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, 5 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 [8] and tools must be run limit knowledge transfer from one tool to another. However, a great many developers *do* know how to write *unit tests* and use a unit testing framework. For example, Google's GoogleTest is used in Chromium, LLVM, OpenCV, and many other major open source projects, and GoogleTest is referenced by more than 1 million GitHub repositories!

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

DeepState [7] provides a single front-end that allows a developer to *write a single parameterized unit test* [20] and then use a variety of fuzzers and/or symbolic execution tools to generate test cases in a single, shared, binary format. A parameterized (or generalized) unit test is a unit test that has "holes" – unknown values to be filled in by a tool. Accordingly, of course, properties must be expressed as a function of the values, rather than fixed, as in *property-based testing* [5]. The goal of DeepState is to enable developers to make use of modern test generation tools, especially including coverage-driven fuzzers, without having to learn a new paradigm, or restrict their testing to finding crashes. With DeepState, unit tests are truly "unbound" from fixed concrete values, and can be used to find problems humans would almost certainly never discover using manually written unit tests.

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), including with a tool for fuzzing R libraries written using the popular Rcpp framework (https://github.com/akhikolla/RcppDeepState); DeepState is being used in various non-public testing efforts, as well.

2 WHY UNIT TESTS NEED DEEPSTATE

The effort to use DeepState to test the popular C-Blosc (https://github.com/Blosc/c-blosc, https://github.com/Blosc/c-blosc2) compression library shows the value of using "unbound" parameterized unit tests. C-Blosc includes extensive tests, including tests checking random data for various round-trip properties. DeepState makes it possible to more succinctly express many of these tests: https://github.com/agroce/deepstate-c-blosc2.

Moreover, DeepState's use of modern fuzzers made it possible to discover a subtle bug (since fixed, see https://github.com/Blosc/c-blosc2/commit/24368ccbcd32c9d3ee6bcafa04c873b7e6751f78) in C-Blosc2, the successor to the original C-Blosc, https://github.com/Blosc/c-blosc2/issues/95. The fully minimized and normalized test (see below) shows that if C-Blosc2 is given a buffer of size 131, consisting of all 0 bytes except for byte 128 (which is 1), then when C-Blosc2 is requested to return the 129th byte without fully decompressing the buffer, it will return a value of 1 instead of 0. No smaller or simpler input will expose the bug.

3 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 unit tests using the very widely used GoogleTest (https://github.com/google/googletest) framework. The critical difference is the use of DeepState API calls to replace fixed, concrete values with tool-generated values, and the

```
#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

use of more general, property-based-testing style, specifications to account for the variability of test values. 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 [17], the Eclipser fuzzer [4], afl in QEMU mode, or DeepState's built-in dumb (but very fast) fuzzer (Figure 2).

Replaying a test case is then trivial:

```
> ./runlen --input_test_file
    bruteforce_fuzzing/5ef02d497259a3c0bd3bda1a2f180cf5be5db973.fail
TRACE: Initialized test input buffer with 20 bytes of data from
‘bruteforce_fuzzing/5ef02d497259a3c0bd3bda1a2f180cf5be5db973.fail‘
TRACE: Running: Runlength_EncodeDecode from Runlen.cpp(55)
CRITICAL: Runlen.cpp(60): ORIGINAL: 'd011', ENCODED: 'dA0A1A',
ROUNDTRIP: 'd01'
ERROR: Failed: Runlength_EncodeDecode
ERROR: Test case
bruteforce_fuzzing/5ef02d497259a3c0bd3bda1a2f180cf5be5db973.fail failed
```

The directory structure for other fuzzers is more complex, but largely consistent across fuzzers. Once users know where output files are, it is easy to, e.g., collect all test cases from all testing approaches applied. DeepState also provides command line option to run only selected tests, if multiple tests are defined in a harness, and other common unit testing features.

Fuzzing with afl, libFuzzer, 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
...
> deepstate-libfuzzer --compile_test ./Runlen.cpp --out_test_name runlen
...
> deepstate-libfuzzer ./runlen.libfuzzer --fuzzer_out --output_test_dir
  libfuzzer_fuzzing
```

DeepState knows the required compiler options to produce instrumented fuzzing executables.

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

```
> clang++ -o runlen Runlen.cpp -ldeepstate
> deepstate-manticore ./runlen --output_test_dir manticore_symex
INFO:deepstate:Setting log level from DEEPSTATE_LOG: 2
INFO:deepstate.core.base:Setting log level from --min_log_level: 2
-04-27 19:16:51,953: [24] m.n.manticore:INFO: Loading program ./runlen
-04-27 19:21:37,415: [24] m.n.manticore:INFO: Loading program ./runlen
...
INFO:deepstate.core.symex:Saved test case in file
manticore_symex/Runlen.cpp/Runlength_EncodeDecode/
a2604eeb87f7be3ff2990e63e02bdeed.crash
...
INFO:deepstate.core.symex:Saved test case in file
manticore_symex/Runlen.cpp/Runlength_EncodeDecode/
d737b2217daf7671e3346b2cf533270.crash
...
...
> deepstate-eclipser ./runlen --output_test_dir eclipser_fuzzing --fuzzer_out
INFO:deepstate:Setting log level from DEEPSTATE_LOG: 2
INFO:deepstate.core.base:Setting log level from --min_log_level: 2
INFO:deepstate.core.fuzz:Calling pre_exec before fuzzing
WARNING:deepstate.executors.fuzz.eclipser:Eclipser doesn't limit child
processes memory.
INFO:deepstate.core.fuzz:Executing command
‘['dotnet', '/home/user/deps/eclipser/Eclipser.dll', 'fuzz', '--program',
'/home/user/deepstate/examples/runlen', '--src', 'file', '--fixfilepath',
  'eclipser.input', '--initarg', '--input_test_file eclipser.input
  --abort_on_fail --no_fork --min_log_level 2', '--outputdir',
  'eclipser_fuzzing/the_fuzzer', '--maxfilelen', '8192', '--timelimit',
  '99999']‘
INFO:deepstate.core.fuzz:Using fuzzer output.
[00:00:00:00] [*] Fuzz target : /home/user/deepstate/examples/runlen
[00:00:00:00] [*] Time limit : 99999 sec
[00:00:00:00] [*] Total 1 initial seeds
[00:00:00:00] [*] 1 initial items with high priority
[00:00:00:00] [*] 0 initial items with low priority
[00:00:00:00] [*] Fuzzing starts
[00:00:00:00] [*] Found by grey-box concolic (9 new edges)
[00:00:00:00] Save crash seed : ("--input_test_file" "eclipser.input"
  "--abort_on_fail" "--no_fork" "--min_log_level" "2")
  File[0]=( 96* ...15bytes... (0) (Right))
[00:00:00:00] [*] Found by grey-box concolic (139 new edges)
[00:00:00:00] Save crash seed : ("--input_test_file" "eclipser.input"
  "--abort_on_fail" "--no_fork" "--min_log_level" "2")
  File[0]=( 7d* ...15bytes... (0) (Right))
[00:00:00:00] [*] Found by grey-box concolic (13 new edges)
[00:00:00:00] Save crash seed : ("--input_test_file" "eclipser.input"
  "--abort_on_fail" "--no_fork" "--min_log_level" "2")
  File[0]=( 64* ...15bytes... (0) (Right))
[00:00:00:00] [*] Found by grey-box concolic (1 new edges)
[00:00:00:00] [*] Found by grey-box concolic (2 new edges)
INFO:deepstate.executors.fuzz.eclipser:Performing decoding on testcases
  and crashes
...
> ./runlen --fuzz --output_test_dir bruteforce_fuzzing
INFO: Starting fuzzing
WARNING: No seed provided; using 1619628901
CRITICAL: Runlen.cpp(60): ORIGINAL: 'd011', ENCODED: 'dA0A1A', ROUNDTRIP: 'd01'
ERROR: Failed: Runlength_EncodeDecode
INFO: Saved test case in file
‘bruteforce_fuzzing/5ef02d497259a3c0bd3bda1a2f180cf5be5db973.fail‘
...
```

Figure 2: Using DeepState to Test the Runlength Encoder

```
  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 3: OneOf in Action

API call sequence testing trivial to implement. Figure 3 shows use of OneOf in a harness for testing a file system developed at U. Toronto [19]. 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. In addition to `OneOf`, DeepState also provides `OneOfP`, which allows users to control probabilities of options in a nondeterminstic choice. DeepState mediates the translation, since probabilities are meaningless in pure symbolic execution, but may be very useful for fuzzers, in particular the brute-force fuzzer.

4 SUPPORTED FUZZERS AND BACK ENDS

DeepState includes full-featured front-ends for libFuzzer, afl [22], Eclipser [4], Angora [1], and Honggfuzz (https://github.com/google/honggfuzz), as well as a built-in, extremely fast (if dumb) fuzzer for quick discovery of bugs. It also, using the same interface, allows users to generate tests via symbolic execution using Manticore [17], 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. Support for more fuzzers and symbolic execution tools is in progress.

5 MORE THAN JUST A FRONT-END

DeepState, in addition to allowing users to test complex properties with a variety of fuzzer and symbolic execution backends, also offers features not included in the back-end tools, including powerful methods for improving the effectiveness of tests.

5.1 Swarm Testing

Swarm testing [13, 14] is a low-cost, high-impact approach to improving test generation, widely used in compiler testing [6, 15] and as a foundation for the test approach for FoundationDB, the back-end database for Apple and Snowflake cloud services [24]. 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.

5.2 State-of-the-Art Test Reduction

Fuzzer-generated tests, particularly of API call sequences, are notoriously hard to understand [23]. DeepState therefore provides a state-of-the-art test reduction facility. In addition to producing smaller tests that omit unnecessary steps, DeepState's reducer attempts to *normalize* failing tests [11, 16] to simplify test values and make it easier to perform bug triage [2].

Reducing a test produced by any back-end is simple:

```
> deepstate-reduce ./runlen
  bruteforce_fuzzing/5ef02d497259a3c0bd3bda1a2f180cf5be5db973.fail
  min.fail
INFO:deepstate:Setting log level from DEEPSTATE_LOG: 2
Original test has 20 bytes
Applied 4 range conversions
Writing reduced test with 20 bytes to min.fail
==================================================
Iteration #1 0.01 secs / 2 execs / 0.0% reduction
Removed 1 byte(s) @ 0: reduced test to 19 bytes
Writing reduced test with 19 bytes to min.fail
 0.02 secs / 3 execs / 5.0% reduction
==================================================
...
Byte range removal: PASS FINISHED IN 0.0 SECONDS, RUN: 0.11 secs / 28 execs /
  95.0% reduction
Reduced byte 0 from 3 to 2
Writing reduced test with 1 bytes to min.fail
 0.12 secs / 31 execs / 95.0% reduction
...
Completed 2 iterations: 0.14 secs / 36 execs / 95.0% reduction
Padding test with 11 zeroes
Writing reduced test with 12 bytes to min.fail
> ./runlen --input_test_file min.fail
TRACE: Initialized test input buffer with 12 bytes of data from 'min.fail'
TRACE: Running: Runlength_EncodeDecode from Runlen.cpp(50)
CRITICAL: Runlen.cpp(55): ORIGINAL: 'aa', ENCODED: 'aA', ROUNDTRIP: 'a'
ERROR: Failed: Runlength_EncodeDecode
ERROR: Test case min.fail failed
```

The original failing input (`"d011"`, see Figure 2) is reduced to the "simplest" possible failing input, `"aa"`. The criteria for reduction can be more complex than simply preserving failure, including checking an arbitrary regular expression over the failure symptom, or even running a script to ensure maintenance of code coverage.

5.3 Ensemble Fuzzing

Ensemble fuzzing [3] 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.

5.3.1 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

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

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 sitestep such questions.

5.4 Work in Progress: Advanced Features

In addition to these fully developed features, work is in progress to allow DeepState to generate standalone GoogleTest unit tests from any fuzzing-produced test (a prototype version of this functionality exists, which additionally allows afl to directly generate unit test files as it runs), to enable DeepState to check properties of the output (logging or internal printing) from test execution as in the LogScope language [10], and numerous other useful features that are unlikely to be added to any of DeepState's back-ends (e.g., autoamatic checks for assertions that are never executed).

6 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 [20, 21] 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 [5] tool, where specifications work like those in any property-based testing tool for an imperative language. The idea of a universal automated test generation tool where the underlying technology is not the focus arises from efforts to test Mars Rover code at NASA/JPL [8, 9, 12]. Finally, the very recent work of Reid et al. [18], proposes the same goal as we do: to improve adoption of powerful modern testing methods by essentially allowing developers to work with paradigms with which they are familiar, though with a focus on formal verification and Rust, not fuzzing and C/C++.

7 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] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[3] 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 *USENIX Security Symposium*, pages 1967–1983, 2019.

[4] 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.

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

[6] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using CLP. In *International Conference on Automated Software Engineering*, pages 482–493, 2015.

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

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

[9] 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.

[10] Alex Groce, Klaus Havelund, and Margaret Smith. From scripts to specifications: The evolution of a flight software testing effort. In *International Conference on Software Engineering*, pages 129–138, 2010.

[11] Alex Groce, Josie Holmes, and Kevin Kellar. One test to rule them all. In *International Symposium on Software Testing and Analysis*, page 1–11, 2017.

[12] 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.

[13] 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.

[14] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Int. Symposium on Software Testing and Analysis*, pages 78–88, 2012.

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

[16] David MacIver and Alastair F. Donaldson. Test-case reduction via test-case generation: Insights from the Hypothesis reducer. In *ECOOP*, pages 13:1–13:27, 2020.

[17] 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 *International Conference on Automated Software Engineering*, pages 1186–1189, 2019.

[18] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. Towards making formal methods normal: meeting developers where they are. In *Human Aspects of Types and Reasoning Assistants*, 2020.

[19] 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.

[20] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *International Symposium on Foundations of Software Engineering*, pages 253–262, 2005.

[21] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests with Unit Meister. In *International Symposium on Foundations of Software Engineering*, pages 241–244, 2005.

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

[23] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.

[24] 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) Begin with the problem: developers could find many bugs using modern automated test generation tools, but few developers know how to use even afl, much less symbolic execution tools.
(2) Show the code for using the afl fuzzer to test a simple function that takes three ints as input, parsing the raw binary data into integers.
(3) Show a Manticore script for binary analysis of a C program.
(4) Say that while developers don't know how to use these tools, they DO know how to use unit testing, often. Show the GoogleTest website and a search showing the vast number of github repos referencing GoogleTest. Point out that there are other C/C++ unit testing frameworks, all look fairly similar.
(5) Discuss that DeepState meets developers where they are: it lets them write parameterized unit tests in C/C++ using an API similar to GoogleTest, then use various back-ends to generate tests.
(6) First, show DeepState github repo, how to build deepstate docker image, compile DeepState directly on a linux/mac machine.
(7) Launch a DeepState docker image. Navigate to the examples directory. Remove the Boring test from the Runlen example, explain that DeepState has options to control which test(s) run, but for demonstration purposes, we'll stick to just the one interesting test. Step through the source code, showing the call that makes the test parameterized.
(8) Compile the test at the command line, and show how to run:
    - The brute-force fuzzer
    - Eclipser
    - The Manticore symbolic execution tool

    Explain that manticore will be much slower than the fuzzers, but can sometimes find problems other tools cannot. Using Manticore without DeepState is painful (even though Manticore is "friendly" as binary analysis tools go), this was the actual first motivation for writing DeepState!
(9) Show how DeepState handles compiling code that has to be instrumented. Run afl and libFuzzer on the same example.
(10) Replay failing tests from the brute force fuzzer and afl.
(11) Show how to use DeepState to reduce/simplify a failing test. The brute force and afl tests reduce to the same input, making bug triage easier.
(12) Show the DeepState C-Blosc2 harness, briefly step through the code, and show the bug found using DeepState.
(13) Briefly mention swarm testing, show the deepstate-stack example in docker, show how with swarm it trivially finds the stack overflow, but without it, afl will need an hour or more to find the bug, the brute force fuzzer will NEVER find it (do back of the envelope math showing need for trillions of years of runtime).
(14) Mention related work, the general goal of meeting developers where they are, invite users to play with the uploaded demo docker image.