`signatr`: A Data-Driven Fuzzing Tool for R

Alexi Turcotte
turcotte.al@northeastern.edu
Northeastern University
Boston, MA, USA

Pierre Donat-Bouillud
donatpie@fit.cvut.cz
Czech Technical University Prague
Prague, Czech Republic

Filip Křikava
filip.krikava@fit.cvut.cz
Czech Technical University Prague
Prague, Czech Republic

Jan Vitek
j.vitek@northeastern.edu
Northeastern University
Boston, MA, USA

Abstract

The fast-and-loose, permissive semantics of dynamic programming languages limit the power of static analyses. For that reason, soundness is often traded for precision through dynamic program analysis. Dynamic analysis is only as good as the available runnable code, and relying solely on test suites is fraught as they do not cover the full gamut of possible behaviors. Fuzzing is an approach for automatically exercising code, and could be used to obtain more runnable code. However, the shape of user-defined data in dynamic languages is difficult to intuit, limiting a fuzzer's reach.

We propose a feedback-driven blackbox fuzzing approach which draws inputs from a database of values recorded from existing code. We implement this approach in a tool called `signatr` for the R language. We present the insights of its design and implementation, and assess `signatr`'s ability to uncover new behaviors by fuzzing 4,829 R functions from 100 R packages, revealing 1,195,184 new signatures.

CCS Concepts: • **Software and its engineering → Software notations and tools**.

Keywords: R, fuzzing, dynamic program analysis, dynamic programming languages

ACM Reference Format:
Alexi Turcotte, Pierre Donat-Bouillud, Filip Křikava, and Jan Vitek. 2022. `signatr`: A Data-Driven Fuzzing Tool for R. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, **??** pages. https://doi.org/10.1145/3567512.3567530

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SLE '22, December 06–07, 2022, Auckland, New Zealand
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9919-7/22/12.
https://doi.org/10.1145/3567512.3567530

1 Introduction

Dynamic analysis is often the only practical way to analyse code written in dynamic languages as the semantics of these languages severely limits static analyses. Dynamic analysis, however, requires both code to run and valid inputs for said code. To draw conclusions about a code base, one could run the existing, runnable code, *e.g.*, tests, but such code paints an incomplete picture as it is challenging to fully cover the range of behaviors allowed by dynamic languages.

To increase coverage, one could make use of a *fuzzer*, a tool that exercises code by generating random inputs. Fuzz testing has seen widespread adoption, primarily to find bugs, performance pathologies, and security vulnerabilities. However, dynamically typed languages such as Python, JavaScript, and R pose unique challenges that revolve around the idea that *dynamic code tends to hide runtime errors*. For instance, accessing a non-existent field of an object in JavaScript yields the value `undefined` instead of crashing, and basic functions in R will readily coerce values whose types do not match. A tool that tries to run code automatically will thus have very little to go on vis-à-vis the correctness of the code being generated as there is no clear observable witness of an error. On top of this, the lack of static types leaves fuzzers with very little information about what values are expected to begin with. Finally, it is difficult to generate complex values automatically in dynamically typed languages; in a statically typed language like Java, the shapes of user-defined objects can be inferred from a static class definition. In contrast, there is no such guide in dynamic languages, limiting a fuzzer's ability to generate realistic inputs.

To get around this, we propose an approach to fuzzing that relies on an extensive database of observed values. We develop a tracer that collects information about function calls and values created during code execution, and store this information in a database with an expressive query API. Then, we leverage this database to generate new function calls using the recorded values. This approach is implemented in a tool called `signatr` for the R programming language.

To validate our tool, we revisit an application of dynamic analysis, namely trace typing [**?**] where the goal is to guess the signatures of library functions by observing the values

that they accept and return. We perfom that experiment on R libraries and use the `contractr` type inference tool [**?**], wherein function types were inferred from recorded calls in R package test, example, and vignette code. Fuzzing 4,829 of those R functions with our tool generated 1.2M new unique signatures compared to the original study.

2 Background and Related Work

Related Work. One of the earliest fuzzers is Randoop, a feedback-driven random test generation tool for Java [**?**], where sequences of method calls are generated to test classes, and arguments are randomly generated for these calls. For primitive values, a random value is selected from a predefined, but user-extensible list; for reference types, a value is selected at random from those which have been seen, and if none are available then `null` is selected. While this technique is effective at generating tests involving non-trivial objects that are built up from a number of method calls, data science languages like R often require generating realistic data. QuickCheck is another interesting tool in this space [**?**]. While it can generate random inputs, the strategy for constructing inputs needs to be specified. The lack of type information in R limits the usefulness of such approaches. American Fuzzy Lop (AFL) is a state-of-the-art industrial fuzzer [**?**]. AFL takes a program and one example file as input, calls the program with the input, and then uses a variety of heuristics to transform the input and fuzz the program. AFL aims to find defects, while our approach aims to find novel inputs that successfully execute.

The R Language. We present a fuzzing tool called `signatr` for R, a language which sports an unusual mix of language features making it a challenging target for tooling [**?**]:

[–]

The lack of a static type system, so there is little to suggest what expected arguments or return values are.

Primitive values (booleans, integers, ...) have a separate "NA" value indicating that data is "not available".

Values are automatically and silently coerced across types. Each function may coerce parameters as it sees fit.

Most values are vectors and can be annotated by key-value pairs called *attributes* (*e.g.* the `dim` attribute turns a vector into a matrix). Attributes coupled with reflection are the building blocks for advanced features such as object-orientation. In R, there is no description of a shape of an object, it is simply a value with the `class` attribute.

Shared values are copied on write. A value is shared if it is accessible from multiple variables. Thus visible side effects are less frequent than in traditional imperative languages.

Function arguments are evaluated lazily resulting in unpredictable ordering of side-effects.

There has been work on a static type system for R [**?**]. A simple type annotation language has been evaluated on a corpus of 400 R packages. In Section **??**, we will revisit that work and demonstrate improvements by discovering new signatures for the previously studied functions.

3 Approach

There are two main phases to our approach: [(1)]
recording R code as it runs (Fig. **??**); this involves capturing function arguments and return values in our value database, and
fuzzing by drawing inputs to functions from the database (Fig. **??**).

For the recording phase, runnable code is extracted from examples and tests in packages. This runnable code consists of a set of scripts that can be run independently. Next, the scripts are executed in parallel, and all arguments are recorded into separate instances of our database, called the `sxpdb`. Finally, all the database instances are merged into one, in the process duplicate values are discarded. GNU parallel is used for orchestration [**?**]. Every R package hosted on R's main package repository (called CRAN) is required to have runnable example scripts to show how the package should be used, so there is a lot of code available.

As for the fuzzing phase, a list of functions to fuzz as well as a `sxpdb` are taken as input. The functions are fuzzed in parallel. The output of the fuzzer is a list of successful function calls, where a successful call is defined as one that generated no warnings, errors, and did not cause R to crash. The fuzzer relies on hooks before and after function execution: the hook before invocation allows the fuzzer to process inputs, and the hook after allows errors to be signaled and handled and the return value to be processed. Concretely, the fuzzer runs the functions using an extended R virtual machine that supports attaching callbacks to various runtime events [**?**]. Users can also plug a dynamic analysis directly into the pipeline.

Since R can crash (and does), to avoid losing results each instance of the fuzzer consists of two processes: a worker that fuzzes, and a supervisor that can spawn workers if needed. Furthermore, the fuzzer runs isolated in a container as calling arbitrary code could have dire consequences.

The tool itself consists of three standalone components: `argtracer`, the tracer responsible for running code and recording function invocation, `sxpdb`, the value database, and finally `generatr`, the fuzzer responsible for generating inputs. They are R packages written in a combination of R and C++.

3.1 Tracer

The tracer is built on top of the aforementioned extended virtual machine. The two runtime events we use are function exit, where all arguments are captured and stored, and a context jump, which is necessary to keep the call stack balanced as the interpreter uses long jumps for loop control flow, return statements, and error handling. (On a long jump, the exit hook is ignored, so we maintain our own version of the call stack to capture all function exits.) When the tracer

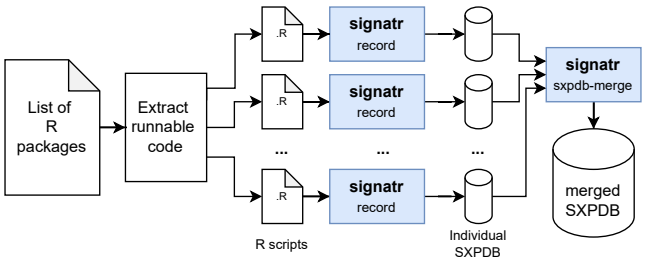

Figure 1. Recording pipeline.

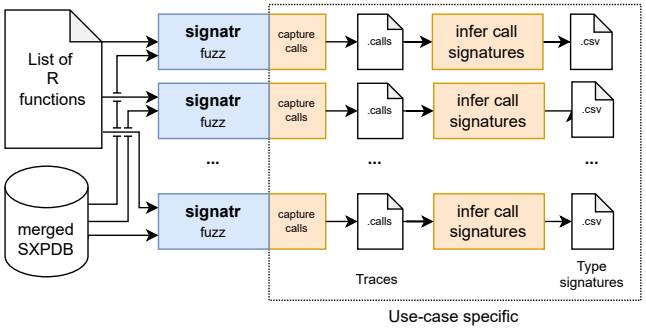

Figure 2. Fuzzing pipeline.

sees a call, it only sees a pointer to a closure, and the function's name and its package can only be found by searching through the loaded namespaces and the symbols they contain. Thus, the tracer eagerly builds an index of package and function names when namespace functions are loaded.

Regarding performance, there is 1.2— 7.8 (3.3 on average) slowdown when running with tracing enabled (based on the running / tracing 3,236 R scripts recording 8M unique values of 3GB size). This cost is mostly due to the value serialization. The tracer is written in 600 lines of C++ code.

3.2 Database of Values

The database is hand-written to leverage domain-knowledge of R values and optimize it for the queries supported by our API. It is implemented in 5K lines of C++ and 1.5K lines of R.

Storage. The database stores unique values. We use XXH-128 hashes for uniqueness in combination with a hash table based on RobinHood hashing[1]. While the hashing is fast, we need to lower R values into a binary format. R provides a binary serialization XDR,[2] but it is costly. We also strip the serialization of sources of non-determinism related to character encodings. Since many values are pushed to the database in an average recording session, we try to avoid serialization as much as possible. For this, we use the `trace` bit that is part of each value.[3] If not set, the value is fresh and we serialize it, compute its hash, store it. The trace bit is then set to avoid repeated serialization. In spite of R's copy-on-write semantics, values can be modified in place before being shared, and these values may be serialized repeatedly to capture the updates performed by the program.

The database also stores metadata about values and their origin. We also keep a unique id for each sequence of arguments coming from the same call site. This allows us to replay the calls as they were observed.

The database maintains tables for the hashes, runtime metadata (e.g., how often a value was seen), static metadata, origins, call ids, and class names. Variable length data are stored in a combination of 2 tables, one giving an offset into the other table which holds the size of the value and the value itself. Origin strings and class names are interned, i.e., each unique string is stored separately, and referred to by pointer. Search indices are built using fast compressed bitsets [**?**]. The database supports all values except external pointers (e.g. pointer to C allocated data). Environments and closures were not stored in the database during our experiments since they dramatically increased memory pressure.

Finally, opening the database in read mode only loads metadata. Retrieving values from disk is done on demand, making it possible to query larger-than-memory databases.

Queries. Values can be queried based on their `typeof`-type or class, on the presence of NAs, number of attributes, and dimensions. The database can be queried for a random value with the desired metadata, or can be queried by providing an existing value along with a list of search parameters to be relaxed.

3.3 Fuzzing

The value database is at the core of our fuzzing approach, which is similar in spirit to mutation-based fuzzing, where valid inputs are taken and mutated to try to exercise new functionality (rather than have inputs be randomly generated). Instead of mutating arguments to previous calls, new argument values are selected based on previous ones.

The fuzzer generates calls to a function and chooses arguments to these calls as depicted in Algorithm **??**. In addition to the target function and database, the algorithm considers how many query parameters to relax on (*numRelax*), as well as all of the previously seen successful calls to the function (*succs*). For each parameter, the algorithm determines how to relax (this may change from one iteration to the next), finds all values that inhabited that parameter in successful calls, chooses one value, and queries the database for a value similar to it save for the relaxation. If no successful calls to the function have been observed, random values can be chosen.

[1] cyan4973.github.io/xxHash, github.com/martinus/robin-hood-hashing
[2] cran.r-project.org/doc/manuals/r-release/R-ints.html#Serialization-Formats
[3] A bit in the C struct that represents a value, cran.r-project.org/doc/manuals/r-release/R-ints.html#Rest-of-header.

The fuzzing approach itself is depicted in Algorithm **??**. First, the collection of already known calls to the function is obtained from the database. The main idea of the approach is to start by selecting new arguments essentially at random by querying the database and relaxing on many parameters, and gradually reduce the number of parameters being relaxed as the fuzzer progresses. Concretely, the number of parameters being relaxed is reduced every *tick*, which is determined by dividing the total fuzzing budget by the number of parameters that can be relaxed (*numRelaxParams*). The function will be fuzzed for as long as the budget allows, and initially all database parameters will be relaxed. Arguments for a new call are generated through the approach depicted in Algorithm **??** (*getArgs*), the call is performed, and the results are saved in *res*. If there were no errors, warnings, or crashes, then the successful call is added to the list of successful calls *succs* and iteration continues until the budget is exhausted.

Algorithm 1 Selecting Arguments for A Call

```
1:  procedure GETARGS(f, numRelax, db, succs)
2:      params ← getParams(f)
3:      for p in params do
4:          ▷ relax on numRelax params
5:          relax ← pickSome(relaxParams, numRelax)
6:          ▷ get all values that p had in successful calls to f
7:          seed ← getArgsFor(p, succs)
8:          ▷ choose one at random
9:          v ← pickOne(seed)
10:         ▷ sample a similar value from the database
11:         args[p] ← sampleSimilar(v, db, relax)
12:     end for
13:     return args                          ▷ the args for the new call
14: end procedure
```

3.4 Intended Use

The tracing discussed in Section **??** is costly in that running large swaths of R code takes time. Thankfully, this only needs to be done *once* to construct a database, which can then be reused by the fuzzer. For simplicity, our artifact[4] contains instructions to download a 10GB sample database.

4 Assessment

We used `signatr` to stress-test one of the proposed type system for R [**?**]. The type system was designed empirically, with the help of a dynamic analysis that inferred function signatures from the types of values observed while running extracted code from packages. In a nutshell, a type was inferred from each call to a function, and these types were unified into a single function type, and the more unique successful calls there are, the more precise the inferred signature—the types inferred from successful calls are referred to as *call signatures*. It would be therefore interesting to see *how many additional successful calls can* `signatr` *generate?*

[4] https://github.com/PRL-PRG/sle22-signatr-artifact

Algorithm 2 Fuzzing

```
1:  procedure FUZZWITHDB(f, db, budget)
2:      succs ← getSuccessfulCalls(f, db)
3:      tick ← budget/numRelaxParams
4:      relaxThisTime ← numRelaxParams
5:      i ← 1
6:      while i ≠ budget do
7:          ▷ gradually relax on fewer params
8:          if i mod tick = 0 then
9:              relaxThisTime ← relaxThisTime − 1
10:         end if
11:         args ← getArgs(f, relaxThisTime, db, succs)
12:         res ← call(f, args)
13:         ▷ add successful call to succs
14:         if no warnings, errors, crashes in res then
15:             succs ← succs + res
16:         end if
17:         i ← i + 1
18:     end while
19:     return succs                         ▷ the successful calls to f
20: end procedure
```

We ran all of our experiments on two Ubuntu 18.04 servers, each with a 72 core Intel Xeon 6140 2.30GHz processor and 256GB of RAM.

Recording. First, we created a `sxpdb` for the fuzzer. For this we used the extracted runnable code from the same corpus as the original study which consists of 17,463 R scripts containing 389.7K lines of code (excluding comments and new lines). The database was generated in 16 hours and occupies 287.17 GB of disk space. It contains 39.4M unique values recorded from 20.5M calls to 38.1K functions in 652 packages. Figure **??** shows the distribution of main value types. The vast majority of values are vectors and matrices of real numbers which is unsurprising as R is mostly used for numerical computing. That said, many of them (47.2%) contain attributes which is what makes them interesting, as attributes add semantic meaning. The next big group are lists, which can be divided into two groups, data frames (two dimensional, column-major structure representing observations) and records. Note that 0.5% are logical *vectors* of varying length, not simply `true` or `false`.

Fuzzing. Armed with this database, `signatr` fuzzed 4,829 functions from 100 packages, a subset of the original corpus, in 19 hours. The fuzzing budget was set to 5,000, with 64 functions being fuzzed in parallel. In total, 24.1M calls were made, and out of that, 13.9% were successful, resulting in 3,351,753 traces of 2,315 functions coming from 98 packages. The vast majority of errors were exceptions, but in 211 cases, the R process crashed. While the aim of this work is not bug

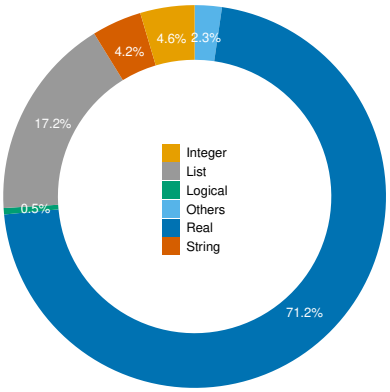

Figure 3. The sxpdb value type distribution

finding, we have investigated one such crash in `stringi`[5], and found that it was caused by memory corruption by large input. The issue was reported, acknowledged and fixed.

Results. The results of fuzzing is shown in Figure **??**. From the 4,829 functions in our 100 packages corpus, the fuzzer managed to generate call signatures for 2,189 functions (45.3%). In comparison, tracing, *i.e.*, recording calls by running the extracted code covers 3,135 functions (64.9%). While the fuzzer covered fewer functions, it generated over 1.2M new unique call signatures (on average 562.7 per function), *i.e.*, 105.1 times more call signatures in comparison to tracing. Out of the 2,189 functions, 607 functions were covered only by fuzzing. Fuzzed signatures overlapped with traced ones in only 1,082 cases in 597 functions.

The reason that `signatr` failed to generate a single successful call for 2,640 functions is because they require a very specific shape of one or more of its arguments, or the arguments depended on one-another. As an example of the latter case, certain functions from the `dplyr` data manipulation package required a data frame alongside an unevaluated expression made up of column names from the data frame. Besides that, problematic functions include those that use non-standard scoping, manipulating arguments as unevaluated expressions and evaluating them in custom environments. In some cases the error message could have been used as a feedback to the fuzzer (and we plan to address it in the next iteration), but this is not easy to generalize.

Next, we looked at code coverage to see if the new call signatures translated to more code being exercised. Using the `covr` package[6], we computed line coverage of R source code for 1,342 functions using the fuzzed calls, and separately using the traced calls. This is not all of the functions that `signatr` managed to fuzz, as running `covr` on some functions caused runtime errors, and also repeating certain calls failed (both fuzzed and traced). For 294 functions, `signatr` improved code coverage on average by 20.4% (as compared

[5]A sting processing library, one of the most downloaded package in R.
[6]The only tool for R code coverage, *cf.* https://covr.r-lib.org

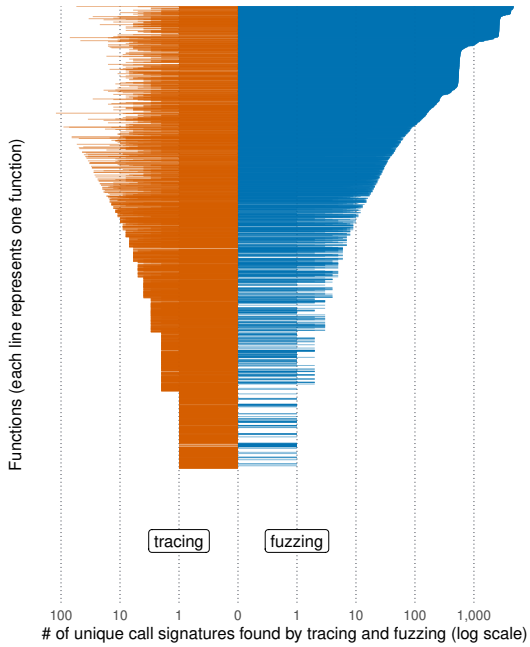

Figure 4. Number of unique call signatures.

with coverage obtained by simply running package tests, examples, and vignettes). 1.2M new calls yielded only 20.4% more coverage—while new code paths were explored, the extensive polymorphism and use of coercion in R mean that most of these new calls did not exercise new R code.

Summary. `signatr` uncovers many calls with new type signatures, significantly expanding on what tracing alone can discover, particularly for polymorphic functions. Together, existing and generated calls yields a wealth of interesting runnable code that will be essential for the further design of a possible type system for the R programming language.

5 Conclusions and Future Work

Fuzzing is a useful technique not only for finding bugs and security vulnerabilities, but also for getting insights about code. However, it is hindered by the permissive semantics of dynamic languages as well as the dynamic nature of how complex data is defined. In this work, we proposed a fuzzing approach that relies on a database of observed values to provide complex and realistic inputs for functions. We implement this approach in a tool called `signatr` for the R programming language. We show that `signatr` uncovers many new call signatures for R functions and it can be a useful tool in a data-driven language evolution toolbox.

There are various avenues to improve the tool itself. For instance, the fuzzing component of our approach could be enhanced to consider the types of default function parameters. Further, we could expand the database with knowledge of how values co-occur, which would add more dimensions to

how values can be selected. While this approach was implemented in a tool for R, it is broadly applicable in all languages, and the only real language-specific aspect is the set of parameters in the database. For instance, one could implement a similar tool for object-oriented languages, where database metadata could include object field names (*e.g.*, JavaScript).

A `signatr` demonstration

The following is a short demonstration of the basic `signatr` functionality, *i.e.*, how to create the value database by running R code and then how to use it for fuzzing. The tool is packaged as an R library. It can be used both in a script or interactively from an R REPL.

We begin by starting R (concretely, *R-dyntrace* version 4.0.2 which has the hooks used by the *argtracer*) and loading `signatr`. In the following listings, the `$` indicates shell prompt and `>` denotes the R REPL.

```
$ R
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
...
> library(signatr)
```

To generate a database of values, we need some code to run. One way to get it is to extract it from an existing R package, for example `stringr`:

```
> extract_package_code("stringr", output_dir = "demo")
...
7 examples/str_detect.Rd.R examples
...
```

This will extract all the runnable snippets from the package documentation and tests into the given directory. For example:

```
$ cat demo/examples/str_detect.Rd.R
...
fruit <- c("apple", "banana", "pear", "pinapple")
str_detect(fruit, "a")
str_detect(fruit, "^a")
...
```

Next, we trace the file by executing it and recording all the calls using the `trace_file` function:

```
> trace_file("demo/examples/str_detect.Rd.R", db_path = "demo.sxpdb")
  status  time  db_size  error
  0  0.024      20     NA
```

The resulting database is stored as `demo.sxpdb`. In this example, after running the `str_detect.Rd.R` file, the database contains 20 unique values. This can be repeated for all the other files. To trace multiple files in parallel, one database is created per file, which are merged using the `merge_dbs` function once they are all available. This also allows us to run larger experiments on multiple machines.

Once the database is ready, we can start fuzzing. The fuzzer has a number of configuration points, but the easiest starting point is the `quick_fuzz` helper function:

```
> R <- quick_fuzz("stringr", "str_detect", "demo.sxpdb",
budget = 100, action = "infer")

started a new runner:PROCESS 'R', running, pid 4157
fuzzing stringr:::str_detect [======] 100/100 (100%) 39s
stopped runner:PROCESS 'R', running, pid 4157
```

The `infer` action will use the `infer_call_signature` function that infers types for each call argument and return value using the type annotation language of Turcotte et al. [**?**] (with the *contractr* type inference tool). `infer_call_signature` returns a data frame with details for each call. Here, the data frame includes the inferred signature in the `result` column.

```
> print(R)
# A tibble: 100 x 6
args_idx error              status result            time
<list>   <chr>              <int> <chr>             <drtn>
1 <int [3]> "Error in UseMeth...   1  NA              0.0363
2 <int [3]> NA                     0  (character[],... 0.0351
...
```

The above listing shows two calls: a failed one (non-zero status) with an error message, and a successful one with an inferred signature. The `args_idx` column contains the indices in the `sxpdb` of the values; thus, the actual argument values can be obtained by looking up the `args_idx` in the `sxpdb`.

One advantage of using R is that we can use R's many data analysis functions. For example, we can look at the resulting signatures:

```
> count(R, result)
# A tibble: 4 x 2
result                                                    n
<chr>                                                 <int>
1 (character[], ^character[], double) => ^logical[]       1
2 (character[], character, integer) => logical[]          1
3 (list<integer>, character[], list<integer>) => logical[]   1
4 NA                                                     97
```

This shows that in three cases, the fuzzer managed to generate a call that was successful, and the signatures of those calls. If you are repeating these steps, it is possible that your results will be different since fuzzing is non-deterministic.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.