# Fuzz Testing the Compiled Code in R Packages

Akhila Chowdary Kolla
Northern Arizona University, USA
ak2296@nau.edu

Alex Groce
Northern Arizona University, USA
Alex.Groce@nau.edu

Toby Dylan Hocking
Northern Arizona University, USA
tdhock5@gmail.com

2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE) | 978-1-6654-2587-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSRE52982.2021.00040

***Abstract*—R packages written in the widely used Rcpp framework are typically tested using expected input/output pairs that are manually coded by package developers. These manually written tests are validated under various CRAN checks, using both static and dynamic analysis. Such manually written tests allow for subtle bugs, since they do not anticipate all possible inputs and miss important code paths. Fuzzers pass random, unexpected, potentially invalid inputs to a function, in order to identify bugs missed by manually written tests.**

**This paper presents RcppDeepState, an R package that uses the DeepState framework to provide automatic fuzzing and symbolic execution for R packages written using the Rcpp framework. Using RcppDeepState, a package developer can systematically fuzz test their Rcpp functions, without having to manually write any inputs nor expected outputs. Randomly generated inputs are passed to each Rcpp function, and Valgrind is used to check for various memory access violations and memory leaks. In our system, a test harness can be used to fuzz test an Rcpp function using different backend fuzzers including afl, libFuzzer, and HonggFuzz. For even more flexibility, R package developers can write their own random generation functions and assertions.**

**We implemented random generation functions for 8 of the most common Rcpp data types, then used these functions to fuzz test 1,185 Rcpp packages. Valgrind reported issues for more than 2,000 functions (over nearly 500 packages) which were not detected using standard CRAN checks on manually specified test/example inputs. Developers confirmed for several of these issues that the problem was reproducible and represented missing or flawed code. These results suggest that RcppDeepState is useful for finding subtle flaws in Rcpp packages.**

***Index Terms*—fuzzing, R language, C++ libraries, automated test generation, statistical software, memory errors**

## I. Introduction

Effective testing is important for all real-world software systems, but is especially important for software libraries, since library code is often used by many other software systems. Those systems are unlikely to perform their own testing of library code they use, and any bugs in the libraries may cause problems for client code. Because developers understand their own code better than library code they simply make use of, this also leads to hard-to-debug problems.

Most software testing, including for reused libraries, remains rooted in manual testing. Developers of libraries write tests that check whether, given certain concrete inputs chosen by the developer (or a test engineer, for large enough projects) the software produces expected outputs. Essentially, for many libraries used in mathematics, statistics, and scientific computing, the basic approach is one of specifying a fixed set of input/output pairs.

This is basically the use of non-parameterized unit tests. Parameterized unit tests [1] in contrast apply the ideas of property-based testing [2], [3] to generalize a test so that inputs are *generated* and then the behavior of such *arbitrary* inputs is checked. A key problem in parameterized unit testing or property-based testing is the development of a specification that can check correctness of arbitrary inputs values. In the case of mathematical or scientific code, where no easy method for checking results on arbitrary values may exist, this can be an especially difficult challenge.

However, drawing from the large body of work on fuzzing, one important kind of software failure can be identified without a full specification. First, users of libraries generally expect that code will return an error code rather than crashing, given invalid inputs. Thus, crashing on any input can often be seen as a fault. Second, and perhaps more importantly, illegal memory accesses and memory leaks, as well as other kinds of *undefined behavior* in C and C++ programs, can be automatically identified. Library code, no matter what input it is given, should generally not write to memory it is not supposed to write to, or depend on uninitialized memory values. Such bugs, even if they do not immediately result in incorrect outputs (and so might be missed by most manual testing in practice, even if the triggering inputs were identified), can lead to particularly devastating problems, where immediate results of the bad call are correct, but memory has been corrupted, so values later computed (including by code without bugs) can be incorrect.

Memory leaks may not lead to incorrect results, but in large-scale scientific applications, a memory leak in a library

2332-6549/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSRE52982.2021.00040



Authorized licensed use limited to: Universite De Sherbrooke. Downloaded on August 26,2024 at 03:17:33 UTC from IEEE Xplore. Restrictions apply.

function can make it impossible to carry out needed work.

### A. R and Rcpp

The R programming language is a free software environment for statistical computing and graphics [4]. It is widely used among statisticians and data miners for developing statistical software and data analysis. An R package is the fundamental unit of shareable code in the R system [5]. An R package may include tests that are manually written by the package author, and which validate the functional requirements of that package. The standard distribution mechanism for R packages is the CRAN (Comprehensive R Archive Network). It is a network of FTP and web servers around the world that stores identical, up-to-date, versions of code and documentation for R packages. As of Jan 2021, there are around 10,000 R packages available through CRAN.

Although R is very easy to use it is not the fastest or most memory efficient language. R is a garbage-collected system, with the inefficiencies sometimes introduced in managed-memory settings. For this reason, many widely used R libraries are actually largely written in C++, a lower-level, faster, and less safe language. Rcpp is a an R package that facilities extending R with C++ [6], [7], [8]. Rcpp makes it very simple to connect C++ to R, by providing C++ versions of all the data types provided by R. Thousands of R packages make use of Rcpp, including widely used packages critical to various scientific efforts. Use of Rcpp is documented in many R tutorials and books covering advanced usage of R [9].

Because Rcpp-based packages use the C++ language, which allows for many memory-safety violations and undefined behaviors in compiling code, often with subtle and hard-to-predict results, Rcpp introduces the potential for bugs that can cause frustration to developers in the form of hard-to-understand crashes or, in the worst case, incorrect results in important scientific or other applications of R. At heart, the manual memory management that is a feature of C++ is also the largest danger to library-developers, and, perhaps more importantly, library *users*, relying on C++ to make R more efficient.

### B. DeepState

As noted above, parameterized unit testing generalizes unit testing by allowing the generation of input values, rather than relying on a small set of provided values. The most promising approach to generating bug-inducing inputs to software in recent years has been fuzzing [10]. Fuzzing is based on the generation of randomized input data, and is, essentially, an extension of random testing, which has been practiced since the earliest days of software testing. Most fuzzers go beyond pure random testing by incorporating some form of genetic-algorithm-like feedback, where inputs that are deemed interesting (due to, e.g., producing novel code coverage) are mutated to find further interesting inputs.

There are many fuzzing tools, including afl [11], libFuzzer (https://llvm.org/docs/LibFuzzer.html), Eclipser [12], and HonggFuzz (https://github.com/google/honggfuzz).

Learning to use each of these tools, and guessing which one(s) will work best, is difficult. Moreover, fuzzers tend to expect that software under test takes as input either a file or an arbitrary byte stream, and provide little or (usually) no support for testing library functions that take, e.g., multiple parameters of varying types.

DeepState [13] is a front-end that allows tests to be written in a format similar to the widely used Google Test unit testing framework, but with parameterization. DeepState then allows users to generate input data using a wide variety of fuzzers, including afl, libFuzzer, Eclipser, and Honggfuzz. DeepState also includes a very fast brute-force fuzzer that does not use feedback, but has almost no overhead for input data generation. DeepState also allows users to apply symbolic execution tools, such as Manticore [14], again without changing their *test harness*.

A test harness [15] is another name for the parameterized unit test: code that controls the types and constraints on input values, and the expected test results. A unit test is a particularly simple form of test harness, and a parameterized unit test follows a *choose/assume/assert* framework [15] that is applicable to a wide variety of test generation tools.

DeepState does not automate the generation of test harnesses/parameterized unit tests. This is left to developers, and while DeepState provides an API to generate most base types in C/C++, and to orchestrate choosing among code fragments to execute, writing generators for more complex input types, such as those provided by Rcpp, is a substantial burden on users. Moreover, using DeepState requires users to drop out of the R development environment they are likely most familiar with.

### C. RcppDeepState

In this paper we propose a library and tool, **RcppDeepState**. RcppDeepState (1) provides generators for 8 of the most commonly used Rcpp data types, (2) automatically produces a DeepState harness, including some automated assertions, and (3) orchestrates using various fuzzers to generate data for the generated harness. RcppDeepState gives R library developers using Rcpp a push-button solution to fuzzing their code.

*1) Valgrind and RcppDeepState:* While library developers can extend the RcppDeepState-generated harnesses with their own custom assertions, it is essential to give developers an easy way to look for the most common C++ memory problems, which are perhaps the largest danger of using Rcpp. Valgrind [16], [17], [18] is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs. RcppDeepState lets users run their fuzzing (or tests generated by fuzzing without Valgrind) under Valgrind, to detect memory problems that go beyond simple crashes.

### D. Contributions

We applied RcppDeepState to a large number of Rcpp packages in the CRAN repository, and discovered a number of potentially dangerous memory safety issues and memory leaks. We reported these to developers, and while we await



Authorized licensed use limited to: Universite De Sherbrooke. Downloaded on August 26,2024 at 03:17:33 UTC from IEEE Xplore. Restrictions apply.

confirmation of many issues, we have received feedback from multiple developers acknowledging the problems we discovered. In addition to showing the feasibility of using RcppDeepState (we were able to apply it to a large number of packages without any special developer insight into those packages), our results shed some light on the effectiveness of various fuzzing tools and methods in the setting of R language libraries using C++ for speed and memory efficiency.

## II. Related work

We propose a new R package, RcppDeepState: an integrated fuzzing tool that supports advanced automated test generation for R packages using Rcpp to interface to C++ code. Conventionally, R packages are tested using unit tests under unit testing frameworks, including testthat [19], RUnit [20], tinytest [21], and unitizer [22]. In these frameworks, test cases and expectations are written using R code. Assertions or expectations must be manually written, and are often basically input/output pairings.

The CRAN repository also performs automatic additional analysis of code, running included tests under compiler sanitizers (e.g., clang/LLVM's ASAN) and even under Valgrind. When R packages are deployed to the CRAN repository, CRAN performs these and other checks using the manual test included.

Fuzzr [23] is another R package that fuzz tests R functions by passing in a wide array of pre-defined inputs. It does not really offer full-featured automated test generation for R code.

DeepState, and thus RcppDeepState, are inspired by the property-based testing paradigm. Property-based tools, following on QuickCheck [2], including PropEr [24], Hypothesis [3], and ScalaCheck [25], are usually based on some form of random data generation, without symbolic execution or feedback-based fuzzing. Other tools, drawing on a JUnit paradigm, but providing property-based testing, include Pex/IntelliTest [26] and UDITA [27].

## III. Implementation and Architecture of RcppDeepState

### A. The Test Harness

The most important functionality provided by RcppDeepState is the generation of a DeepState harness for testing a function. Fuzzing itself and other aspects of testing are provided via DeepState.

In order to explain what RcppDeepState provides, we will show a simple test harness generated using the tool (Figure 1).

The code begins with three includes that use C++ libraries essential to RcppDeepstate; only one of these is the RcppDeepState library itself. To integrate an external C++ file with an Rcpp application package, RcppDeepstate uses RInside [28]. To use RInside in any program first we need to create an instance of the RInside class, which represents an embedded R interpreter for a C/C++ program. The RInside package is designed to make it easier to embed R in a C++ class. Second, the RcppDeepState library itself provides data generators for commonly used R/Rcpp types. Finally, we include the DeepState library, which allows access to the DeepState core API and turns the code into a test harness by automatically adding either a `main` or a libFuzzer entry point, depending on context.

```
#include <RInside.h>
#include <RcppDeepState.h>
#include <DeepState.hpp>
IntegerVector MergeSort (IntegerVector x,
    IntegerVector y);
TEST(vectorsort, merge){
  RInside R;
  x = RcppDeepState_IntegerVector();
  y = RcppDeepState_IntegerVector();
  try {
    MergeSort(x, y);
  } catch (Rcpp::exception& e){
    cout << "Exception Handled" << endl;
  }
}
```

Fig. 1. An Example RcppDeepState-Generated Test Harness

The developer does not have to produce this code; instead the developer specifies that the function to be tested is MergeSort, which takes as input two integer vectors. RcppDeepState does the hard work, including adding an exception handler so that normal Rcpp exceptions do not cause test failure.

### B. RcppDeepState Provided Datatypes

We performed an analysis of the frequency of the datatypes used in Rcpp functions in the R packages on CRAN, in order to determine the most important generators for RcppDeepState to implement. Table I shows the frequencies of types in functions and packages.

Although SEXP and `Rcpp::List` are very frequently used datatypes, no random generation functions were implemented for these types because they are extremely dynamic and the generation of useful values for these inputs is library dependent; without further context, fuzzing these types would generate a large number of false positives. In contrast, `Rcpp::XPtr<XPtrTorchTensor>`, also highly dynamic, is present only in 1 package and 371 functions, so cannot be seen as a high priority for implementation. We implemented DeepState generators for the other most frequently used datatypes (`Rcpp::NumericVector`, `Rcpp::NumericMatrix`, `arma::mat`, `std::string`, `Rcpp::CharacterVector`, and `Rcpp::IntegerVector`). Although DeepState has built-in generators for `int` and `double` we needed to typecast them from raw byte-based C types to R/Rcpp types. Implementing generators for these 8 types allowed us to fuzz 6,860 functions over 1,185 packages. RcppDeepState generates fuzzer specific inputs only for the 8 most frequently occurring data types used in Rcpp functions in R packages. These most frequently occurring data types contribute 70% of the total tested methods from the packages on CRAN. The remaining 30% of the methods have at least one argument whose data type is not frequent or is highly dynamic (SEXP,



Authorized licensed use limited to: Universite De Sherbrooke. Downloaded on August 26,2024 at 03:17:33 UTC from IEEE Xplore. Restrictions apply.

Rcpp::List, etc.). When testing thousands of methods the probability of missing the methods with the least common data types is acceptable, considering the amount of work that we need to put into generating useful defaults for those data types.

TABLE I
DATATYPE FREQUENCIES

| Data type | # Functions | # Packages |
|---|---|---|
| `SEXP*` | 455 | 90 |
| Rcpp::NumericVector | 393 | 174 |
| `Rcpp::XPtr<XPtrTorchTensor>*` | 371 | 1 |
| Rcpp::NumericMatrix | 291 | 146 |
| arma::mat | 286 | 127 |
| `Rcpp::List*` | 269 | 86 |
| std::string | 216 | 84 |
| int | 123 | 73 |
| Rcpp::CharacterVector | 121 | 54 |
| Rcpp::IntegerVector | 99 | 43 |
| double | 89 | 50 |

### C. RcppDeepState Architecture

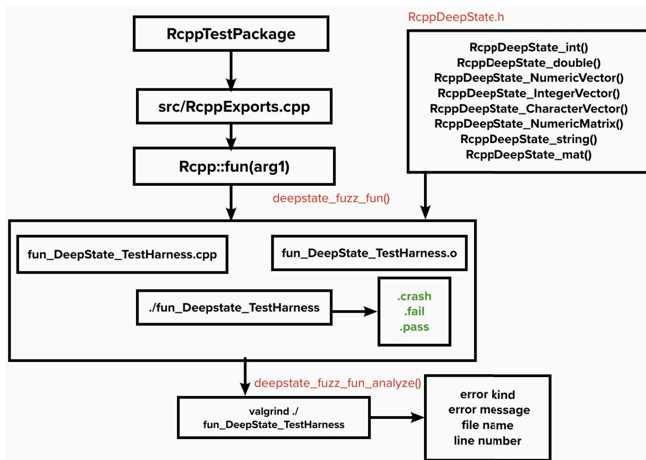


Fig. 2. RcppTestPackage Workflow When Tested Using RcppDeepState.

An Rcpp package has a structure based on a standard set of folders, namely R (containing the critical file `RcppExports.R`), man, src, and, most importantly for our purposes, standardized description and namespace files. RcppDeepState operates on this standard organization scheme as follows:

- First, `src\RcppExports.cpp` is parsed and a harness is generated for each exported function call; these are the functions visible to users of the library, so they are the functions we need to fuzz. A harness is based on a generic template, e.g.,

  $$Rcpp::fun1(datatype1, datatype2)$$

  The compiled test harness, e.g.,

  $$fun1_DeepState_TestHarness.o$$

  , is linked against the TestPackage's dynamic library (TestPackage.so), producing a fuzzable executable.
- DeepState is used to perform fuzzing, and the output files for specific fuzzers are processed to bucket results into `.pass`, `.fail`, and `.crash` files. These can also be replayed as regression tests, or as a corpus for future fuzzing.
- Valgrind is used to further analyze passing runs for memory use problems that do not cause a crash or property violation.

Figure 2 shows the basic flow, including the underlying RcppDeepState functions called, for testing a hypothetical RcppTestPackage using RcppDeepState, including generation of Valgrind traces for presentation to the user (discussed next).

### D. Valgrind Analysis

Valgrind memcheck can detect the use of uninitialized memory, reading/writing memory after it has been free'd (use-after-free), reading/writing off the end of malloc'd blocks, reading/writing inappropriate areas on the stack, and memory leaks (where pointers to malloc'd blocks are lost forever or there is mismatched use of `malloc/new/new []` vs. `free/delete/delete []`).

We run Valgrind not during fuzzing itself, which imposes too high an overhead for many fuzzers, but on the corpus of interesting tests generated by each fuzzer, in particular the `.pass` files (there is little point in using Valgrind on already-failing tests).

## IV. RCPPDEEPSTATE OVERVIEW

The Figure 2 depicts the stages that include in testing an Rcpp package using RcppDeepState.

1) **Parse & Extract:** Once the user has the source code for the Rcpp test package available in his local machine that is either downloaded from the CRAN repository or developed by the user we need to parse the data that is present in the src folder's RcppExports.cpp file. This file contains all the available Rcpp functions defined by the user along with the list of arguments and their datatypes.
2) **Type-Match Extraction:** RcppDeepState defines a test harness only for those functions that only take arguments from the specified 8 datatypes. After parsing the RcppExports.cpp file we match those captured data types and generate the test harness.
3) **Primary Fuzz Run:** The system automatically generates the test harnesses for each function in the package in the function folder in the inst/testfiles path. This test harness is stored in the function folder along with the required makefiles. The system then compiles and runs the test harness and stores the fuzzer generated inputs in the outputs folder.
4) **Secondary Fuzz Analysis:** In this step we analyze the previously generated outputs from the fuzzers. This analysis runs the test harness executable using the memory debugging tool Valgrind.



Authorized licensed use limited to: Universite De Sherbrooke. Downloaded on August 26,2024 at 03:17:33 UTC from IEEE Xplore. Restrictions apply.

5) **Result Analysis:** The Valgrind run stores the obtained results in an XML format and we parse the XML data and generate a user-readable data frame with details including the error kind, error message, and the trace showing the line number and the file name making it easier for the user to rectify the error based on the details provided.

### A. RcppDeepState Tool Demo:

In this section we will look at testing of an Rcpp function *mi* from the *BNSL* package. Consider the following function BNSL::mi where the NumericVectors x and y and int proc are the parameters for the function:

```
// [[Rcpp::export]]
double mi(NumericVector x, NumericVector y,
          int proc=0){
          if(proc==0)return(Jeffreys_mi(x,y,0,0));
          else if(proc==1)return(MDL_mi(x,y,0,0));
          else if(proc==2)return(BDeu_mi(x,y,0,0,1));
          else if(proc==3)return(empirical_mi(x,y));
          else if(proc==9)return(empirical_mi(x,y));
          else if(proc==10)return(cont_mi(x,y));
          else return(Jeffreys_mi(x,y,0,0));
}
```

The inserted code includes a proc estimation that is based on Jeffrey's prior, the MDL, BDeu, cont, and empirical principle. If the argument proc is missing, proc=0 (Jeffreys') is assumed. All Rcpp packages have documentation written by the developer which provides a short overview of the functions and exposes the semantics of the function by providing valid input examples.

```
mi> n=100
mi> x=rbinom(n,1,0.5); y=rbinom(n,1,0.5); mi(x,y)
[1] 0
mi> z=rbinom(n,1,0.1); y=(x+z); mi(x,y);
[1] 0.5204089
mi> x=rnorm(n); y=rnorm(n); mi(x,y,proc=10)
[1] 0
mi> x=rnorm(n); z=rnorm(n); y=0.9*x+sqrt(1-0.9^2)*z;
      mi(x,y,proc=10)
[1] 0.4281646
>
```

However, these (trivial) examples/tests don't explore all the possible paths of the function. In this case, the inputs only explore the 2 paths where the proc value is 0 or 10.

```
> res <- deepstate_read_valgrind_xml("mi_example.R")
> res
Empty data.table (0 rows and 5 cols): err.kind,
     message, file.line, address.msg, address.trace
```

When we run the predefined examples/tests for the mi function under Valgrind we don't see any errors. Testing the code on these predefined inputs is inadequate for claiming code is likely bug free. We need to explore all the possible paths that increase the code coverage by passing randomized or unexpected inputs to the function. To do that we need to test the function under RcppDeepState as follows:

```
> pkg.path <- "/home/user/BNSL"
> fun <- "mi"
> RcppDeepState::deepstate_fuzz_fun(pkg.path,fun,
     time.limit.seconds=3)
[1] "mi"
```

The deepstate_fuzz_fun() call results in test harness generation for the provided Rcpp function, followed by running the test harness for the provided Rcpp function. It also generates .crash, .fail, .pass extension files depending upon the type of response obtained by running the inputs on the executable.

The next step includes analyzing those generated inputs under Valgrind looking for the bugs/errors.

```
> path <- "/home/user/BNSL/inst/testfiles/mi"
> seed_analyze <- deepstate_fuzz_fun_analyze(path,
     time.limit.seconds=10)
running the executable ..
```

The deepstate_fuzz_fun_analyze() returns a data table with the inputs and the error messages and the position where the error occurred.

```
> str(seed_analyze$logtable)
List of 1
 $ :Classes data.table and data.frame:  1 obs. of  5
      variables:
  ..$ err.kind      : chr "InvalidRead"
  ..$ message       : chr "Invalid␣read␣of␣size␣8"
  ..$ file.line     : chr "mi_cmi.cpp:55"
  ..$ address.msg   : chr "Address␣0x9d66358␣is␣0␣
      bytes␣after␣a␣block␣of␣size␣184␣alloc'd"
  ..$ address.trace: chr NA
  ..- attr(*, ".internal.selfref")=<externalptr>
```

The output shows there was an issue in file mi_cmi.cpp at line 55. If we trace back to that function line the code combines vector x and y to produce a new table c_xy. If the size of vectors x and y are not equal the system generates an issue because we are trying a create a combined table for two unequal vectors, which causes an invalid read. The fuzzer specific datatypes produce unequal vectors exposing the invalid read which was not identified running the predefined inputs. Therefore we need to specify a condition to check if the sizes of the vectors x and y are equal.

## V. Results and Comparison of Fuzzers

We applied RcppDeepState, using only the default settings of the tool and the provided `RcppExports.R` files to fuzz a large set of CRAN packages using Rcpp. We restricted our analysis to functions where all inputs were a type supported by the RcppDeepState generators.

Our purpose was twofold: first to demonstrate that even in the absence of additional specification of correctness by developers, RcppDeepState could find memory safety problems and memory leaks. Second, we wanted to compare a few different fuzzers, to see if there was an obvious difference in performance between fuzzers.

### A. Low Budget Random Testing

In real-world fuzzing campaigns against security interfaces or compilers, it is often important to fuzz a system for hours or, often, days [**?**]. This is inconvenient for using fuzzing in a property-based testing context, or during normal development



Authorized licensed use limited to: Universite De Sherbrooke. Downloaded on August 26,2024 at 03:17:33 UTC from IEEE Xplore. Restrictions apply.

of an R package. Many property-based testing tools set a default timeout of only one minute [29], and this is a short enough budget that developers can easily perform fuzzing after every change to their code. However, feedback-based fuzzers generally require some startup time simply to calibrate which inputs to focus on, so one minute is somewhat unrealistic.

Because R developers are unlikely to be experienced in fuzzing, and are more likely to use a tool that provides results even without a substantial investment of time or attention, we therefore performed our experiments using a twenty minute fuzzing budget for each fuzzer. This is a compromise between the very short testing budgets typical of property-based testing tools and the multi-hour runs typical of security-oriented fuzzing campaigns.

### B. Fuzzers Compared

We wanted to compare fuzzing using tools likely to be available to all DeepState users on any unix-like platform, and tools widely perceived as easy to install and useful. The most well known fuzzers are afl [11] and libFuzzer, the function-based fuzzer included with recent versions of LLVM/clang. We additionally used DeepState's built-in brute-force fuzzer, which lacks coverage-driven feedback and mutation of inputs, but has extremely high throughput for test generation. We hypothesized that even brute-force fuzzing would work well for developers looking for memory safety issues in Rcpp-based packages. Because afl, unlike libFuzzer and DeepState's fuzzer, requires an initial seed corpus, we used the example values that are included in every R package with each function as initial seeds for afl in one run, in addition to the default null file corpus provided by DeepState. All experiments were performed on an Intel Xeon(E)E-2136 CPU 3.30Ghz, with 32GB of RAM, using a single core (the fuzzers are all essentially single-threaded so multiple cores are only useful for running multiple fuzzer instances).

### C. Experimental Results

We downloaded 2,077 Rcpp packages from CRAN. Of these, only 1,185 packages could be analyzed, due to our limited set of generators, or the absence of an `RcppExports` file. We fuzzed 6,860 functions. Of these, 965 produced at least one Valgrind warning. Table II shows overall data on our experiments.

TABLE II
SUMMARY OF RESULTS

| Type | # Packages | # Functions |
|---|---|---|
| Total Rcpp Packages | 2149 | 19735 |
| No RcppExports | 247 | NA |
| At least one unfuzzable type | 717 | 12,875 |
| Executed under RcppDeepState | 1,185 | 6,860 |
| afl (no corpus) | 5 | 8 |
| afl (corpus) | 27 | 59 |
| libFuzzer | 73 | 117 |
| DeepState Fuzzer | 478 | 911 |

The last three rows are the core result. This shows, for each fuzzer used, the number of packages/functions for which at least one problem (crash or Valgrind issue) was found. All of the fuzzers were useful, and as we later show, they identified *different* problems in *different* packages and functions. However, in terms of raw numbers, afl performed poorly using the null corpus, and much better when seeded with the data in the R package `test/example` directories. However, libFuzzer performed better overall than afl, even with a better corpus than libFuzzer. Finally, the brute force DeepState fuzzer, while probably less capable of finding complex, deep, bugs requiring discovery of deep code paths, was extremely effective for finding corner-case inputs for these functions. In a sense, the DeepState fuzzer is simply the original Miller et al. [**?**] fuzzing proposal, adapted to type-correct function input generation. We speculate that R code vulnerabilities may have shallow paths amenable to fast brute-force approaches, and not requiring large fuzzing budgets or advanced feedback, unlike more typical fuzzing targets, e.g., media parsers or web browsers. The last row shows the number of Valgrind warnings produced for the test code included with each package. None of the inputs provided as examples resulted in exposing any of the memory-safety problems.

Most of the problems we identified were Valgrind errors; afl did not identify any crashing inputs, and the brute force fuzzer identified 478 only when run under Valgrind; libFuzzer identified 378 crashes using address sanitizer in place of Valgrind, however.

TABLE III
VALGRIND ERROR COUNT

| Error Type | DeepState | afl | libFuzzer |
|---|---|---|---|
| Invalid read | 405 | 11 | 20 |
| Invalid write | 97 | 13 | 53 |
| Use of uninitialized values | 6 | 7 | 1 |
| Conditional jump or move(s) | 113 | 7 | 3 |
| Argument size | 16 | 0 | 0 |
| Possibly lost data | 274 | 34 | 16 |

Table III shows detailed data on the types of Valgrind warnings/errors, for one package. A more detailed explanation of these error categories makes it easier to understand these results:

- Invalid read: there was an attempt to read from an invalid memory address.
- Invalid write: there was an attempt to write to an invalid memory address.
- Use of uninitialised values: there was a read from an uninitialized value in memory.
- Conditional jump or move(s): a branch depends on uninitialized data.
- Argument size: an incorrect value was passed to a system call, e.g., `malloc(-1)`
- Possibly lost data: there is a potential memory leak, where allocated memory was not deallocated.

While the memory leaks may be harmless in some cases



Authorized licensed use limited to: Universite De Sherbrooke. Downloaded on August 26,2024 at 03:17:33 UTC from IEEE Xplore. Restrictions apply.

(Valgrind's detection of memory leaks can be less precise than other errors) most of the other problems indicate at least a potential for serious problems, including reliance of function values on the compiler version and previous contents of memory, or potential corruption of memory.

Figures 3 and 4 show Venn diagrams of the overlap in results between the fuzzers. Note that while the DeepState brute force fuzzer was the most successful, there were a large number of functions or packages where problems were only identified by afl and/or libFuzzer, as well. It is good to make use of all three fuzzers, which is easy to do with RcppDeepState. It is likely other fuzzers would find other problems. We did not run Eclipser, for example, which is good at finding many problems other fuzzers do not, because its approach really needs a larger time budget than twenty minutes. For final checks before submitting a package to CRAN, use of all fuzzers supported, for longer timeouts (at least an hour) would be a sound development practice. Scientific or economically important results can depend on R packages, so for final production versions, more substantial testing is in order. RcppDeepState makes that easy.

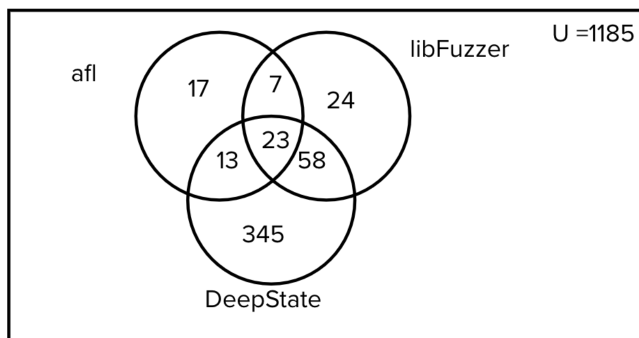


Fig. 3. A Venn diagram for the Universe(U) of 1,185 test packages, representing the count of packages that have issues identified by each of the fuzzers.

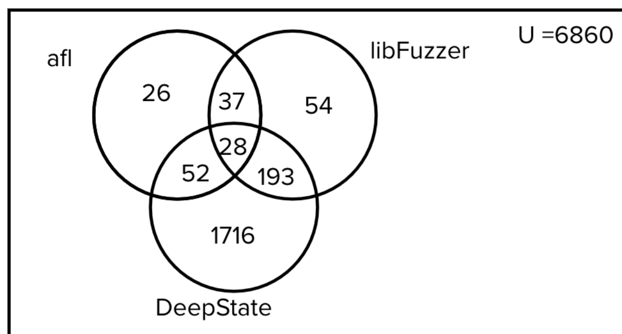


Fig. 4. A Venn diagram for the Universe(U) of 6,860 functions, representing the count of functions that have issues identified by each of the fuzzers.

Surprisingly, the difference in fuzzer performance is not obviously attributable to throughput differences. As Table IV shows, while the DeepState fuzzer was faster than other fuzzers, the difference was not large, due to the fact that RcppDeepState writes all tests to disk in order to allow offline Valgrind analysis. The need to write outputs accounts for the fact that libFuzzer, which is normally much faster than afl, was actually somewhat slower in our experiments: the advantages of function-call fuzzing are reduced in our setting. If users add custom properties and/or are only interested in actual crashes, throughput for libFuzzer and the DeepState fuzzer could be made much higher, and that of afl somewhat higher, by turning off the writing of non-failing inputs to disk.

TABLE IV
INPUTS GENERATED PER MINUTE

| Fuzzer | Mean | Median | Standard deviation |
|---|---|---|---|
| afl | 86 | 77 | 4.94 |
| libFuzzer | 79 | 63 | 5.15 |
| DeepState | 97 | 81 | 7.15 |

We contacted developers with the results of our analysis, and five developers (so far) responded to us, confirming our results. In some cases, the response suggested that the library expects users to provide valid inputs, and our fuzzing results did not satisfy such preconditions. However, given the potential harm from memory corruption if users make a mistake in calling a package function, it would be best for such checks to be added to the package code, since most seem to be simple, cheap checks (e.g., two vectors must match in size). Our harness will not report a problem if a function detects invalid inputs and gracefully raises an exception, alerting a user that the input is invalid. And, in fact, three of the developers in their initial response explicitly noted that we had identified missing important checks on data validity, that should be added to the library code.

## VI. DISCUSSION:

In this section we discuss our reasoning for the decisions we made that might call into question the validity of our approach.

### A. *Threshold Running Time:*

The selected approach of running the test harnesses for a threshold time limit of 20 minutes for each fuzzer comes with an overhead. Running the fuzzers along with valgrind for more than the chosen time limit incurs significant performance issues. R uses a large amount of memory and generates large vectors and matrices. Fuzzers are incredibly fast; they generate thousands of inputs in a short span, but the issue occurs when we are trying to serialize those inputs to R level inputs. The longer we run the fuzzers the larger the data that needs to be translated, and the larger the size of vectors to be allocated in R. On average the fuzzer generated inputs when run for more than the chosen threshold resulted in inputs of size 1.3GB which exceed the size of the vectors that could be allocated. For 61% of the packages the issues were identified under 20m in Table II. The budgets are in some ways arbitrary, and insufficient for an evaluation of the various fuzzers. However, we chose budgets that were clearly in scope for practical



Authorized licensed use limited to: Universite De Sherbrooke. Downloaded on August 26,2024 at 03:17:33 UTC from IEEE Xplore. Restrictions apply.

use during development by Rcpp developers. Shorter runs might find too little, but longer runs would discourage some developers.

### B. Exported vs Non-Exported Functions:

There are situations where the corner cases can trigger a bug inside a method exposed by Rcpp, but the method itself is effectively guarded by R code. There might be no practical case for testing those functions as these functions are not directly exposed to the end-users. Hence, we have focused our analysis in the paper on the subset of Rcpp functions that are directly callable from R, for which users can provide arbitrary inputs. Although our analysis deals with fuzz testing both exported and unexported functions (as designated by the R package's NAMESPACE API), the main focus of RcppDeepState is to handle issues that come from exported functions; mostly no run-time checks are performed by the package developers on the inputs provided to these functions.

RcppDeepState found issues in 156 exported, visible functions from 74 packages (out of 243 exported functions tested). These are clearly relevant problems because users call those Rcpp functions directly, and there is no opportunity for R-level code to protect users from providing bad inputs.

## VII. Conclusions and Future work

R is a widely used (estimates of the number of users range from 250K to over 2 million, and there are over 200K repositories using the R language on GitHub) statistical analysis language. R code drives important scientific and commercial data analysis projects. R packages written in C++, using the popular Rcpp framework allow R users to take advantage of the speed and lower-level memory management provided by C++ code. In some cases, this is essential for handling large data sets efficiently. However, using manual memory management rather than R's managed garbage collection exposes users to subtle memory safety flaws, and the possibility of memory leaks.

Rcpp-based packages are usually tested using a handful of manually devised input values. Although these tests are run under tools such as Valgrind that can expose memory safety issues, the fact that, as with most manual unit tests, the input values are ones the developer has obviously thought about when writing the code mean that in practice such tests seldom expose subtle problems.

We present RcppDeepState, a tool that automatically generates a test harness given a standard Rcpp-using R package, based on the function export information. RcppDeepState uses the DeepState testing front-end to provide push-button fuzzing, using powerful modern fuzzing tools, for developers of Rcpp-based R packages. RcppDeepState provides generators for the most commonly used Rcpp data types that are amenable to automated fuzzing. Using RcppDeepState, we analyzed 1,185 Rcpp-based packages and 6,860 functions in those packages. Our efforts exposed a large number of potential memory vulnerabilities.

As future work, we would like to mine R code, including example code included in packages, to determine implicit constraints on input values, in order to avoid some false positives (or at least mark them as likely involving invalid inputs). Such *implicit type* inference would also make it possible to effectively support widely used, but hard-to-generate, R types such as `SEXP` and `List`. We would also like to add further tools for developers to write their own custom properties to check, including algorithmic complexity checks. RcppDeepState is available at *https://github.com/akhikolla/RcppDeepState*.

## Acknowledgements

The authors thank Peter Goodman, Joe Ranweiler, Pawel Platek, and Trail of Bits for their work on development of DeepState. We also thank Dirk Eddelbuettel from the Rcpp Core Team for his assistance. This project has received funding from the R Consortium Infrastructure Steering Committee (ISC) under the Linux Foundations Projects 2020.

## References

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

[2] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000, pp. 268–279.

[3] D. R. MacIver, "Hypothesis: Test faster, fix more," 2013.

[4] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2020. [Online]. Available: https://www.R-project.org/

[5] R. C. Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2020. [Online]. Available: https://r-pkgs.org/

[6] D. Eddelbuettel and R. François, "Rcpp: Seamless R and C++ integration," *Journal of Statistical Software*, vol. 40, no. 8, pp. 1–18, 2011. [Online]. Available: https://www.jstatsoft.org/v40/i08/

[7] D. Eddelbuettel, *Seamless R and C++ Integration with Rcpp*. New York: Springer, 2013, iSBN 978-1-4614-6867-7.

[8] D. Eddelbuettel and J. J. Balamuta, "Extending extitR with extitC++: A Brief Introduction to extitRcpp," *The American Statistician*, vol. 72, no. 1, pp. 28–36, 2018. [Online]. Available: https://doi.org/10.1080/00031305.2017.1375990

[9] H. Wickham, *Advanced R*. CRC Press, 2019.

[10] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, pp. 1–13, 2018.

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

[12] J. Choi, J. Jang, C. Han, and S. K. Cha, "Grey-box concolic testing on binary code," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 736–747.

[13] P. Goodman and A. Groce, "Deepstate: Symbolic unit testing for c and c++," in *NDSS Workshop on Binary Analysis Research*, 2018.

[14] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *International Conference on Automated Software Engineering*, 2019, pp. 1186–1189.

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

[16] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[17] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, 1995.



Authorized licensed use limited to: Universite De Sherbrooke. Downloaded on August 26,2024 at 03:17:33 UTC from IEEE Xplore. Restrictions apply.

[18] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.

[19] H. Wickham, "testthat: Get started with testing," *The R Journal*, vol. 3, pp. 5–10, 2011.

[20] T. K. Matthias Burger, Klaus Juenemann, "R functions implementing a standard unit testing framework, with additional code inspection and report generation tools." *The R Journal*, 2018. [Online]. Available: https://cran.r-project.org/web/packages/RUnit/

[21] M. van der Loo, "A method for deriving information from running r code," *The R Journal*, p. Accepted for publication, 2020. [Online]. Available: https://arxiv.org/abs/2002.07472

[22] B. Gaslam, "unitizer simplifies creation, review, and debugging of tests in r," *The R Journal*, 2017. [Online]. Available: https://github.com/brodieG/unitizer

[23] M. Lincoln, ""fuzz tests" for your r functions," in *fuzztests R functions*, August 2018. [Online]. Available: https://github.com/mdlincoln/fuzzr

[24] M. Papadakis and K. Sagonas, "A proper integration of types and function specifications with property-based testing," in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, 2011, pp. 39–50.

[25] M. R.Nilsson, S.Auckland and S. Sahayam, "Scalacheck user guide," *Scala*, September 2016. [Online]. Available: https://github.com/rickynils/scalacheck/blob/master/doc/UserGuide.md

[26] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs*. Springer, 2008, pp. 134–153.

[27] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 225–234.

[28] D. Eddelbuettel, R. Francois, and L. Bachmeier, *RInside: C++ Classes to Embed R in C++ (and C) Applications*, 2020, r package version 0.2.16. [Online]. Available: https://CRAN.R-project.org/package=RInside

[29] J. Holmes, I. Ahmed, C. Brindescu, R. Gopinath, H. Zhang, and A. Groce, *Using relative lines of code to guide automated test generation for Python*, 2020. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3408896



Authorized licensed use limited to: Universite De Sherbrooke. Downloaded on August 26,2024 at 03:17:33 UTC from IEEE Xplore. Restrictions apply.