Fuzzing an API with DeepState (Part 1)

POST JANUARY 22, 2019 1 COMMENT

Alex Groce, Associate Professor, School of Informatics, Computing and Cyber Systems, Northern Arizona University

Using DeepState, we took a handwritten red-black tree fuzzer and, with minimal effort, turned it into a much more fully featured test generator. The DeepState fuzzer, despite requiring no more coding effort, supports replay of regression tests, reduction of the size of test cases for debugging, and multiple data-generation back-ends, including Manticore, angr, libFuzzer, and AFL. Using symbolic execution, we even discovered artificially introduced bugs that the original fuzzer missed. After reading this article, you should be ready to start applying high-powered automated test generation to your own APIs.

Background

In 2013, John Regehr wrote a blog post on "How to Fuzz an ADT Implementation." John wrote at some length about general issues in gaining confidence that a data-type implementation is reliable, discussing code coverage, test oracles, and differential testing. If you have not yet read John's article, then I recommend reading it now. It gives a good overview of how to construct a simple custom fuzzer for an ADT, or, for that matter, any fairly self-contained API where there are good ways to check for correctness.

The general problem is simple. Suppose we have a piece of software that provides a set of functions or methods on objects. Our running example in this post is a red-black tree; however, an AVL tree, a file-system, an in-memory store, or even a crypto library could easily be swapped in. We have some expectations about what will happen when we call the available functions. Our goal is to thoroughly test the software, and the traditional unit-testing approach to the problem is to write a series of small functions that look like:

[code lang="cpp"]result1 = foo(3, "hello");
result2 = bar(result1, "goodbye")
assert(result2 == DONE);[/code]

That is, each test has the form: "do something, then check that it did the right thing." This approach has two problems. First, it's a lot of work. Second, the return on investment for that work is not as good as you would hope; each test does one specific thing, and if the author of the tests doesn't happen to think of a potential problem, then the tests are very unlikely to catch that problem. These unit tests are insufficient for the same reasons that AFL and other fuzzers have been so successful at finding security vulnerabilities in widely used programs: humans are too slow at writing

many tests, and are limited in their ability to imagine insane, harmful inputs. The randomness of fuzzing makes it possible to produce many tests very quickly and results in tests that go far outside the "expected uses."

Fuzzing is often thought of as generating files or packets, but it can also generate sequences of API calls to test software libraries. Such fuzzing is often referred to as random or randomized testing, but *fuzzing is fuzzing*. Instead of a series of unit tests doing one specific thing, a fuzzer test (also known as a property-based test or a parameterized unit test) looks more like:

[code lang="cpp"]foo_result = NULL;
bar_result = NULL;
repeat LENGTH times:
switch (choice):
choose_foo:
foo_result = foo(randomInt(), randomString());
break;
choose_bar:
bar_result = bar(foo_result, randomString());
break;
choose_baz:
baz_result = baz(foo_result, bar_result);
break;
checkInvariants();[/code]

That is, the fuzzer repeatedly chooses a random function to call, and then calls the chosen function, perhaps storing the results for use in later function calls.

A well-constructed test of this form will include lots of generalized assertions about how the system should behave, so that the fuzzer is more likely to shake out unusual interactions between the function calls. The most obvious such checks are any assertions in the code, but there are numerous other possibilities. For a data structure, this will come in the form of a `repOK` function that makes sure that the ADT's internal representation is in a consistent state. For red-black trees, that involves checking node coloring and balance. For a file system, you may expect that `chkdsk` will never find any errors after a series of valid file system operations. In a crypto library (or a JSON parser, for that matter, with some restrictions on the content of message) you may want to check round-trip properties: `message == decode(encode(message, key), key)`. In many cases, such as with ADTs and file systems, you can use another implementation of the same or similar functionality, and compare results. Such *differential* testing is extremely powerful, because it lets you write a very complete specification of correctness with relatively little work.

John's post doesn't just give general advice, it also includes links to a working fuzzer for a red-black tree. The fuzzer is effective and serves as a great example of how to really hammer an API using a solid test harness based on random value generation. However, it's also not a completely practical testing tool. It generates inputs, and tests the red-black tree, but when the fuzzer finds a bug, it simply prints an error message and crashes.

- 2012

CATEGORIES

- Apple (12)
- Attacks (7)
- Authentication (5)
- Binary Ninja (11)
- Blockchain (38)
- Capture the Flag (10)
- Compilers (20)
- Conferences (27)
- Containers (2)
- Cryptography (28)
- Crytic (2)
- Cyber Grand Challenge (7)
- DARPA (18)
- Dynamic Analysis (12)
- Education (13)
- Empire Hacking (7)
- Engineering Practice (10)
- Events (5)
- Exploits (22)
- Fuzzing (24)
- Go (3)
- Guides (8)
- Internship Projects (18)
- iVerify (4)
- Kubernetes (2)
- Linux (1)
- Malware (7)
- Manticore (13)
- McSema (11)
- Meta (10)
- Mitigations (9)
- osquery (20)
- Paper Review (11)
- Press Release (22)
- Privacy (7)
- Products (5)
- Program Analysis (15)
- Research Practice (9)
- Reversing (12)
- Rust (3)
- Sponsorships (12)
- Static Analysis (20)
- Symbolic Execution (14)
- Training (1)
- Year in Review (4)

Tweets by @trailofbits

Trail of Bits Retweeted

Real World Crypto
@RealWorldCrypto

Replying to @RealWorldCrypto

Thanks to @CryptoExperts @digicert @ISARACorp @NuCypher Technology Innovation Institute and @trailofbits for

You don't learn anything except "Your code has a bug. Here is the symptom." Modifying the code to print out the test steps as they happen slightly improves the situation, but there are likely to be hundreds or thousands of steps before the failure.

Ideally, the fuzzer would automatically store failing test sequences in a file, minimize the sequences to make debugging easy, and make it possible to replay old failing tests in a regression suite. Writing the code to support all this infrastructure is no fun (especially in C/C++) and dramatically increases the amount of work required for your testing effort. Handling the more subtle aspects, such as trapping assertion violations and hard crashes so that you write the test to the file system before terminating, is also hard to get right.

AFL and other general-purpose fuzzers usually provide this kind of functionality, which makes fuzzing a much more practical tool in debugging. Unfortunately, such fuzzers are not convenient for testing APIs. They typically generate a file or byte buffer, and expect that the program being tested will take that file as input. Turning a series of bytes into a red-black tree test is probably easier and more fun than writing all the machinery for saving, replaying, and reducing tests, but it still seems like a lot of work that isn't directly relevant to your real task: figuring out how to describe valid sequences of API calls, and how to check for correct behavior. What you really want is a unit testing framework like GoogleTest, but one that is capable of varying the input values used in tests. There are lots of good tools for random testing, including my own TSTL, but few sophisticated ones target C/C++, and none that we are aware of let you use any test generation method other than the tools' built-in random tester. That's what we want: GoogleTest, but with the ability to use libFuzzer, AFL, HonggFuzz, or what you will to generate data.

Enter DeepState

DeepState fills that need, and more. (We'll get to the 'more' when we discuss symbolic execution).

Translating John's fuzzer into a DeepState test harness is relatively easy. Here is a DeepState version of "the same fuzzer." The primary changes for DeepState, which can be found in the file deepstate_harness.cpp, are:

- Remove `main` and replace it with a named test `(TEST(RBTree, GeneralFuzzer))`
  - A DeepState file can contain more than one named test, though it is fine to only have one test.
- Just create one tree in each test, rather than having an outer loop that iterates over calls that affect a single tree at a time.
  - Instead of a fuzzing loop, our tests are closer to very generalized unit tests: each test does one sequence of interesting API calls.
  - DeepState will handle running multiple tests; the fuzzer or symbolic execution engine will provide the "outer loop."
- Fix the length of each API call sequence to a fixed value, rather than a random one.
  - The `#define LENGTH 100` at the top of the file controls how many functions we call in each test.

- Having bytes be in somewhat the same positions in every test is helpful for mutation-based fuzzers. Extremely long tests will go beyond libFuzzer's default byte length.
- So long as they don't consume so many bytes that fuzzers or DeepState reach their limits, or have trouble finding the right bytes to mutate, longer tests are usually better than shorter tests. There may be a length five sequence that exposes your bug, but DeepState's brute-force fuzzer and even libFuzzer and AFL will likely have trouble finding it, and more easily produce a length 45 version of the same problem. Symbolic execution, on the other hand, will find such rare sequences for any length it can handle.
- For simplicity, we use a `#define` in our harness, but it is possible to define such testing parameters as optional command-line arguments with a default value, for even greater flexibility in testing. Just use the same tools as DeepState uses to define its own command-line options (see DeepState.c and DeepState.h).

- Replace various `rand() % NNN` calls with `DeepState_Int()`, `DeepState_Char()` and `DeepState_IntInRange(...)` calls.
  - DeepState provides calls to generate most of the basic data types you want, optionally over restricted ranges.
  - You can actually just use `rand()` instead of making DeepState calls. If you include DeepState and have defined `DEEPSTATE_TAKEOVER_RAND`, all `rand` calls will be translated to appropriate DeepState functions. The file easy_deepstate_fuzzer.cpp shows how this works, and is the simplest translation of John's fuzzer. It isn't ideal, since it doesn't provide any logging to show what happens during tests. This is often the easiest way to convert an existing fuzzer to use DeepState; the changes from John's fuzzer are minimal: 90% of the work is just changing a few includes and removing `main`.
- Replace the `switch` statement choosing the API call to make with DeepState's `OneOf` construct.
  - `OneOf` takes a list of C++ lambdas, and chooses one to execute.
  - This change is not strictly required, but using `OneOf` simplifies the code and allows optimization of choices and smart test reduction.
  - Another version of `OneOf` takes a fixed-size array as input, and returns some value in it; e.g., `OneOf("abcd")` will produce a character, either `a`, `b`, `c`, or `d`.

There are a number of other cosmetic (e.g. formatting, variable naming) changes, but the essence of the fuzzer is clearly preserved here. With these changes, the fuzzer works almost as before, except that instead of running the `fuzz_rb` executable, we'll use DeepState to run the test we've defined and generate input values that choose which function calls to make, what values to insert in the red-black tree, and all the other decisions represented by `DeepState_Int`, `OneOf`, and other calls:

[code lang="cpp"]int GetValue() {
if (!restrictValues) {
return DeepState_Int();
} else {
return DeepState_IntInRange(0, valueRange);
}
}

Jun 23, 2020

Trail of Bits Retweeted

How-To Geek
@howtogeek

This open-source software allows you to host your own VPN.
howtogeek.com/669848/how-to-… by @ianpaul

How to Host Your Own VPN with …
Companies all over the world sell V…
howtogeek.com

Jun 23, 2020

Trail of Bits Retweeted

Built In NYC
@BuiltInNewYork

Remote employees @trailofbits, @CleancultTeam and @radarlabs share their best tips for maintaining productivity and work-life balance. ow.ly/Ot2y50AcHyZ

How to Make the Most of Remote …
shutterstock Before the coronavirus…
builtinnyc.com

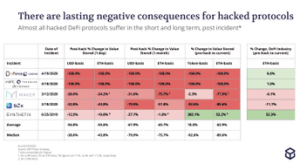
Jun 23, 2020

Trail of Bits Retweeted

Prysm Group
@prysmeconomics

Replying to @prysmeconomics

Join us Friday as we review hacks and future security threats in DeFi. What are the short-term and long-term economic implications to the underlying networks? @prysmeconomics @compoundfinance @trailofbits
Register here: bit.ly/2Ydt6RX



Jun 22, 2020

Trail of Bits Retweeted

Zokyo
@Zokyo_

This is great talk

```
...
for (int n = 0; n < LENGTH; n++) {
OneOf(
[&] {
int key = GetValue();
int* ip = (int*)malloc(sizeof(int));
*ip = key;
if (!noDuplicates || !containerFind(*ip)) {
void* vp = voidP();
LOG(TRACE) << n << ": INSERT:" << *ip << " " << vp;
RBTreeInsert(tree, ip, vp);
containerInsert(*ip, vp);
} else {
LOG(TRACE) << n << ": AVOIDING DUPLICATE INSERT:" << *ip;
free(ip);
}
},
[&] {
int key = GetValue();
LOG(TRACE) << n << ": FIND:" << key;
if ((node = RBExactQuery(tree, &key))) {
ASSERT(containerFind(key)) << "Expected to find " << key;
} else {
ASSERT(!containerFind(key)) << "Expected not to find " << key;
}
},
...
```

Thanks @dguido
https://twitter.com/dguido/status/1273627083449806849

Jun 19, 2020

Installing DeepState

The DeepState GitHub repository provides more details and dependencies, but on my MacBook Pro, installation is simple:

```
[code lang="bash"]git clone https://github.com/trailofbits/deepstate
cd deepstate
mkdir build
cd build
cmake ..
sudo make install[/code]
```

Building a version with libFuzzer enabled is slightly more involved:

```
[code lang="bash"]brew install llvm@7
git clone https://github.com/trailofbits/deepstate
cd deepstate
mkdir build
cd build
CC=/usr/local/opt/llvm\@7/bin/clang
CXX=/usr/local/opt/llvm\@7/bin/clang++ BUILD_LIBFUZZER=TRUE cmake ..
sudo make install[/code]
```

AFL can also be used to generate inputs for DeepState, but most of the time, raw speed (due to not needing to fork), decomposition of compares,

and value profiles seem to give libFuzzer an edge for this kind of API testing, in our (limited experimentally!) experience. For more on using AFL and other file-based fuzzers with DeepState, see the DeepState README.

Using the DeepState Red-Black Tree Fuzzer

Once you have installed DeepState, building the red-black tree fuzzer(s) is also simple:

[code lang="bash"]git clone https://github.com/agroce/rb_tree_demo
cd rb_tree_demo
make[/code]

The `make` command compiles everything with all the sanitizers we could think of (address, undefined, and integer) in order to catch more bugs in fuzzing. This has a performance penalty, but is usually worth it.

If you are on macOS and using a non-Apple clang in order to get libFuzzer support, you'll want to do something like

[code lang="bash"]CC=/usr/local/opt/llvm\@7/bin/clang
CXX=/usr/local/opt/llvm\@7/bin/clang++ make[/code]

in order to use the right (e.g., homebrew-installed) version of the compiler.

This will give you a few different executables of interest. One, `fuzz_rb`, is simply John's fuzzer, modified to use a 60-second timeout instead of a fixed number of "meta-iterations." The `ds_rb` executable is the DeepState executable. You can fuzz the red-black tree using a simple brute-force fuzzer (that behaves very much like John's original fuzzer):

[code lang="bash"]mkdir tests
./ds_rb –fuzz –timeout 60 –output_test_dir tests[/code]

If you want to see more about what the fuzzer is doing, you can specify a log level using `--min_log_level` to indicate the minimum importance of messages you want to see. A `min_log_level` of `0` corresponds to including all messages, even debug messages; `1` is `TRACE` messages from the system under test (e.g., those produced by the `LOG(TRACE)` code shown above); `2` is `INFO`, non-critical messages from DeepState itself (this is the default, and usually appropriate); `3` is warnings, and so forth up the hierarchy. The `tests` directory should be empty at the termination of fuzzing, since the red-black tree code in the repo (to my knowledge) has no bugs. If you add `--fuzz_save_passing` to the options, you will end up with a large number of files for passing tests in the directory.

Finally, we can use libFuzzer to generate tests:

[code lang="bash"]mkdir corpus
./ds_rb_lf corpus -use_value_profile=1 -detect_leaks=0 -max_total_time=60[/code]

The `ds_rb_lf` executable is a normal libFuzzer executable, with the same command line options. This will run libFuzzer for 60 seconds, and place any interesting inputs (including test failures) in the `corpus` directory. If there is a crash, it will leave a `crash-` file in the current directory. You can tune it to perform a little better in some cases by determining the maximum input size your tests use, but this is a non-trivial exercise. In our case at length 100 the gap between our max size and 4096 bytes is not extremely large.

For more complex code, a coverage-driven, instrumentation-based fuzzer like libFuzzer or AFL will be much more effective than the brute force randomness of John's fuzzer or the simple DeepState fuzzer. For an example like the red-black-tree, this may not matter as much, since few states may be very hard to reach for a fast "dumb" fuzzer. Even here, however, smarter fuzzers have the advantage of producing a corpus of tests that produce interesting code coverage. DeepState lets you use a faster fuzzer for quick runs, and smarter tools for more in-depth testing, with almost no effort.

We can replay any DeepState-generated tests (from libFuzzer or DeepState's fuzzer) easily:

[code lang="bash"]./ds_rb –input_test_file file[/code]

Or replay an entire directory of tests:

[code lang="bash"]./ds_rb –input_test_files_dir dir[/code]

Adding an `--exit_on_fail` flag when replaying an entire directory lets you stop the testing as soon as you hit a failing or crashing test. This approach can easily be used to add failures found with DeepState (or interesting passing tests, or perhaps corpus tests from libFuzzer) to automatic regression tests for a project, including in CI.

Adding a Bug

This is all fine, but it doesn't (or at least shouldn't) give us much confidence in John's fuzzer or in DeepState. Even if we changed the `Makefile` to let us see code coverage, it would be easy to write a fuzzer that doesn't actually check for correct behavior – it covers everything, but doesn't find any bugs other than crashes. To see the fuzzers in action (and see more of what DeepState gives us), we can add a moderately subtle bug. Go to line 267 of `red_black_tree.c` and change the `1` to a `0`. The `diff` of the new file and the original should look like:

[code]
267c267
< x->parent->parent->red=0;
—
> x->parent->parent->red=1;
[/code]

Do a `make` to rebuild all the fuzzers with the new, broken `red_black_tree.c`.

Running John's fuzzer will fail almost immediately:

[code lang="bash"]time ./fuzz_rb
Assertion failed: (left_black_cnt == right_black_cnt), function
checkRepHelper, file red_black_tree.c, line 702.
Abort trap: 6

real 0m0.100s
user 0m0.008s
sys 0m0.070s[/code]

Using the DeepState fuzzer will produce results almost as quickly. (We'll let it show us the testing using the `--min_log_level` option, and tell it to stop as soon as it finds a failing test.):

[code lang="bash"]time ./ds_rb –fuzz –min_log_level 1 –exit_on_fail –
output_test_dir tests
INFO: Starting fuzzing
WARNING: No seed provided; using 1546625762
WARNING: No test specified, defaulting to last test defined
(RBTree_GeneralFuzzer)
TRACE: Running: RBTree_GeneralFuzzer from deepstate_harness.cpp(78)
TRACE: deepstate_harness.cpp(122): 0: DELETE:-747598508
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(122): 1: DELETE:831257296
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(134): 2: PRED:1291220586
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(154): 4: SUCC:-1845067087
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(113): 6: FIND:-427918646
TRACE: deepstate_harness.cpp(190): checkRep...
...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(103): 44: INSERT:-1835066397
0x00000000ffffff9c
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(154): 46: SUCC:-244966140
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...

TRACE: deepstate_harness.cpp(103): 48: INSERT:1679127713
0x00000000ffffffa4
TRACE: deepstate_harness.cpp(190): checkRep...
Assertion failed: (left_black_cnt == right_black_cnt), function
checkRepHelper, file red_black_tree.c, line 702.
ERROR: Crashed: RBTree_GeneralFuzzer
INFO: Saved test case to file
`tests/6de8b2ffd42af6878875833c0cbfa9ea09617285.crash`
...
real 0m0.148s
user 0m0.011s
sys 0m0.131s[/code]

I've omitted much of the output above, since showing all 49 steps before the detection of the problem is a bit much, and the details of your output will certainly vary. The big difference from John's fuzzer, besides the verbose output, is the fact that DeepState *saved a test case*. The name of your saved test case will, of course, be different, since the names are uniquely generated for each saved test. To replay the test, I would do this:

[code lang="bash"]./ds_rb –input_test_file
tests/6de8b2ffd42af6878875833c0cbfa9ea09617285.crash[/code]

and I would get to see the whole disaster again, in gory detail. As we said above, this lengthy sequence of seemingly arbitrary operations isn't the most helpful test for seeing what's going on. DeepState can help us here:

[code lang="bash"]deepstate-reduce ./ds_rb
tests/6de8b2ffd42af6878875833c0cbfa9ea09617285.crash minimized.crash
ORIGINAL TEST HAS 8192 BYTES
LAST BYTE READ IS 509
SHRINKING TO IGNORE UNREAD BYTES
ONEOF REMOVAL REDUCED TEST TO 502 BYTES
ONEOF REMOVAL REDUCED TEST TO 494 BYTES
...
ONEOF REMOVAL REDUCED TEST TO 18 BYTES
ONEOF REMOVAL REDUCED TEST TO 2 BYTES
BYTE RANGE REMOVAL REDUCED TEST TO 1 BYTES
BYTE REDUCTION: BYTE 0 FROM 168 TO 0
NO (MORE) REDUCTIONS FOUND
PADDING TEST WITH 49 ZEROS

WRITING REDUCED TEST WITH 50 BYTES TO minimized.crash[/code]

Again, we omit some of the lengthy process of reducing the test. The new test is (much!) easier to understand:

[code lang="bash"]./ds_rb –input_test_file minimized.crash
WARNING: No test specified, defaulting to last test defined
(RBTree_GeneralFuzzer)
TRACE: Initialized test input buffer with data from `minimized.crash`
TRACE: Running: RBTree_GeneralFuzzer from deepstate_harness.cpp(78)
TRACE: deepstate_harness.cpp(103): 0: INSERT:0 0x0000000000000000

```
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(103): 1: INSERT:0 0x0000000000000000
TRACE: deepstate_harness.cpp(190): checkRep...
TRACE: deepstate_harness.cpp(192): RBTreeVerify...
TRACE: deepstate_harness.cpp(103): 2: INSERT:0 0x0000000000000000
TRACE: deepstate_harness.cpp(190): checkRep...
Assertion failed: (left_black_cnt == right_black_cnt), function
checkRepHelper, file red_black_tree.c, line 702.
ERROR: Crashed: RBTree_GeneralFuzzer[/code]
```

We just need to insert three identical values into the tree to expose the problem. Remember to fix your `red_black_tree.c` before proceeding!

You can watch the whole process in action:

In Part 2, we'll look at how to assess the quality of our testing: is our DeepState testing as effective as John's fuzzer? Are both approaches unable to find certain subtle bugs? And what about symbolic execution?

By Alex Groce Posted in Dynamic Analysis, Fuzzing, Manticore, Symbolic Execution

← How McSema Handles C++ Exceptions

Fuzzing an API with DeepState (Part 2) →

One thought on "Fuzzing an API with DeepState (Part 1)"

Pingback: Fuzzing an API with DeepState (Part 2) | Trail of Bits Blog

Leave a Reply

Enter your comment here...