeepState is available as an open source tool on GitHub https://github.com/trailofbits/deepstate, and includes support for automatically building a docker environment with all supported fuzzers and symbolic execution tools installed. The version used in the demonstration and in preparing this paper can be downloaded via `docker pull agroce/deepstate_demo:latest`. With DeepState, every developer who knows how to write unit tests in C and C++ can take advantage of the power of modern fuzzers and symbolic execution tools. Moreover, we believe that DeepState may be a useful basis for experiments comparing various fuzzers, as it provides a common semantics for tests, without forcing researchers to develop equivalent test harnesses.

REFERENCES

[1] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[2] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1967–1983, 2019.

[3] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *Proceedings of the International Conference on Software Engineering*, pages 736–747, 2019.

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

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

[7] Alex Groce and Martin Erwig. Finding common ground: Choose, assert, and assume. In *International Workshop on Dynamic Analysis*, pages 12–17, 2012.

[8] Alex Groce, Klaus Havelund, Gerard Holzmann, Rajeev Joshi, and Ru-Gang Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.

[9] Alex Groce and Rajeev Joshi. Random testing and model checking: Building a common framework for nondeterministic exploration. In *Workshop on Dynamic Analysis*, pages 22–28, 2008.

[10] 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.

[11] 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.

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

[13] David R. MacIver. Hypothesis: Test faster, fix more. http://hypothesis.works/, March 2013.

[14] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[15] Jack Sun, Daniel Fryer, Ashvin Goel, and Angela Demke Brown. Using declarative invariants for protecting file-system integrity. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, 2011.

[16] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.

[17] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests with Unit Meister. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 241–244, 2005.

[18] Michal Zalewski. american fuzzy lop (2.35b). http://lcamtuf.coredump.cx/afl/, November 2014.

[19] 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.