Fuzz-testing R-based research software for robustness

Marco Colombo (@mcol)

Institute of Geography, Heidelberg University

UNIVERSITÄT
HEIDELBERG
ZUKUNFT
SEIT 1386

REPLAY: Reproducible Luminescence Analyses

REPLAY is a DFG-funded programme that aims to provide a **robust**, **correct** and **accessible** platform for **luminescence data analysis** (geochronology).

Funded by

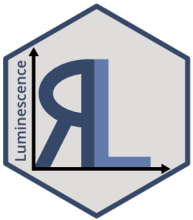
DFG Deutsche Forschungsgemeinschaft

German Research Foundation

https://replay.geog.uni-heidelberg.de

The ‘Luminescence’ package

‘Luminescence’: Comprehensive luminescence dating data analysis



A long history:

- First version on CRAN: v0.1.7 (May 2012)
- 67 releases on CRAN
- 25 developers overall

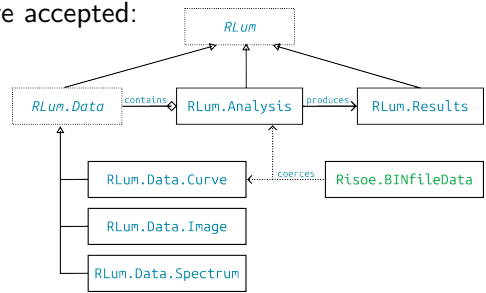
Some current stats:

- 142 exported functions
- Over 32,000 lines of R code
- 394 pages in the PDF manual

Data types accepted in ‘Luminescence’

Across all functions, the following data types are accepted:

- vectors
- matrices
- data frames
- RLum.* S4 objects
- lists containing any of the above



Each function generally supports only a subset of those data types.

User experience in 'Luminescence' I

Historically, 'Luminescence' has always tried to be helpful to users

```
data("ExampleData.RLum.Analysis")
analyse_SAR.CWOSL(IRSAR.RF.Data)
```

```
Error : [analyse_SAR.CWOSL()] No record of type 'OSL', 'IRSL',
        'POSL' detected! NULL returned.
In addition: Warning message:
[analyse_SAR.CWOSL()] No signal or background integral applied,
        because they were set to NA!
```

The consistent labelling of output with function names will play an important role later on!

User experience in ‘Luminescence’ II

However, there were a number of frustrating failures:

```
plot_NRt(IRSAR.RF.Data)
```

```
Error in xy.coords(x, y) : 'x' and 'y' lengths differ
In addition: Warning message:
In nat[, 2]/reg[, 2] :
 longer object length is not a multiple of shorter object length
```

How to ensure that users are not exposed to error messages like this?

Enter fuzz testing

Fuzz testing is a software testing technique that

- generates *random*, *unexpected or malformed data* and feeds them into a program

Enter fuzz testing

Fuzz testing is a software testing technique that

- generates *random*, *unexpected or malformed data* and feeds them into a program
- with the aim to *uncover potential weaknesses*, *vulnerabilities and security flaws* in applications

REPLAY

Enter fuzz testing

Fuzz testing is a software testing technique that

- generates *random, unexpected or malformed data* and feeds them into a program
- with the aim to *uncover potential weaknesses, vulnerabilities and security flaws* in applications

Conventional setting for fuzz testing:

- systems where maintaining the integrity of the software and of the machine running it is of crucial importance (operating systems, hardware drivers, language compilers, browsers, etc)
- *statically-typed* programming languages or languages that provide *type annotations*

REPLAY

Enter fuzz testing

Fuzz testing is a software testing technique that

- generates *random, unexpected or malformed data* and feeds them into a program
- with the aim to *uncover potential weaknesses, vulnerabilities and security flaws* in applications

Conventional setting for fuzz testing:

- systems where maintaining the integrity of the software and of the machine running it is of crucial importance (operating systems, hardware drivers, language compilers, browsers, etc)
- *statically-typed* programming languages or languages that provide *type annotations*

It would seem that the R ecosystem doesn't require and cannot support fuzz testing!

Fuzz testing in R

In R, fuzz testing can be used to:

- identify functions that do not have sufficient argument validation
- identify sets of inputs that, while satisfying the implicit typing of a function signature, are problematic inside the function body.
  - presence of missing values, `NULL` entries, dimensionality- or sign-related errors.

REPLAY

Fuzz testing in R

In R, fuzz testing can be used to:

- identify functions that do not have sufficient argument validation
- identify sets of inputs that, while satisfying the implicit typing of a function signature, are problematic inside the function body.
  - presence of missing values, `NULL` entries, dimensionality- or sign-related errors.

Being a *dynamically-typed* language, in R there is no prior set of inputs that the fuzzer can be restricted to explore

- The task of rejecting invalid inputs falls on the package developer.

Existing packages

To date, there are only a handful of packages that provide an implementation of fuzz testing within the R ecosystem:

- **'fuzzr'** (on CRAN since 2016, last updated in 2018)
- **'RcppDeepState'** (not on CRAN) focused on fuzzing packages that use *C++* extensions generated via 'Rcpp'
- **'CBTF'** (on CRAN since 2025, actively maintained)

REPLAY

Caught by the Fuzz! ('CBTF')

First implementation (November 2024): 52 lines of actual code

Discovered over 280 failure cases across 59 issues in 'Luminescence'

Main driver:

```
fuzz(funs,              # names of functions to fuzz
     args,              # default argument list passed to all functions
     what,              # list of possible test inputs
     ignore_patterns) # whitelist patterns in error/warnings
```

https://mcol.github.io/caught-by-the-fuzz/

REPLAY

Functions to fuzz

While `fuzz()` can be applied to any function, it makes more sense to fuzz mainly the exported package functions (public interface)

Functions to fuzz

While `fuzz()` can be applied to any function, it makes more sense to fuzz mainly the exported package functions (public interface)

```
funs <- get_exported_functions("Luminescence")
```

- Uses `base::getNamespaceExports(package)` as workhorse
- Discards non-fuzzable exports (functions with no arguments)
- Applies optional ignore patterns (blacklisted functions, deprecated functions)

REPLAY

Inputs to test

Each possible input is specified as a list

```
what <- list(vector_with_NA = c(1:4, NA),
             empty_df = data.frame(),
             empty_RLum.Results = set_RLum("RLum.Results"))
```

Inputs to test

Each possible input is specified as a list

```
what <- list(vector_with_NA = c(1:4, NA),
             empty_df = data.frame(),
             empty_RLum.Results = set_RLum("RLum.Results"))
```

'CBTF' provides 85 default inputs across 14 different types (scalar, numeric, integer, logical, character, factor, data.frame, matrix, array, date, time, raw, NA, list):

```
test_inputs()  # "all"
test_inputs(c("scalar", "matrix"))
test_inputs(skip = c("raw", "time"))
```

REPLAY

Generation of fuzzed argument lists

Each element of the `args` list is modified in turn by each object in the input list (`what`):

```
res <- fuzz(funs = "rnorm",
            args = list(n = 11, mean = 22),
            what = list(NA, ""))
print(res, group = "function")
```

Generation of fuzzed argument lists

Each element of the `args` list is modified in turn by each object in the input list (`what`):

```
res <- fuzz(funs = "rnorm",
            args = list(n = 11, mean = 22),
            what = list(NA, ""))
print(res, group = "function")
```

```
-- Function `rnorm`:
 FAIL  invalid arguments              | n = NA, mean = 22
 FAIL  invalid arguments              | n = "", mean = 22
 WARN  NAs produced                   | n = 11, mean = NA
 FAIL  invalid arguments              | n = 11, mean = ""

 [ FAIL 3 | WARN 1 | SKIP 0 | OK 0 ]
```

Example run

```r
funs <- get_exported_functions("Luminescence")
res <- fuzz(funs, args = list(iris, numeric(), 100, NA_real_))
```

```
i Fuzzing 140 functions with 319 inputs (using 2 daemons)
i 44660 tests run  [19.8s]
x     CAUGHT BY THE FUZZ!

-- Test input [[165]]: iris, numeric(), 1.2, NA_real_
 analyse_SAR.TL  FAIL  missing value where TRUE/FALSE needed
       calc_IEU  FAIL  missing value where TRUE/FALSE needed
   calc_MaxDose  FAIL  [calc_MinDose()] 'sigmab' should be a single positive value # FP
convert_CW2pHMi  FAIL  result would be too long a vector
       get_RLum  FAIL  [<>()] 'class' should be of class 'character' or NULL       # FP

[ FAIL 1100 | WARN 15 | SKIP 5742 | OK 37803 ]
```

Whitelisting: removal of false positives

Default whitelist patterns:

- If the string “is missing, with no default” appears

Whitelisting: removal of false positives

Default whitelist patterns:

- If the string “is missing, with no default” appears
- If the name of the fuzzed function appears in the error or warning message
  - here we exploit the ‘Luminescence’ style of error reporting

Whitelisting: removal of false positives

Default whitelist patterns:

- If the string “is missing, with no default” appears
- If the name of the fuzzed function appears in the error or warning message
  - here we exploit the ‘Luminescence’ style of error reporting
- Custom whitelist regexp patterns, which can also be applied post-run

REPLAY

Whitelisting: removal of false positives

Default whitelist patterns:

- If the string “is missing, with no default” appears
- If the name of the fuzzed function appears in the error or warning message
  - here we exploit the ‘Luminescence’ style of error reporting
- Custom whitelist regexp patterns, which can also be applied post-run

```
res <- whitelist(res, "should be")
```

```
-- Test input [[165]]: iris, numeric(), 1.2, NA_real_
 analyse_SAR.TL  FAIL  missing value where TRUE/FALSE needed
       calc_IEU  FAIL  missing value where TRUE/FALSE needed
convert_CW2pHMi  FAIL  result would be too long a vector

[ FAIL 529 | WARN 15 | SKIP 5742 | OK 38374 ]
```

Performance considerations

Fuzzing a large number of functions with a large number of inputs can get very expensive

- 'CBTF' uses the 'mirai' package for asynchronous operations and parallelisation
  - Execution occurs on persistent background processes (daemons)
  - Queueing system to feed jobs to daemons
  - Support for timeouts (2 seconds by default) for functions that don't fail

REPLAY

Performance considerations

Fuzzing a large number of functions with a large number of inputs can get very expensive

- 'CBTF' uses the 'mirai' package for asynchronous operations and parallelisation
  - Execution occurs on persistent background processes (daemons)
  - Queueing system to feed jobs to daemons
  - Support for timeouts (2 seconds by default) for functions that don't fail

Fuzzing the first 6 arguments for all 'Luminescence' functions performs 93,686 tests in 18 seconds with 4 daemons.

Conclusions

Fuzzing can be an extremely powerful tool to discover issues with validation of inputs:

- Very useful for packages that aim for full coverage and thoroughly handled errors
- Quick and consistent feedback to user
- Simplification of function bodies
- The value of fuzzing is largely independent of the amount of test coverage in a package ('Luminescence' had already 99% coverage when 'CBTF' was introduced)

REPLAY