Signatr Artifact

We also provide pdf and html versions of this README. If reading locally and not on github, we advise to use the html version.

The artifact contains the `signatr` tool, and the pipelines to create an R value database and to fuzz R functions with the database to find type signatures. The pipeline to create a valeu database is in `pipeline-dbgen`. The fuzzing pipeline will generate the inputs for the `sle.Rmd` R markdown notebook. That notebook can then be rendered to get all the results (tables, figures) we use in the paper.

To use the artifact:

1. Install the docker image (see Install the docker image). Installing locally is possible but involved. Following the steps described in the `docker-image/Dockerfile` should help if this is the hard path you are choosing!
2. Experiment with the tool on a small example: see Experimenting the tool
3. Reproduce the analysis pipeline: see The analysis pipeline)

Tool

The tool is packaged as an R library. It is hosted at https://github.com/PRL-PRG/signatr and uses the following building blocks:

- sxpdb: R value database
- generatr: fuzzing utilities
- contractr:type signature parsing and checking for R
- argtracer: trace R values using a patched R interpreter and store them in the R value database.

The tool and its dependencies are pre-installed in a convenient Docker image.

Install the docker image

You can:

- pull the docker image with `docker pull prlprg/sle22-signatr`, or
- build the docker image (it takes time!):

```
cd docker-image
make
```

After installing the docker image, **make sure** to run all the following commands in a shell inside the docker image (for Linux, macOS) from the artifact directory. To start the docker image:

```
./enter
```

Experimenting with the tool

Run the R interpreter *inside the docker image*. It will start the patched R interpreter. The tool *does not run* in the standard R interpreter.

In the following listings, $ indicates the shell and > denotes the R REPL.

```
R version 4.0.2 (2020-06-22) -- "Taking Off again"
...

> library(signatr)
```

Database

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

```
> 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 running it (in the patched R interpreter) and recording all the calls, using the trace_filefunction:

```
trace_file("demo/examples/str_detect.Rd.R", db_path = "demo.sxpdb")

        status time                          file    db_path db_size error
elapsed      0 0.04 demo/examples/str_detect.Rd.R demo.sxpdb      20    NA
```

The database generation is also automated in the pipeline-dbgen directory in the artifact, and handles there tracing on multiple files and merging the results. See Generate the database for more details.

Fuzzing

Once the database is ready, we can start fuzzing the str_detect function of the stringr package:

```
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 infer types for each call argument and return value using the type annotation language supported by `contractr`. It returns an R data frame with the inferred call 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
```

If you are repeating these steps, it is possible that your results will be different since fuzzing is non-deterministic.

The 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 of the values of the arguments in the database: the actual argument values can be obtained by looking up the `args_idx` in the database:

```
> library(sxpdb)
> db <- open_db("demo.sxpdb")
> get_val_idx(db, 0) # value at index 0
[1] "a"
> close(db)
```

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 3 cases, the fuzzer managed to generate a call that was successful, and so the signatures of those calls.

The analysis pipeline

The following tutorial demonstrates how to run the analysis pipeline to reproduce the results of the paper. It consists of a series of steps that at the end generates the input for the analysis.

In this write up, we will run it on a small subset of the original packages (cf. `data/packages.txt`). The reason is that the size of the data require is fairly large. For example, just the value database is over 287GB and its generation take over half a day (on a 72 core Intel Xeon 6140 2.30GHz server). Also one would have to download and install all the packages and their dependencies which again takes space and time. If you are however interested and have the computational resource, we will be happy to share the data, please contact the AEC chair.

Note: You will be running code downloaded from a public repository. CRAN is a curated repository, yet it should be done with caution. Run it inside the container.

Steps

The following is essentially what is in the Figure 1 and Figure 2 in the paper, packaged in scripts for simpler use using GNU parallels for parallel execution. All steps shouls be run inside a docker container. To enter the container, run:

```
./enter.sh
```

which should give you a bash shell prompt, like (modulo the hostname):

```
r@eaf63037fd02:/work$
```

It automatically mounts the content of the folder from which you run the command into the `/work` directory in the container.

0. get the sample sxpdb database

For the experiment we need a value database (sxpdb database) that will be used for the fuzzing. You can either build one yourself, or download one we have prepared using the same steps.

To get the prebuilt one do the following:

```
cd data
wget -O cran_db.tar.xz https://owncloud.cesnet.cz/index.php/s/aHprMbas4haELVf/download
tar xvJf cran_db.tar.xz
```

The extracted database has about 10GB.

Building it yourself The database generation uses targets to orchestrate the pipeline.

The database for the SLE paper is obtained by tracing 400 packages from `data/packages-typer-400.txt`.

To start tracing, after opening an R session and specifying an adequate number of parallel workers:

```
cd pipeline-dbgen
cp packages-typer-400.txt packages.txt
targets::tar_make_future(workers = 64)
```

The extracted code of the packages will be in `data/extracted-code`. The resulting database will be generated as `data/sxpdb/cran_db`. It will also output a call id companion file in `data/callids.csv`. Depending on your machine, the generation of the database for the 400 packages can take from a few hours to a few days.

We provide other variants of `packages.txt`. For instance, `packages-4.txt` includes 2 huge and common R packages, `dplyr` and `ggplot2`. We provide pre-extracted code for a few packages already, including `stringr`, `dplyr`, and `ggplot2`.

1. create a corpus

The corpus consists of the following:

- R package sources in `data/sources`
- installed R packages `data/library`
- extracted code from R packages `data/extracted-code`
- corpus metadata file `data/corpus.csv`

This is bootstrapped using the `data/packages.txt` file which contains a new-line separated list of packages to include in the corpus.

To create a corpus, run the following:

```
./create-corpus.R
```

Depending on the number of packages (and their transitive dependencies), it might take a while. For the sample of 5 packages (small corpus, though of the very popular packages), it might be ~20 minutes.

It could happen, that some dependencies won't install.

The result should be something like:

```
data/extracted-code  <--- extracted code from R packages
data/library         <--- installed R packages
data/sources         <--- R package sources
data/corpus.csv      <--- corpus metadata
```

2. fuzz the installed functions

Next, we will run the fuzzer using the values from the sample database:

```
./run-fuzz.sh
```

By default this will sample 100 functions from the `corpus.csv` and fuzz each 100 times. Both can adjusted by setting the `FUNS` and `BUDGET` environment variables. Using all the functions (e.g. `FUNS=$(wc -l data/corpus.csv)` and 5000 runs (e.g. `BUDGET=5000`), the experiment might take about a day. That is why we recommend to scale it down. By default, it will run 16 jobs in parallel. The can be changed using the `JOBS` environment variable.

The result will be:

```
data/fuzz             <--- directory with the fuzzer output
data/run-fuzz.csv     <--- metadata about the run, duration, exitcodes, ...
```

You could view the intermediate results using the `qcat.sh` utility. For example:

```
./qcat.R 'data/fuzz/dplyr::arg_name'
```

shall show results for a function `arg_name` from `dplyr` package:

```
# A tibble: 100 × 9
    args_idx  error        exit status dispatch     result ts    fun_n...¹ rdb_p...²
    <list>    <chr>       <int>  <int> <list>        <int> <drt> <chr>    <chr>
  1 <int [2]> "Error in ...    NA      1 <named list>     NA 0.08... dplyr:... ../rdb...
  2 <int [2]> "Error in ...    NA      1 <named list>     NA 0.11... dplyr:... ../rdb...
  3 <int [2]> "Error in ...    NA      1 <named list>     NA 0.14... dplyr:... ../rdb...
  4 <int [2]> "Error in ...    NA      1 <named list>     NA 0.15... dplyr:... ../rdb...
  5 <int [2]> "Error in ...    NA      1 <named list>     NA 0.09... dplyr:... ../rdb...
  6 <int [2]> "Error in ...    NA      1 <named list>     NA 0.53... dplyr:... ../rdb...
  7 <int [2]> "Error in ...    NA      1 <named list>     NA 0.11... dplyr:... ../rdb...
  8 <int [2]>  NA            NA      0 <named list>     30 0.09... dplyr:... ../rdb...
  9 <int [2]>  NA            NA      0 <named list>     31 0.09... dplyr:... ../rdb...
 10 <int [2]>  NA            NA      0 <named list>     32 0.09... dplyr:... ../rdb...
...
```

It indicates 7 failed calls and 3 good ones. Please note that due to random sampling your results will likely be different.

3. type the results

To type the traces, run the following:

```
./run-type.sh
```

By default, it will run 16 jobs in parallel. The can be changed using the `JOBS` environment variable.

The result will be:

```
data/types             <--- directory with the type output
data/run-type.csv      <--- metadata about the run, duration, exitcodes, ...
```

We can again peek the results:

```
./qcat.R 'data/types/dplyr::arg_name'
```

which should show types inferred from the fuzzed calls:

```
# A tibble: 40 × 3
   fun_name           id signature
   <chr>           <int> <chr>
 1 dplyr::arg_name     8 (list<list<class<unit, unit_v2> | double | integer> | ...
 2 dplyr::arg_name     9 (class<gList>, list<class<factor> | double | integer>)...
 3 dplyr::arg_name    10 (pairlist, list<character | double[]>) => class<glue, ...
 4 dplyr::arg_name    13 (list<list<class<matrix> | double[] | integer | intege...
 5 dplyr::arg_name    14 (character[], list<character | logical>) => class<glue...
 6 dplyr::arg_name    15 (list<class<unit, unit_v2>>, list<list<class<expectati...
 7 dplyr::arg_name    17 (list<class<call>>, double[]) => class<glue, character>
 8 dplyr::arg_name    24 (list<class<margin, simpleUnit, unit, unit_v2> | class...
 9 dplyr::arg_name    28 (class<matrix>, list<class<expectation_success, expect...
10 dplyr::arg_name    30 (double, class<titleGrob, gTree, grob, gDesc>) => clas...
```

4. fuzz coverage

Computing the function source code coverage from the fuzzed calls is done by running the following:

```
./run-coverage.sh
```

This will use the traced data to recreate the calls while using the covr tool to record code coverage. By default, it will run 16 jobs in parallel. The can be changed using the `JOBS` environment variable.

The result will be:

```
data/coverage          <--- directory with the coverage output
data/run-coverage.csv  <--- metadata about the run, duration, exitcodes, ...
```

5. baseline tracing

In this steps we will run all the extracted code to create the baseline for the comparison.

```
./run-baseline.sh
```

This will use the GNU parallel to trace all the runnable code extracted from the installed packages. By default, it will run 16 jobs in parallel. The can be changed using the `JOBS` environment variable.

6. type the baseline traces

7. compute baseline coverage

8. create a report

We just have to render the RMkardown file `sle.Rmd`. It will output an `experiment-uf.tex` file with macros for all the experimental values in the paper, and a pdf file (`uf-call-signatures.pdf`) for Figure 4 in the paper.

```
R -e 'rmarkdown::render("sle.Rmd")'
```