Why do we Eval?

A large-scale study of Eval usage in R

Anon
Under review

Abstract

Most dynamic languages allow users to turn text into code using various functions, often named `eval`, with language-dependent semantics. The widespread use of these reflective functions hinders static analysis and prevents compilers from performing optimizations. This paper aims to provide a better sense of why programmers use `eval`. Understanding why `eval` is used in practice is key to finding ways to mitigate its negative impact. We have reasons to believe that reflective feature usage is language and application domain specific; we focus on data science code written in R, and compare our results to previous work that analyzed web programming in JavaScript. Dynamic analysis of a corpus of 4.5M lines of libraries and end-user code confirms that `eval` is indeed in widespread use; R's `eval` is more dangerous in some ways, and safer in others, than what was previously reported for JavaScript.

2012 ACM Subject Classification Software and its engineering → Scripting languages; General and reference → Empirical studies; Software and its engineering → General programming languages

Keywords and phrases R, reflection, eval

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.23

1 Introduction

Most dynamic languages provide their users with a facility to transform unstructured text into executable code and evaluate that code. We refer to this reflective facility as `eval` bowing to its origins in LISP, all the way back in 1956 [?]. `Eval` has been much maligned over the years. In computing lore, it is as close to a boogeyman as it gets. Yet, for McCarthy, `eval` was simply the way to write down the definition of LISP, he was surprised that someone coded it up and offered it to end users. Since then, reflective facilities have been used to parameterize programs over code patterns that can be provided after the program is written. The presence of such a feature in a language is a hallmark of dynamism; it is a form of delayed binding as the behavior of any particular call to `eval` will only be known when the program is run and that particular call site is evaluated.

Trouble in Paradise. Reflective facilities hinder most attempts to reason about, or apply meaning-preserving transformation to, the code using them. In practice, `eval` causes static analysis techniques to loose so much precision as to become pointless. For compilers, anything but the most trivial, local, optimizations are unsound after a use of `eval`. Furthermore, the addition of arbitrary code — code that could have been obtained from a network connection — as a program is running is a security vulnerability waiting to happen. To illustrate these challenges, consider the interaction of a static analysis tool with a dynamic language. An abstract interpretation-based program analyzer computes an over-approximation of the set of possible behaviors exhibited by the program under study [?]. A reflective may have *any* behavior that can be expressed in the target language; i.e. `eval` can be replaced by any legal sequence of instructions. As dynamic languages tend to be permissive, the analysis has to, for example, assume that many (or all) functions in scope may have been redefined, e.g. that `‘+‘` now opens a network connection or something equally surprising. A single occurrence of `eval` causes the static analyzer to loose all information about program state and meaning of identifiers. This loss of precision can sometimes be mitigated by analyzing the string argument [?] to bound its possible behavior but when the string comes from outside the program not much can be done. A frustrated group researchers argued giving up on soundness and, instead, under-approximating dynamic features (soundiness) [?]. In their words "a practical analysis, therefore, may pretend that `eval` does nothing, unless it can precisely resolve its string argument at compile time." Alas, assuming that `eval` does not have side-effects, or that side-effects will not affect the results of the analysis, may be unduly optimistic.

Is Past Prologue? Previous work investigated how `eval` is used in web programming, specifically in websites that use JavaScript [?]. In 2010, 17 of the largest website used the feature. In 2011, 82% of the 10,000 most accessed sites used `eval` [?]. Yet, the strings passed to `eval`, and their behaviors when executed, are far from random; it was shown that when one can observe several calls to `eval`, the "shape" of future calls can be predicted with 97% accuracy [?]. Overall, practical usage suggested that most reflective calls were relatively harmless. While this backs up the soundiness squad's approach, does it generalize to other application domains than web programming and to other languages?

The Here and Now. In this study, we investigate the usage of `eval` in programs written in the R programming language. R is language designed by statisticians for applications in data science [?, ?]. What makes looking at R after JavaScript interesting is that, while both languages are dynamic, they are quite different. While one can program in an object-oriented style in R, like in JavaScript, R is primarily a lazy, untyped, functional language. JavaScript was designed to run untrusted code in browser, while R is used for statistical computing on desktops. JavaScript is a general purpose language used by a wide community of programmers; while R is used for scientific computing by data scientists and domain experts with, often, limited programming experience. One can distinguish between library implementers, developers with some programming experience and a working knowledge of R, and end-users, who are typically not expert programmers and often have only a cursory knowledge of the language.[1] Our goal is thus to highlight the differences in usage between JavaScript and R, and try to explain those differences in terms of language features, application domain and programmer experience. Hopefully some of our observations will generalize to other languages.

The What and How. One significant benefit of choosing R is that every package in the CRAN repository is curated and comes with examples of typical usage. This gives us a large code base that we can analyze dynamically. To observe `eval` we built a two-level monitoring infrastructure:[2] we can monitor R programs by instrumentation — this gives us access to many user-visible properties of R programs — but we can also monitor the inner-workings of the R interpreter — this allows us to capture details not exposed at the source level. Dynamic analysis is limited, it can only observe behaviors triggered by the particular inputs passed to a program. Luckily, CRAN libraries come with many tests and use-cases. The choice of corpus is crucial. Our corpus has been constructed to reflect the levels of sophistication of the R community. We distinguish between *CRAN packages* (500

[1] Consider that R evaluates function argument lazily, just like Haskell. We informally surveyed end-users, including computer scientists, and did not find a single user aware of this fact. Library developers, on the other hand, know about laziness and program defensively around it.

[2] Our infrastructure is open source and publicly available, our anonymized code is at `http://url.com`, a more complete artifact will be submitted to the artifact evaluation committee.

curated packages that pass stringent quality checks and are equipped with tests and sample data) and *Kaggle scripts* (1,619 end-user written programs that performs a particular data analysis task). It is reasonable to expect that `eval` usage differ between these datasets: the libraries represent a lively ecosystem with new libraries added each day, while end user code is often thrown together, run once, and never revisited.

Why do we Eval? **A short summary of the results should go here.**

2 Background and Previous work

This section provides a short introduction to R as it has a few surprising features that impact the use of reflective features of the language. We then look at the semantics of `eval` in R and discuss some design choices. Lastly, we put this paper in context of previous work.

2.1 R in a nutshell

R is a lazy functional programming language [?] with dynamic features that allow to write object-oriented code. Most data types are vectorized. Values are constructed by the `c(...)` function, e.g. `c("a","bc")` creates a vector of two strings. To enable equational reasoning, R copies values accessible through multiple variables when they are written to. Values can be tagged by user-defined attributes. For instance, one can attach the attribute `dim` to the value `x<-c(1,2,3,4)` with `attr(x,"dim")<-c(2,2)`. This causes arithmetic functions to treat `x` as a 2×2 matrix. Another attribute is `class` which can be bound to a list of names, e.g., `class(x)<-"human"`. This sets the class of `x` to `human`; classes are used for object-oriented dispatch. Every R linguistic construct is desugared to a function call, even control flow statements, assignments, and bracketing. Furthermore, all functions can be redefined in user code. This makes R both flexible and challenging to compile [?]. Arguments to user-defined functions are bundled into thunks called *promises*. Logically, a promise combines an expression's code, its environment, and its value. To access the value of a promise, one must force it. Forcing a promise triggers evaluation and the computed value is captured for future reference.

2.2 Eval Semantics

The expressive power of `eval` depends on design decisions along two axes:

- **Scoping:** What environment does `eval` executes its argument in? JavaScript and R evaluate it in the current environment, thus exposing local variables and parameters and breaking the caller's abstraction boundary. Julia is more restrictive, `eval` runs at the "top level", in the global environment. JavaScript has a global `eval` that behaves like Julia, and a "strict mode" in which `eval` may access local variables, but fobidden from injecting new ones in the local environment. In Java, one can implement `eval` with Julia-semantics.[3]
- **Reflective API:** What other reflective operations are exposed to user code? JavaScript provides few reflective functions (other than enumeration of an object's properties and string-based indexing), the impact of `eval` is thus limited. R, on the other hand, has a

[3] While `eval` used to be the purview of interpreted languages, just-in-time compilation lifted this restriction. `Eval` takes its input, wraps it in a static method of a new anonymous class, generates bytecode for that class, invokes the class loader to install that code, and finally reflectively calls the method.

rich reflective API. For example, R allows user code to walk the call stack and arbitrarily add and delete local variables. This means that there are few limits to the side-effects of an `eval`.

These design choices impact our ability to reason about code when reflection is used. A more expressive `eval` translates directly into additional restrictions for program analysis and transformation tools. The reason Julia restricts `eval` to the top level is to preserve the compiler's ability to optimize code [**?**]. In particular, `eval` had to be prevented from changing the type of local variables as this would require recompilation of the method. Julia even adds a versoning mechanism, called world age, to ensure that code added during an `eval` does not invalidate inlining optimizations [**?**]. More permissive languages such as R are much harder to deal with from a compiler's perspective.

Eval in R. The reflective interface exposed by R is somewhat complex. The core library exports four functions with slightly different semantics, `eval`, `evalq`, `eval.parent`, and `local`. `Eval` is the more general function, and, unlike in JavaScript, it takes three arguments, an expression, its environment and the enclosing environment. The following discussion simplify some details which are not relevant to this paper. For the interested reader we recommend the excellent book by Hadley Wickham [**?**]. The definition of `eval` starts as follows:

```
eval
  <- function(expr,
              envir = parent.frame(),
              enclos = if(is.list(envir)) parent.frame() else baseenv())
        { ...body... }
```

The parameters are `expr`, the value to be evaluated, `envir`, the environment in which evaluation happens, and `enclos`, the environment to look up objects not found in `envir` (by default either the top level, `baseenv` or the environment of the caller). The `expr` parameter can take values of different type, for our purposes we focus on the most common, `expressions`. It is easiest to think of an `expression` as an abstract syntax tree as returned by R's parser. The simplest way to create an `expression` is to call `quote` passing some R expression:

```
> quote(a + b)
# a + b
```

Here the return value is an abstract syntax tree. From a string, use the `parse` function:

```
> parse(text="a+b")
# expression(a+b)
```

The most common way to create an expression is to extract it from an argument. Each argument comes packaged inside a promise which retains the source code of the argument. Consider the following function definition, `f` is a function with a single parameter `x`. When the call `f(a+b)` is evaluated, `a+b` is stored inside a newly created promise.

```
> f <- function(x) substitute(x), list(a=1)))
> f(a+b)
# expression(1+b)
> substitute(a+b,list(a=1,b=2))
# 1+2
```

The call to `substitute(x)` extracts, from the promise bound to `x`, the expression passed in the call to `f`, i.e., `a+b`. The `substitute(exp,env)` function takes two arguments, the second is an object that can be used as an environment. This can be either a list of named values, a data frame or an actual environment. `substitute` will look for occurrences of each symbol from `env` in `exp` and replace them with their value taken from `env`.

```
> environment()
# <environment: R_GlobalEnv>
```

The above shows how to access the current environment, this can belong to the current function or, as above, the top level. Environments are nested, with each environment having a parent. This nesting is used when looking up names. When calling `new.env`, the default parent is the current environment. The chain of environments can be traversed with `parent.env`, ultimately one arrives at `emptyenv`. `parent.frame` and `sys.frame` grant access to environments further up in the calling stack, up to the global environment `.GlobalEnv`, and packages. One can also directly read, modify or create new bindings, given any environment:

- `env$v` and `get("v",envir=env)` read variable `v` in environment `env`;
- `env$v<-2` and `assign("v",2,envir=env)` write 2 in `v`. If not found, `v` is created in `env`.

Environments are often used as hashtables by programmers as they have reference semantics (other are copy-on-write) and have a built-in string look up. across function calls, or to create package namespaces.

```
evalq <- function(exp,env,enclos) eval(quote(exp),env,enclos)
eval.parent <- function(exp,n) eval(exp,parent.frame(n))
local <- function(exp,env,enclos) evalq(exp,new.env())
```

The three `eval` variants can be expressed as calls to `eval`. The `evalq` form quotes the argument to prevent it from being evaluated in the current environment. The `eval.parent(e,n)` form evaluates `e` in the environment of the `n`-th caller of this function. Finally, `local` evaluates an `exp` in a new environment to avoid polluting the current one.

2.3 Eval Usage in R

The R language was intended to be extensible, the combination of lazy evaluation, `substitute` and `eval` are the tools given to developers to this end. This API is slightly more complex than just passing a string, it is conceivable that this may discourage some casual users. `Eval` is also being used to reduce boilerplate code and provide convenience features for programmers. We now give some representative examples of its usage.

Intercession. A common use case for `eval` is to be combined with `match.call`. `match.call` walks up the call stack, captures the code that invoked the currently executing function, and returns it as an unevaluated expression. The pattern is to transform a call to some function `f` into a call to `g` with some arguments retained and others modified. As an illustration, consider the `vcpart` package's function `tvcglm` that is translated to a call to `tvcm` with two modifications to the argument list: argument `control` can't be missing and `fit` is set `"glm"`. The function ends with a call to `eval.parent` to ensure that the rewritten call is evaluted in the same environment the original call was.

```
tvcglm <- function(formula, data, family, control=tvcglm_control(), ...) {
  Call <- match.call()
  Call[[1L]] <- as.name("tvcm")
  if (!"control" %in% names(Call)) Call$control<-formals(tvcglm)$control
  if ("fit" %in% names(list(...))) warning("'fit'␣is␣ignored.␣")
  Call$fit <- "glm"
  return(eval.parent(Call))
}
```

This pattern is recognizable by the fact that the expression is a call and the target environment is that of the parent.

Code Generation. The more traditional use of `eval` is to execute code that was assembled by the programmer into a string. Here we show the method `plot` for class `sback` in package `wsbackfit`, simplified for explanatory purposes. The function takes a long argument list, the names of which are captured in the list `opt`. The string `stub` is composed of a subset of the arguments passed to this function; the variable is used to construct a call to `base::plot` which will draw a plot. *Note: This example is a bit messed up. What is var? Does the use of ... in the eval grab the arguments of this call? If yes does it not end up with xlab twice? HELP* The `parse` and `eval` combination is used by the the `source` function in the base library to load R code from a file in the current workspace.

```
plot.sback <- function(x,...) {
  opt <- names(list(...))
  stub <- paste(
   ifelse("xlab" %in% opt,"",paste(",xlab=\"",var,"\"",sep="")),
   ifelse("main" %in% opt,"",main.aux),
   ifelse("type" %in% opt,"",",type=\"l\""),
   sep = "")
  plot <- paste("plot(x.data,",stub,",...)",sep="")
  eval(parse(text=plot))
  ...
```

This pattern is recognizable by its use of `parse` to turn a string into an expression.

Debloating. `Eval` is often used as a means to reduce boilerplate code; simple and repetitive code can easily be replaced with judcious use of `eval`. For example, the `data.table` package uses `eval` to calls the `options` function with named arguments taken from a vector of strings. While the benefits are limited in this example, it is an attractive tool for programmers.

```
 opts = c("datatable.verbose"="FALSE", # ... many others
 for (i in names(opts))
   eval(parse(text=paste0("options(",i,"=",opts[i],")")))
```

This pattern is a special case of code generation, recognizable by the fact that `eval` is executed in a loop.

Trivial. When values are passed to `eval`, they are returned unchanged. They are an example of trivial uses of `eval`. Another trivial use is the empty expression, often found in JavaScript, but rare in R.

2.4 Previous Work

Richards et al. [**?**] provided the first large-scale study of the runtime behavior of `eval` in JavaScript. They dynamically analyzed a corpus of the 10,000 most popular websites with an instrumented web browser to gather execution traces. They show that `eval` is pervasive with 82% of the most popular websites using it. The reasons for its use include the desire to load code on demand, deserialization of JSON dataq and lightweight meta-programming to customize web pages. While many uses were legitimate, just as many were unnecessary and could be replaced with equivalent and safer code. They categorized inputs to `eval` so as to cover the vast majority of input strings. Restricting themselves to `eval` in which all named variables refer to the global scope, many patterns could be replaced by more disciplined code [?, ?]. The work did not measure code coverage, so the numbers presented are a lower bound on the possible behaviors. Furthermore, JavaScript usage in 2011 is likely different from today, e.g. Node.js was not covered by Richards. More details about dynamic analysis of JavaScript can be found in [?].

Wang et al. [?] analyzed use of dynamic features in 18 Python programs to find if they affect file change-proneness. Files with dynamic features are significantly more likely to be the subject of changes, than other files. Chen et al. looked at the correlation between code changes and dynamic features, including `eval`, in 17 Python programs [?]. They did not observe many uses of `eval`. Callau et al. [?] performed an empirical study of the usage of dynamic features in 1,000 Smalltalk projects. While `eval` itself is not present, Smalltalk has a rich reflective interface. The authors found that reflective are used in less than 2% of methods. The most common reflective method is `perform:`; it send a message that is specified by a string. These features are mostly used in the core libraries.

Bodden et al. [?] looked at usage of reflection in the Java DaCapo benchmark suite. They found that dynamic loading was triggered by the benchmark harness. The harness then executes methods via reflection, this caused static analysis tools to generate an incorrect call graph for the programs in DaCapo.

Morandat et al. [?] had a short section on the usage of `eval` in R. They found the it widely used in R code with 8500 call sites in CRAN and 2.2 million dynamic calls. The 15 most frequent call sites account for 88% of those. The `match.arg` function is the highest used one with 54% of all calls. In the other call sites, they saw two uses cases. The most common is the evaluation of the source code of a promise retrieved by `substitute` in a new environment; e.g. as done in the `with` function. The other use case is the invocation of a function whose name or arguments are determined dynamically. For this purpose, R provides `do.call` and thus `eval` is overkill.

3 Methodology

This section explains our methodology for selecting the corpus that will be analyzed and how we implemented our dynamic analysis.

3.1 Corpus

Our corpus is assembled out of two set of programs, one consisting of popular libraries obtained from the Comprehensive R Archive Network[4] (we refer to it as **CRAN**) and the other from the Kaggle data science competition (written **Kaggle**). The intent here is to contrast code written by experienced developpers (CRAN) with code authored by typical end users of the language (Kaggle).

All the code in our is runnable, i.e. we have scripts that invoke the CRAN packages and the Kaggle programs have input data. Clearly code coverage will vary; as is the case for any dynamic analysis, the quality of results is predicated on how representative the executions

[4] CRAN is the largest repository of R code with over 16.2K packages. It receives about 6 new package submissions a day [?]. Unlike other open code repositories such as GitHub, CRAN is a curated repository. Each submitted package must abide to a number of well-formedness rules that are automatically checked asserting certain quality. Most relevant for this work is that all of the runnable code is tested and only a successfully running package is admitted in the archive.

we can observe are with respect to all possible runs of the corpus. To mitigate the threat to validity attached to limited coverage, we obtain clients that will generate multiple *runs* of CRAN packages. In this context, a run is the execution of a CRAN package or Kaggle program with a particular set of input values.

CRAN Packages. The packages that are included in this study are the top 500 packages based on their reverse dependencies. The set of reverse dependencies for some package P is transitive closure of packages that import P. The hypothesis underlying that choice is that packages with a large set of reverse depencies are likely to be higher in quality and their tests will provide better coverage. The packages are implemented in R with some C and Fortran code. Using `cloc`, we measure a total 2M lines of R code and 2.2M lines of native code. For each package, we use its runnable code (tests, etc.) to compute code coverage. On average, the cde coverage is 55.7% which is acceptable without being exhaustive. Figure **??** shows these packages, the size of the dots reflects the project's size in lines of code. The x-axis indicates code coverage in percents and the y-axis gives the number of call sites to `eval` that were traced, in log scale. Dotted lines indicate means. Packages with over 50 eval call sites are named.

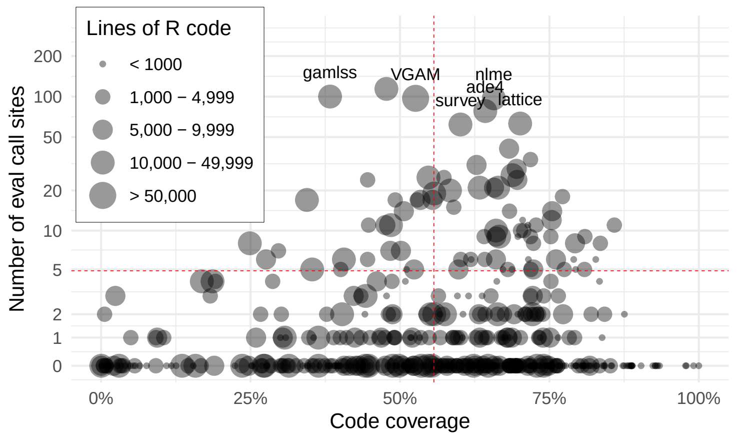

Figure 1 CRAN packages

To analyze a package, we need client code that invokes its methods. There are three sources of built-in runnable code that come with each CRAN package: *tests, examples* and *vignettes.* They are, respectively, traditional unit tests, code snippets from the documentation, and long-form use-cases written in Rmarkdown. Examples and vignettes are automatically extracted and turned into scripts, their input is bundled with the package. Most tests had to be discared due to a limitation of our pipeline: the `testthat` harness uses `eval` and thus causes the entire test to register as an `eval` call. The selected packages are bundled with 16,645 programs; 16.3K examples and 391 vignettes.

Kaggle Scripts. Kaggle is an online platform for data-science and machine-learning. The website allows people to submit data-analysis problems, users compete to find the best solution. The solutions are uploaded to the platform as either plain scripts or notebooks. We chose one of the most popular competition, predicting the survival of passengers the Titanic[5]. Unlike CRAN, Kaggle is not curated. After downloading the 2,890 solutions and extracting

[5] `https://www.kaggle.com/c/titanic`

the R code, we found that 1,042 were duplicates. From the remaining 1,848 solutions, 229 failed to execute. Next to various runtime exceptions, common problems were parse errors and misspelled package names. The final set contains 1,619 programs implemented in 119.1K lines of R code.

3.2 Analysis Pipeline

The results presented in this paper are the result of an automated analysis pipeline that acquires the code of packages, extract metadata, executes programs, traces their behavior and summarizes the observations. Figure ?? shows the main steps of the pipeline along with approximate time to execute each step, the data size, and the number of elements manipulated by the stage. Timings are for runs on an Intel Xeon 6140, 2.30GHz with 72 cores and 256GB of RAM.

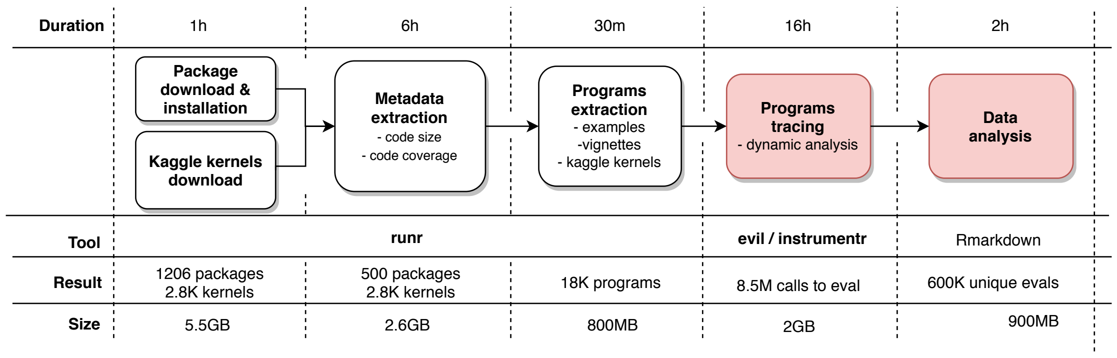

Figure 2 Analysis Pipeline

The first step in the pipeline consist of downloading CRAN packages along with their dependencies and acquiring Kaggle programs with the help of a web crawler. The second step, is to compute code size and coverage metrics for the CRAN packages (we use the *covr* package for coverage). The third step consist in extracting runnable programs from packages: i.e. the tests, examples and vignettes. Each extracted program is wrapped into a call to our dynamic analyzer — the tool is called *evil* for eval inspection library. This step is needed to ensure that we record `eval` usage only for the target package. Without this, the data would include eval calls from the unit testing frameworks as well as from bootstrapping R virtual machine itself. To avoid any interference, each program is run in its own R instance. The fourth step in the pipeline is to perform dynamic analysis for each run of a CRAN package or Kaggle program.

The dynamic analyzer builds upon the dynamic analysis framework, *instrumentr* that we have implemented to enable us to write dynamic analysis logic in R. *instrumentr* serves as an intermediary between *R-dyntrace* and *evil*, it intercepts the hooks exposed by *R-dyntrace* and attaches R functions exported by *evil* as callbacks. The *evil* callbacks execute on corresponding interpreter events.

The data extracted by *evil* from each program is concatenated, cleaned and summarized in the post-processing phase by custom R scripts. Finally, the summarized data is analyzed in RMarkdown notebooks to gather insights. Apart from the figures, the data points included in the paper are also generated by RMarkdown notebooks as latex macros.

The *evil* framework is implemented as a R package in 2K lines of R and 400 lines of C++ code. *instrumentr* is an R package implemented in 2.5K lines of R and 6K lines of C++

code. It internally uses a modified R interpreter, *R-dyntrace* [?], that exposes hooks from within the interpreter implementation for events of interest.

All steps of this pipeline are parallelized using GNU parallel [?] and orchestrated by GNU make. To schedule and parallelize extraction and analysis of programs, we use the *runr* package. Furthermore, *runr* gracefully handles and reports failures across large-scale program runs which greatly aids debugging of the analysis pipeline.

Limitations. Dynamic analysis can only observe calls that are triggered by the program's input. We believe that focusing on R packages with high code coverage does mitigate this to some extent. The results we report here were obtained with R's bytecode compiler turned off, this should not affect the results as the compiler does not optimize `eval`.

. We turn off the bytecode compiler for this study. The bytecode compiler can also call `eval`. We do not get source locations for 285K `eval` calls. In these cases `eval` is either passed as an argument to a higher-order functions or is defined in a function returned by a higher-order function and the R parser does not retain location information for `eval`. However, this is a meager 3.3% of all `eval` calls and is unlikely to affect our analysis. We ignore calls to the native `eval` function exposed by R. We also ignore the `rlang::tidy_eval` function which uses native `eval` internally because `rlang` is used to implement a DSL for data analysis in R. It introduces a new first-class promise object called `quosure` for which it implements special evaluation support in `tidy_eval`.

4 Usage Metrics

To provide a picture of `eval` usage in R, we recorded 20,376,174 invocations of `eval` and its variants in the CRAN corpus. These calls were obtained by XXX runs, with runs that did not trigger `eval` discarded. Out of the 6,126 CRAN packages with `eval` call sites, we observed calls in 200 packages, the difference can be chalk down to incomplete code coverage and occasional failures due to our pipeline.

Figure **??** shows the number of `eval`s generated by each package over all runs that invoked functions in that package. Out of the XXX packages in the corpus, XXX call have a low frequency of `eval` calls, with fewer than 100 of them. On the other hand, XXX packages call eval more than 1,000 times, with one package (???) calling it XXXX times.

`0 – 10`	32	`1K – 100K`	26
`11 – 100`	44	`100K – 1M`	7
`101 – 1K`	45	`1M – 10M`	2
`1K – 10K`	43	`10M – 100M`	1

Figure 3 Call frequency