DeepState: Symbolic Unit Testing for C and C++

Peter Goodman
Trail of Bits, Inc.
peter@trailofbits.com

Alex Groce
School of Informatics, Computing & Cyber Systems
Northern Arizona University
alex.groce@nau.edu

Abstract—**Unit testing is a popular software development methodology that can help developers detect functional regressions, explore boundary conditions, and document expected behavior. However, writing comprehensive unit tests is challenging and time-consuming, and developers seldom explore the obscure (and bug-hiding) corners of software behavior without assistance.**

DeepState is a tool that provides a Google Test-like API to give C and C++ developers push-button access to symbolic execution engines, such as Manticore and angr, and fuzzers, such as Dr. Fuzz. Rather than learning multiple complex tools, users learn one interface for defining a test harness, and can use various methods to automatically generate tests for software. In addition to providing a familiar interface to binary analysis and fuzzing for parameterized unit testing, DeepState also provides constructs that aid in the construction of API-sequence tests, where the tool chooses the functions or methods to call, allowing for even more diverse and powerful tests. By serving as a front-end to multiple tools, DeepState additionally provides a way to apply (novel) high-level strategies to test generation, and to compare effectiveness and efficiency of testing back-ends, including binary analysis tools.

I. Introduction

A key limitation in the advancement of binary analysis and other approaches to improving software security and reliability is that there is very little overlap between security experts familiar with tools such as angr [34], [35], [33], Manticore [27], or S2E [9], and the developers who produce most code that needs to be secure or highly reliable.

Developers, as a class, do not know how to use binary analysis tools; developers, as a class, seldom even know how to use less challenging tools such as fuzzers, even relatively push-button ones such as AFL [41]. Even those developers whose primary code focus is critical security infrastructure such as OpenSSL are often not users, much less expert users, of such tools.

Developers *do*, however, often know how to use unit testing frameworks, such as JUnit [12] or Google Test [1]. DeepState aims to bring some of the power (in particular, high quality automated test generation) of binary analysis frameworks to a larger audience of developers. DeepState makes it possible to write parameterized unit tests [38] in a Google Test-like framework, and automatically produce tests using angr, Manticore, or Dr. Memory's fuzzer, Dr. Fuzz [2].

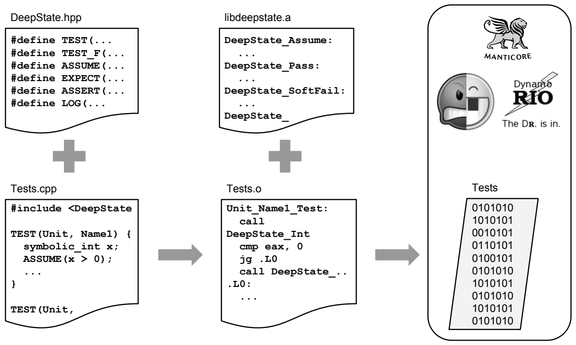
DeepState also targets the same space as property-based testing tools such as QuickCheck [10], ScalaCheck [28], Hypothesis [26], and TSTL [17], [20], but DeepState's test harnesses look like C/C++ unit tests. The major difference from previous tools is that DeepState aims to provide a front-end that can make use of a growing variety of back-end methods for test generation, including (already) multiple binary analysis engines and a non-symbolic fuzzer. Developers who write tests using DeepState can expect that DeepState will let them, without rewriting their tests, make use of new binary analysis or fuzzing advances. The harness/test definition remains the same, but the method(s) used to generate tests may change over time. In contrast, most property-based tools only provide random testing, and symbolic execution based approaches such as Pex [37], [39] or KLEE [8], while similar on the surface in some ways, always have a single back-end for test generation. Even a system such as TSTL, which aims to provide a common interface [14] to different testing methods assumes that all of those methods will be written using the TSTL API and (concrete) notion of state.

In addition to letting developers find the best tool for testing their system (or find different bugs with different tools), the ability to shift back-ends effortlessly addresses a core problem of practical automated test generation. Most of the tools in wide use are research prototypes with numerous bugs. In fact, in our own early efforts to test a file system using DeepState, we discovered a show-stopping fault in angr [3]. Without DeepState, encountering such a bug means stopping testing efforts until the bug can be fixed (which can be difficult, especially if reporting a bug arising from sensitive code is at issue), or re-writing your tests (possibly after learning a new tool). With DeepState, one simply types `deepstate-manticore` instead of `deepstate-angr` to switch from angr to Manticore.

Moreover, sharing a single notion of test harness makes it possible to implement high-level test generation strategies once and thus avoid the effort of re-implementing such approaches for every tool (or, worse yet, every individual testing effort, as happens with many strategies).

In this paper we describe the design and implementation of DeepState (Section II) and show how DeepState's approach enables API-call sequence testing of a simple user mode

Workshop on Binary Analysis Research (BAR) 2018
18 February 2018, San Diego, CA, USA
ISBN 1-891562-50-9
http://dx.doi.org/10.14722/bar.2018.23009
www.ndss-symposium.org



```
#define TEST(...
#define TEST_F(...
#define ASSUME(...
#define EXPECT(...
#define ASSERT(...
#define LOG(...
```

```
DeepState_Assume:
  ...
DeepState_Pass:
  ...
DeepState_SoftFail:
  ...
DeepState_
```

```
#include <DeepState

TEST(Unit, Name1) {
  symbolic_int x;
  ASSUME(x > 0);
  ...
}

TEST(Unit,
```

```
Unit_Name1_Test:
  call
DeepState_Int
  cmp eax, 0
  jg .L0
  call DeepState_..
.L0:
  ...
```

```
0101010
1010101
0010101
0110101
0100101
0101010
1010101
0101010
1010101
0101010
```

Fig. 1. The DeepState workflow mirrors that of Google Test. The developer includes the DeepState API header file (DeepState.hpp) into their testing code (Tests.cpp), and uses the various macros, such as `TEST`, to organize the code to be tested. The compiled tests (Tests.o) are linked against the DeepState runtime (libdeepstate.a), producing a test suite binary (Tests). This binary can be run independently, or under the control of programs like `deepstate-angr`.

```
bool IsPrime(const unsigned p) {
  for (unsigned i = 2; i <= (p/2); ++i) {
    if (!(p % i)) {
      return false;
    }
  }
  return true;
}

TEST(PrimePolynomial, OnlyGeneratesPrimes) {
  symbolic_unsigned x, y, z;
  ASSUME_GT(x, 0);
  unsigned poly = (x * x) + x + 41;
  ASSUME_GT(y, 1);
  ASSUME_GT(z, 1);
  ASSUME_LT(y, poly);
  ASSUME_LT(z, poly);
  ASSERT_NE(poly, y * z)
      << x << "^2 + " << x << " + 41 is not prime";
  ASSERT(IsPrime(Pump(poly)))
      << x << "^2 + " << x << " + 41 is not prime";
}

int main(int argc, char *argv[]) {
  DeepState_InitOptions(argc, argv);
  return DeepState_Run();
}
```

Fig. 2. A simple test case that asserts that every number generated by the polynomial $x^2 + x + 41$ is prime. The `Pump` function (described in Section IV-A) "pre-forks" the executor on possible assignments to `poly`, thereby enabling many state forks to simultaneously execute `IsPrime` at native speeds on many different *concrete* assignments to `poly`.

filesystem (Section III). Finally, we show how the DeepState approach enables novel solutions to scalability challenges in symbolic execution (Section IV).

II. Implementation

A. Architecture

DeepState is made of two components: i) a static library linked into every test suite (Figure 1), and ii) the executor, which is a lightweight orchestrator around symbolic execution engines like Manticore and angr, or fuzzing engines like Dr. Fuzz. Adding new back-ends involves producing a new executor, but does not require changes to the static library.

1) Library: The DeepState library is approximately 1200 lines of C, with an additional 600 lines of C++ which wrap the core C APIs and provide the conveniences of a Google Test-like system. The majority of the DeepState library code is unrelated to the core task of testing. In fact, a lot of complexity in DeepState relates to ensuring that the common practice of `printf`-like logging inside of unit tests does not degrade performance.

2) Executor: The orchestration engine is approximately 350 lines of Python, with an additional 250 lines each for enabling Manticore and angr support. DeepState uses APIs already provided by angr and Manticore to locate and hook key APIs and find all test cases to run. In the case of Dr. Fuzz, a special test case harness function is defined, `DrMemFuzzFunc`, which is Dr. Fuzz's standard "entry-point" into a code-base and the source of mutated input bytes. Eventually we plan to integrate libFuzzer [32] (and perhaps other fuzzers) in a similar way.

This section describes the implementation of DeepState, using a primality test as a running example (Figure 2). The test asserts that Euler's famous prime-generating polynomial [31] , $x^2 + x + 41$, only generates prime numbers.

A DeepState-based test suite is an ordinary C or C++ program that uses C preprocessor macros and APIs provided by the DeepState library to define and run tests. The key elements of a DeepState-based test suite are the definition of tests and test fixtures, the sources of symbolic values, and the pre- and post-conditions that constrain the properties to be tested. Figure 2 shows how these basic components combine into a unit test.

B. Tests

Unit tests are functions that are defined using the special `TEST` macro. This macro denotes the unit name (`PrimePolynomial`) and the test name (`OnlyGeneratesPrimes`) accordingly. Contained within the typical body of a `TEST`-defined function is code that sets up the environment, executes the functions to be tested, and asserts any pre- or post-conditions that must hold.

The `TEST` macro wraps the test code in a function that calls the `DeepState_Pass` function as its last action. The macro also ensures that the wrapped function is registered at runtime into a global linked list of all test cases. At runtime, the executor simulates the program's normal execution (including test registration) up until the call to `DeepState_Run`. At this point, the executor takes control, and forks execution for each registered test. The executor will exhaustively explore the state space of each test case, up until the execution of one of the `DeepState_Pass`, `DeepState_Fail`, or `DeepState_Abandon` functions is called.

1) Test Fixtures: Groups of unit tests that share complex state initialization requirements can be organized into test fixtures, which are C++ classes that package together shared state and functionality. Fixtures provide the developer with the ability to perform common setup and tear-down of the shared state around each unit test. Unit tests based on fixtures are defined with the TEST_F macro.

C. Symbols

Symbolic variables are defined using type names that mirror their C counterparts, such as symbolic_int and symbolic_char. These types are convenient names that are implemented using the more generic Symbolic class template. With this template, test cases can create more involved data structures, such as symbolic strings and sequences.

Under the hood, symbolic types are syntactic sugar wrapping DeepState's core C API. For example, the declarations of x, y, and z in Figure 2 are equivalent to the following code:

```
// symbolic_unsigned x, y, z;
unsigned x = DeepState_UInt();
unsigned y = DeepState_UInt();
unsigned z = DeepState_UInt();
```

When run using angr or Manticore, each call to DeepState_UInt returns a fresh unsigned integer symbol. This is implemented by building up an integer from a global array of bytes (shown below). The executor initializes these bytes with unconstrained symbol when the DeepState_Run is executed. Deferring the overwriting of the bytes ensures that the test suite startup is deterministic and that no state forking happens before the call to DeepState_Run.

```
unsigned DeepState_UInt(void) {
  unsigned index = DeepState_InputIndex;
  unsigned byte0 = DeepState_Input[index++];
  unsigned byte1 = DeepState_Input[index++];
  unsigned byte2 = DeepState_Input[index++];
  unsigned byte3 = DeepState_Input[index++];
  DeepState_InputIndex = index;
  return (byte0 <<  0) | (byte1 << 8) |
         (byte2 << 16) | (byte3 << 24)
}
```

D. Preconditions

Developers assert preconditions using the ASSUME family of macros. For example, in Figure 2, ASSUME_GT(x, 0) tells the execution engine that the condition $x > 0$ must hold for the remainder of the execution. This macro is implemented as a thin wrapper around a regular if statement, similar to the code shown below. This formulation relies on the symbolic execution engine to fork and explore both branches of the if statement.

```
// ASSUME_GT(x, 0);
if (!(x > 0)) {
  DeepState_Abandon("Failed assumption x > 0");
  // Never reached.
}

// Assumption holds during rest of execution.
```

If the assumption does not hold, then the DeepState_Abandon function will be executed. This function is hooked by the executor and terminates symbolic exploration of the state. When the test suite is run natively or with Dr. Fuzz, this function executes a longjmp to exit the current test.

E. Postconditions

Developers assert postconditions using either the ASSERT or CHECK family of macros. ASSERT and CHECK are implemented in a similar way to ASSUME; however, internally they make use of the DeepState_Fail and DeepState_SoftFail functions, respectively. DeepState_Fail stops execution of the state and marks the test as having failed. DeepState_SoftFail marks the test as failing but does not stop the execution.

```
// ASSERT_GT(x, 0);
if (!(x > 0)) {
  DeepState_Fail();
  // Never reached.
}

// CHECK_GT(x, 0);
if (!(x > 0)) {
  DeepState_SoftFail();
  // Execution continues, but test will fail.
}
```

F. Logging

Log messages are essential to being able to quickly diagnose the results of a failed test. To that end, DeepState supports C++-style log streams as an extension of the macro families like ASSUME, ASSERT, and CHECK, as well as via LOG macros which permit the categorization of log messages into levels (e.g. LOG(INFO), LOG(WARNING), *etc.*).

A key challenge is that logging, including printf-based logging, can introduce "unintentional" forking of symbolic states. For example, logging a symbolic integer x with printf("%d", x) or LOG(INFO) << x; can result in the executor forking to explore paths for each decimal digit and sign (e.g. 0, 1, -1, 10, -10, 100, *etc.*).

DeepState solves this challenge by implementing its C++-style logging and reimplementing standard library functions like printf in terms of special streaming APIs. The following example shows the sequence streaming API functions invoked internally by the DeepState library during a call to printf or LOG(INFO):

```
// printf("Name: %s, Age: %d", name, age);
DeepState_StreamString("%s", "Name: ");
DeepState_StreamString("%s", name);
DeepState_StreamString("%s", ", Age: ");
DeepState_StreamInt("%d", "<I", &age);
DeepState_LogStream(DeepState_LogInfo);

// LOG(INFO) << "Hello " << name;
DeepState_StreamString("%s", "Hello ");
DeepState_StreamString("%s", name);
DeepState_LogStream(DeepState_LogInfo);
```

The executor hooks the streaming APIs and buffers the streamed outputs, deferring actual concretization and formatting of logged symbolic values until the test case ends. This results in "logical" logging: logging only happens when a program state reaches the point where a unit test passes or fails, and a user monitoring the output of the executor is not bombarded with out-of-order log output that is subject to the state scheduling whims of the symbolic executor.

III. File System Example

One goal of DeepState is to make it possible to easily use the same tool to perform both "unit-test" like testing (as above) and API/library random testing [15], [29], [20], where a test is a sequence of method/function calls chosen by the tool, not the developer. To assess the feasibility of using DeepState in this way, we created API tests for TestFS [36], a user-level implementation of an ext3-like filesystem. TestFS stores a complete file system image in a file on disk. This file contains the file system metadata (e.g. super block, allocation bitmaps, inode bitmaps), as well as the data itself. For our study, we augmented TestFS with the capability to operate completely in-memory. This is similar to how real test suites operate on "mock objects." In the case of DeepState, having the TestFS file system reside in-memory means that we are not limited to any one engine's support (or lack thereof) for emulating file system operations.

Figure 3 shows (part of) a harness for testing TestFS, using DeepState's `OneOf` operator. This construct, built using C++ variadic templates, allows a user to express that one of a set of code "chunks" should be executed non-deterministically, behaviorally equivalent to a sequence like:

```
if (DeepState_Bool()) {
  ... // Chunk 1.
} else if (DeepState_Bool() {
  ... // Chunk 2.
} else {
  ... // Chunk 3.
}
```

Using `OneOf` lets DeepState automatically transform this into a `switch` table construct, so that early options are not more probable when using a fuzzer rather than a symbolic execution engine, and the branching structure is appropriately flat. It also lets us automatically apply swarm testing [19], where some options in a "do one of these" construction are omitted to improve testing. That is, instead of running the engine on a single "version" of the program where all the options in the `OneOf` are present, we can generate (either randomly or exhaustively) subsets of a few options at a time. For example, we can generate $\frac{N}{2}$ at a time where N is the total number of possible choices, so in the file system example we might give the engine a version of the harness to explore where only path creation, `open`, and `write` are available (but not `close` and `mkdir`). This sacrifices exploring all combinations, but can make fuzzing more successful (due to changing probability distributions) and improve scalability of symbolic execution, since fewer options means less complex constraints [4]. The tradeoff is often a good one, since many faults do not require all options in a "one of" construct, or even need to avoid certain options [19], [18], [5], [4]. Note that `OneOf` also handles non-deterministic choices among items in a string/array properly, a nice alternative to, e.g., creating a symbolic `char` and then using `ASSUME` to control the values, and also making these choices amenable to swarm testing.

```
static void MakeNewPath(char *path) {
  symbolic_unsigned l;
  ASSUME_GT(l, 0);
  ASSUME_LT(l, PATH_LEN+1);
  int i, max_i = Pump(l);
  for (i = 0; i < max_i; i++) {
    path[i] = OneOf("aAbB/.");
  }
  path[i] = ’\0’;
}

...

TEST(TestFs, FilesDirs) {
  InitFileOperations();
  CreateEmptyFileSystem();

  ...

  char paths[NUM_PATHS][PATH_LEN+1] = {};
  bool used[NUM_PATHS] = {};
  char data[DATA_LEN+1] = {};
  int fds[NUM_FDS] = {};
  int fd, path = -1;
  for (int i = 0; i < NUM_FDS; i++) {
    fds[i] = -1;
  }

  for (int n = 0; n < LENGTH; n++) {
    OneOf(
      [n, &path, &paths, &used] {
        path = GetPath();
        ASSUME(!used[path]);
        MakeNewPath(paths[path]);
        printf("%d: paths[%d] = %s",
               n, path, paths[path]);
        used[path] = true;
      },
      [n, &path, &paths, &used] {
        path = GetPath();
        ASSUME(used[path]);
        ASSUME_GT(strlen(paths[path]), 0);
        printf("%d: Mkdir(%s)",
               n, paths[path]);
        fs_mkdir(paths[path]);
        used[path] = false;
      },
      [n, &fd, &fds, &path, &paths, &used] {
        fd = GetFD();
        path = GetPath();
        ASSUME(used[path]);
        ASSUME_EQ(fds[fd], -1);
        ASSUME_NE(strlen(paths[path]), 0);
        printf("%d: fds[%d] = open(%s)",
               n, fd, paths[path]);
        fds[fd] = fs_open(paths[path],
                          O_CREAT|O_TRUNC);
        used[path] = false;
      },
      [n, &fd, &fds, &data] {
        MakeNewData(data);
        fd = GetFD();
        ASSUME_NE(fds[fd], -1);
        printf("%d: write(fds[%d],\"%s\")",
               n, fd, data);
        fs_write(fds[fd], data, strlen(data));
      },
      [n, &fd, &fds] {
        ASSUME_NE(fds[fd], -1);
        printf("%d: close(fds[%d])", n, fd);
        fs_close(fds[fd]);
        fds[fd] = -1;
      });
  }
}
```

Fig. 3. (Part of) the test harness for the file system

Figure 3 mixes two styles of API-call sequence testing data generation. Some input parameters, such as the data to write, are generated in the traditional way, on the fly at each function

```
template <typename T>
static T Pump(T sym, unsigned max=100) {
  for (auto i = 0U; i < max - 1; ++i) {
    T min_val = Minimize(sym);
    if (sym == min_val) {  // Will fork.
      return min_val;
    }
  }
  return Minimize(sym);
}
```

Fig. 4. Approximate implementation of `Pump`. `Minimize` is hooked by the symbolic engine to return the minimum value that satisfies the constraints of its input symbol.

call. The set of pathnames, however, is stored in an array, and there is a "memory" of previously used paths, a style of test construction known as pool-based [6], [29], [20]. While using pools can result in generating data that is not used, it makes it easier for symbolic execution engines and (especially) fuzzers to re-use a pathname (which is extremely important in most file system testing [15]).

IV. Beyond Unit Testing: Features to Support Scaling Test Generation

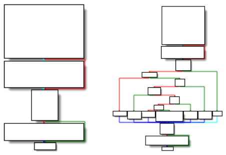
Scaling symbolic execution to operate on "real" programs is challenging. In some ways, developers are the adversary of symbolic execution engines: they continually find ways to write hard-to-analyze code. DeepState changes this dynamic by providing developers with APIs that help them guide symbolic execution without expert knowledge.

A. Pumping

Pumping is a novel scaling strategy, available via the `Pump` function, introduced by DeepState. The key insight is that loops that are bounded by a symbolic upper range, or array accesses that use symbolic indices, can be "pre-forked" to mitigate the state explosion problem.

In Euler's prime polynomial unit test (Figure 2), the `IsPrime` predicate uses a loop to check if any number in the closed range $[2, \frac{\texttt{poly}}{2}]$ evenly divides `poly`. Given a symbolic input number, `IsPrime` will induce a symbolic executor fork twice for each iteration of its loop, and twice for each divisibility check. Ideally, we would like the `IsPrime` predicate to execute "natively" and at full speed on concrete values. To achieve this and avoid state explosion, the test invokes `Pump`, which will internally produce multiple state forks, and in each such fork, return a concrete value that is a satisfying assignment for the input symbol.

Pumping strikes a good compromise between exhaustiveness and scalability. In its current form (Figure 4), `Pump` is clearly not exhaustive; it only generates the first `max` concrete values for `sym`. Other strategies can be implemented, e.g. `MaxPump`, `MinMaxPump`, *etc.* What strategy is appropriate requires a case-by-case decision by the developer writing the test. This developer is also likely to be the person best suited toward making this decision.



```
TEST(Multiplication, IsInvertible) {
  symbolic_int x, y;
  ASSUME_NE(y, 0);
  ASSERT_EQ(x, (x / y) * y);
}
```

Fig. 5. Before and after DeepState's transformation on equality comparisons. The integer comparison of `ASSERT_EQ` is decomposed into byte-wise comparisons.

B. Code Coverage

One challenge with applying coverage-guided fuzzers like Dr. Fuzz to small unit tests can be the lack of branches available for progress measurement. DeepState resolves this by incorporating known useful ideas from other tools, and lifting them to a level where they can be applied whenever appropriate, even if the back-end lacks native support for the strategy. For example, Steelix [25] and some other fuzzers [24] can decompose integer comparisons—which can be hard for fuzzers to bypass due to the mutators having to "guess" the correct value—into byte, nibble, or even bitwise comparisons, in order to produce more visibility for coverage-guided fuzzers (a kind of coarse-grained branch distance [7]). Figure 5 shows how DeepState automatically applies decomposition to `ASSERT_EQ`, `ASSERT_NE`, `CHECK_EQ`, and `CHECK_NE`.

V. Related Work

DeepState is another entry in a relatively new line of tools, which extend widely used programming languages (here C and C++) with constructs to support advanced automated test generation, whether by symbolic execution, model checking, or random testing. These tools go beyond the approach of unit testing frameworks such as JUnit [12] and Google Test [1], in that rather than simply providing language libraries or extensions to help users *write* tests, they extend the language to help users write harnesses that generate tests. Such tools can be divided into property-based tools and tools more similar (on the surface at least) to JUnit, which would include DeepState.

The property-based tools, following on QuickCheck [10] (e.g., PropEr [30], Hypothesis [26] and ScalaCheck [28]), are usually based on some form of random data generation, without symbolic execution or exhaustive exploration. Property-based approaches have been most widely adopted in functional languages.

Among the more "JUnit-like" tools are Pex/IntelliTest [37], UDITA [13], and, in a sense, all model checkers that use the language of the software under test to define the test harness, such as CBMC [23] and Java PathFinder [40]. A related but "inside-out" approach is taken by SPIN [22] (when it is used to model check C code [21]) and by TSTL [17], where the language of the tested system is embedded in a special-purpose language for defining tests and specifications. All of these tools, and DeepState, share the goal of lowering the (often considerable) barriers to the use of automated test generation. More broadly, all of these tools, to varying degrees, are domain-specific languages (DSLs) [11] for testing, usually embedded DSLs where the DSL is an extension of an existing language. DeepState is unique in both back-ends used and novel testing idioms it introduces, including ones that help with scaling for the back-end engines, and ones that increase the scope of *kinds* of testing easily performed.

DeepState in particular, rather than simply generally aiming to provide language constructs easing the use of automated test generation, is primarily targeted towards producing *parameterized unit tests* [38]. The core idea of parameterized unit tests, proposed by Schulte and Tillman [38] and implemented by them in the Unit Meister tool [39] is to take the industry practice of writing "closed" unit tests and "open tests up" by allowing concrete values to be generated that complete tests that take input parameters. DeepState differs in that it is not tied to the .NET framework, in targeting a wider variety of generation methods (fuzzing and static symbolic execution in addition to dynamic symbolic execution), and in the addition of constructs that also support the development of call-sequence random testers [15], [29], not just unit tests. The focus on generality (in both kinds of testing supported, not just unit-like tests, and in back ends) is key to DeepState's eventual goal of providing a general framework for working developers (especially those in security-critical efforts): a universal tool for automated test generation [14], [16], where the generation technology is a detail, not the focus.

VI. Conclusions and Future Work

DeepState aims to be a "one-stop shop" for automated test generation for C and C++ developers, with an approach designed to be as easy to learn as possible for experienced users of unit testing frameworks, especially Google Test. In addition to parameterized unit tests, DeepState also provides constructs to make it easy to write tests that generate API-call sequences to stateful test library-like code.

In the long run, the most important future work is to improve the communication between DeepState and the back-ends it already supports, and to add new back ends for test generation, e.g. adding dynamic symbolic analysis tools [8], and more fuzzers [41], [25]. We also expect that practical use of DeepState will motivate adding new constructs, both "syntactic sugar" like `OneOf`, to increase ease of use, and "strategies" such as `Pump` that can help scale test generation.

DeepState, in fact, should be useful to the research community, as well as developers, because it provides a convenient way to implement high-level strategies (e.g., testing methods that use multiple back-ends or even kinds of back-end, such as two-stage testing [42], or heuristics that generate concrete values). Moreover, DeepState should make it easy to compare the effectiveness of back-ends for solving the same state exploration problem, without the threat of using multiple, subtly different, test harnesses.

DeepState is available under an Apache 2.0 license at https://github.com/trailofbits/deepstate.

References

[1] "Google Test," https://github.com/google/googletest, 2008.

[2] "Dr. Fuzz: Dynamic fuzz testing extension," http://drmemory.org/docs/page_drfuzz.html, 2015.

[3] "Memory access fault," https://github.com/angr/angr/issues/798, December 2017.

[4] M. A. Alipour and A. Groce, "Bounded model checking and feature omission diversity," in *International Workshop on Constraints in Formal Verification*, 2011.

[5] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 70–81. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931056

[6] J. Andrews, Y. R. Zhang, and A. Groce, "Comparing automated unit testing strategies," Department of Computer Science, University of Western Ontario, Tech. Rep. 736, December 2010.

[7] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability*, vol. 23, no. 2, pp. 119–147, 2013.

[8] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Operating System Design and Implementation*, 2008, pp. 209–224.

[9] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 1, p. 2, 2012.

[10] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *ICFP*, 2000, pp. 268–279.

[11] M. Fowler, *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[12] E. Gamma and K. Beck, "JUnit 5," http://junit.org/junit5/.

[13] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *International Conference on Software Engineering*, 2010, pp. 225–234.

[14] A. Groce and M. Erwig, "Finding common ground: Choose, assert, and assume," in *International Workshop on Dynamic Analysis*, 2012, pp. 12–17.

[15] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *International Conference on Software Engineering*, 2007, pp. 621–631.

[16] A. Groce and R. Joshi, "Random testing and model checking: Building a common framework for nondeterministic exploration," in *Workshop on Dynamic Analysis*, 2008, pp. 22–28.

[17] A. Groce and J. Pinto, "A little language for testing," in *NASA Formal Methods Symposium*, 2015, pp. 204–218.

[18] A. Groce, C. Zhang, M. A. Alipour, E. Eide, Y. Chen, and J. Regehr, "Help, help, I'm being suppressed! the significance of suppressors in software testing," in *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, 2013, pp. 390–399.

[19] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *International Symposium on Software Testing and Analysis*, 2012, pp. 78–88.

[20] J. Holmes, A. Groce, J. Pinto, P. Mittal, P. Azimi, K. Kellar, and J. O'Brien, "TSTL: the template scripting testing language," *International Journal on Software Tools for Technology Transfer*, 2017, accepted for publication.

[21] G. Holzmann, R. Joshi, and A. Groce, "Model driven code checking," *Automated Software Engineering*, vol. 15, no. 3–4, pp. 283–297, 2008.

[22] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[23] D. Kroening, E. M. Clarke, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2004, pp. 168–176.

[24] lafintel, "Circumventing fuzzing roadblocks with compiler transformations," August 2016.

[25] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 627–637.

[26] D. R. MacIver, “Hypothesis: Test faster, fix more,” http://hypothesis.works/, March 2013.

[27] M. Mossberg, https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/, April 2017.

[28] R. Nilsson, S. Auckland, M. Sumner, and S. Sahayam, “ScalaCheck user guide,” https://github.com/rickynils/scalacheck/blob/master/doc/UserGuide.md, September 2016.

[29] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *International Conference on Software Engineering*, 2007, pp. 75–84.

[30] M. Papadakis and K. Sagonas, “A PropEr integration of types and function specifications with property-based testing,” in *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*. New York, NY: ACM Press, Sep. 2011, pp. 39–50.

[31] P. Ribenboim, “Eulers famous prime generating polynomial and the class number of imaginary quadratic fields,” *My Numbers, My Friends: Popular Lectures on Number Theory*, pp. 91–111, 2000.

[32] K. Serebryany, “Continuous fuzzing with libfuzzer and addresssanitizer,” in *Cybersecurity, Development (SecDev), IEEE*. IEEE, 2016, pp. 157–157.

[33] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware,” in *NDSS*, 2015.

[34] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “Sok: (state of) the art of war: Offensive techniques in binary analysis,” in *IEEE Symposium on Security and Privacy*, 2016.

[35] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” in *NDSS*, 2016.

[36] J. Sun, D. Fryer, A. Goel, and A. D. Brown, “Using declarative invariants for protecting file-system integrity,” in *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. ACM, 2011, p. 6.

[37] N. Tillmann and J. De Halleux, “Pex–white box test generation for .NET,” in *Tests and Proofs*, 2008, pp. 134–153.

[38] N. Tillmann and W. 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*, 2005, pp. 253–262.

[39] ——, “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*, 2005, pp. 241–244.

[40] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, “Model checking programs,” *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, Apr. 2003.

[41] M. Zalewski, “american fuzzy lop (2.35b),” http://lcamtuf.coredump.cx/afl/, November 2014.

[42] C. Zhang, A. Groce, and M. A. Alipour, “Using test case reduction and prioritization to improve symbolic execution,” in *International Symposium on Software Testing and Analysis*, 2014, pp. 160–170.