What We Eval in the Shadows

A large-scale study of eval usage in R programs

ANONYMOUS AUTHOR(S)*

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. This paper studied 240,327 scripts extracted from 15,401 R packages, for a total of 49,296,059 calls to `eval`. We find that `eval` is indeed in widespread use; R's `eval` is more pervasive and arguably dangerous than what was previously reported for JavaScript.

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 [McCarthy 1978]. `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 transformations to the code using them. In practice, `eval` causes static analysis techniques to lose so much precision as to become pointless. For compilers, anything but the most trivial, local optimizations are unsound after the 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 [Cousot and Cousot 1977]. A reflective facility may have *any* behavior that can be expressed in the target language, *i.e.*, any legal sequence of instructions can replace `eval`. 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 lose all information about the program state and meaning of identifiers. This loss of precision can sometimes be mitigated by analyzing the string argument [Christensen et al. 2003] to bound its possible behavior, but when the string comes from outside the program, not much can be done. A frustrated group of researchers argued giving up on soundness and, instead, under-approximating dynamic features (soundiness) [Livshits et al. 2015]. 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 [Richards et al. 2010]. In 2010, 17 of the largest websites used the feature. In 2011, 82% of the 10,000 most accessed sites used `eval` [Richards et al. 2011]. 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 [Meawad et al. 2012]. 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 a language designed by statisticians for applications in data science [Ihaka and Gentleman 1996; R Core Team 2017]. 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 a browser, while R is used for statistical computing on desktops. JavaScript is a general-purpose language used by a vast 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] Thus, our goal is 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 codebase 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 the corpus is crucial. Our corpus has been constructed to reflect the levels of sophistication of the R community. We distinguish between *CRAN packages* (15,401 curated packages that pass stringent quality checks and are equipped with tests and sample data) and *Kaggle scripts* (7,931 end-user written programs that perform a particular data analysis task). It is reasonable to expect that `eval` usage differs 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? The results of our study suggest that `eval` is widely used for the implementation of the language, and in many libraries. End-user code makes less frequent and less sophisticated use of `eval`. In many ways, `eval` in R is as bad as it gets: it's varied, performs side-effects and reaches to many environments. By large, the motivations for `eval` relate to various forms of language extensions and meta-programming. `Eval` is used where other languages would provide macros. But, the expressive power of `eval` is higher as it can reach arbitrarily far back in the call stack.

[1] Consider that R is lazy like Haskell. We informally surveyed end-users and did not find a single user aware of this fact. Library developers, on the other hand, know and program defensively around laziness.

[2] Our infrastructure is open source and publicly available and will be submitted to the artifact evaluation committee.

2 BACKGROUND AND PREVIOUS WORK

This section provides a short introduction to R and the reflective features of the language; then looks at the semantics of `eval` in R and discusses design choices; lastly, this work is put in context.

2.1 R, Briefly

Morandat et al. [2012] gave a programming language-centric overview of the R language. They characterized it as a lazy, vectorized, functional language with a rich complement of dynamic features expressive enough to layer several object systems on top of the core language. Most data types are sequences of primitive values constructed by calling the combination function `c`. For instance, `c("Ha","bye")` evaluates to a vector of two strings, constants such as `42` are vectors of length one. To enable equational reasoning, values accessible through multiple aliases are copied when written to. Furthermore, values can be tagged by attributes; these are key-value pairs. For instance, the attribute `dim-c(2,2)` can be attached to the value at `x` by the call `attr(x,"dim")←c(2,2)`. In this case, the addition of this attribute turns `x` into a matrix. The class attribute gives a 'class', in the object-oriented sense, to a value. So, `class(x)<-"human"` sets the class of `x` to `human`; classes are used for method dispatch. Every linguistic construct is desugared to a function call, even control flow statements, assignments, and bracketing. All functions can be shadowed or redefined, making the language at the same time remarkably flexible and exceedingly challenging to compile statically as vividly detailed by Flückiger et al. [2019]. R uses a relaxed call-by-need convention for passing arguments to functions. Each argument is a thunk composed of an expression, its environment, and a slot for the result; these are called *promises*. To get the value of an argument, the corresponding promise must be forced. Once forced, the promise's result is cached for future use.

2.2 On the Expressive Power of Eval

While a data-to-code facility is available in many languages, some design choices affect the expressive power of `eval`. The key choices are the input format, the environment in which generated code is evaluated, and the reflective operations available to that code. Fig. 1 summarizes designs.

Language	Input	Scope	Reflective operations
Julia [Bezanson et al. 2012]	expression	toplevel	data
Java [Liang and Bracha 1998]	bytecode	classloader	data
JavaScript [Richards et al. 2011]	text	current, toplevel	data
R [Ihaka and Gentleman 1996]	expression	programmatic	data, stack, environment

Fig. 1. Design space of `eval`

The input to `eval` can be in any format convertible to code. JavaScript allows arbitrary strings to be used. Both Julia and R are more restrictive as they require expressions (or abstract syntax trees). Finally, Java is the most restrictive as its classloader only accepts complete classes in bytecode form.

The choice of the environment of `eval` is essential as it determines how much of a program `eval` can observe as well as the reach of potential side-effects performed by that operation. The most restrictive semantics is that of Java, where newly loaded code evaluates in the environment defined by the classes visible from the current classloader. Julia and JavaScript's strict mode limit `eval` to the global environment. Finally, R is the most flexible as any accessible environment can be selected and passed to `eval`.

The last degree of freedom is the expressive power of the code executed by `eval`. The main difference between languages lies in how much of the state of a program is accessible through reflective operations. Julia, Java and, JavaScript all allow some form of introspection on the data. R

is more flexible as it is possible to inspect the program's call stack. Thus, any environment in the program can be inspected and modified.

Given the above, the claim that R is amongst the languages with the most powerful `eval` seems plausible. The rationale for R's design seems to have been to expose as much of the language and its internals as possible in order to maximize expressivity. In R; `eval` is a key tool to extend the language and implement DSLs, it is also a replacement for macros. By contrast, the designers of Julia chose to limit `eval`. In Julia, only global variables can be side-effected, and environments cannot be readily manipulated. This is designed to shield optimized code from some of the most pernicious uses of the facility [Bezanson et al. 2018]. Furthermore, Julia provides a versioning mechanism, called world age, to ensure that any methods defined within an `eval` only become visible at well-defined program points and thus that recompilation does not have to occur when optimized code is running [Belyakova et al. 2020].

2.3 Eval in R

R exposes a rich reflective interface with functions, eval, evalq, eval.parent, and local. The following simplifies details not relevant to this paper, the interested reader should consult Wickham [2014].

```
eval ← function(e, envir = parent.frame(),
                encl = if(is.list(envir)) parent.frame() else baseenv()) ...
```

The most general is `eval`. Parameter e is the expression to evaluate, envir is the evaluation environment, and encl is used to look up variables not found in envir. The default value of the latter is either the caller's environment or the base environment `Eval` can take several types for e, for our purposes on the expression. These can be thought of as the abstract syntax trees returned by the parser. An expression can be obtained by calling quote or parse with a string:

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

Usually, expressions are obtained from promises using substitute. Consider a function called with x = a+b, then f returns an expression object.

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

The call to substitute(x) extracts the unevaluated expression from the promise x. Function substitute(e,envir) takes two arguments, the second is used to substitute free variables in e.

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

The environment function returns the current environment. Environments nest, each has a parent. When creating a new environment with new.env, the parent is the current environment. Environment chains can be traversed with parent.env, until emptyenv is reached. The top-level environment is .GlobalEnv, it has parents that represent the packages that have been loaded. One can also directly read, modify or create new bindings, given any environment:

- envir$v and get("v",envir=envir): read v from envir;
- envir$v<-2 and assign("v",2,envir=envir): store 2 in v, if not found, v is created in envir.

Environments are used as hash maps as they have reference semantics and a built-in string lookup. Functions parent.frame and sys.frame return environments further up the call stack.

```
evalq ← function(e,envir,encl) eval(quote(e),envir,encl)
eval.parent ← function(e,n) eval(e,parent.frame(n))
local ← function(e,envir,encl) evalq(e,new.env())
```

The three variants can be expressed with `eval`: evalq quotes its argument to prevent evaluation in the current environment, eval.parent(e,n) evaluates e in the environment of the n-th caller of this function; finally, local evaluates e in a new environment to avoid polluting the current one.

2.4 Previous Work

Richards et al. [2011] 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 data, 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 [Jensen et al. 2012; Meawad et al. 2012]. The work did not measure code coverage, so the numbers presented are a lower bound on possible behaviors. Furthermore, JavaScript usage in 2011 is likely different from today, *e.g.*, Node.js was not covered. More details about dynamic analysis of JavaScript can be found in [Gong 2018].

Wang et al. [2015] 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 [Chen et al. 2018]. They did not observe many uses of `eval`. Callaú et al. [2013] 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 methods 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 primarily used in the core libraries.

Bodden et al. [2011] looked at the 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.

Arceri and Mastroeni [2021] study `eval` in JavaScript from a software security point of view. The authors report that 53% of the malware they studied used `eval` as a means to obfuscate attack code or mount attacks. They propose an abstract interpretation-based approach to analyzing dynamic languages. One must construct a static approximation of the argument to `eval` and then analyze possible behaviors of the interpreter when evaluating the generated code.

Morandat et al. [2012] had a short section on the usage of `eval` in R. They found the that `eval` is 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 use 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 how we selected our corpus and how we obtained the reported results.

3.1 Corpus

Our corpus is assembled from three sources: the *Base* libraries bundled with the language implementation, packages hosted on *CRAN*, and scripts from *Kaggle*.

Base. The dataset contains 13 libraries performing basic arithmetics, statistics, and operating system functionalities. These libraries are bundled with R and executed pervasively. Together they contain 342 call sites to `eval` in 200 functions. Some of these functions provide basic functionalities such as package loading, and thus, there is hardly any code that does not invoke one of these.

CRAN. The Comprehensive R Archive Network (cran.r-project.org) is the largest curated repository of R packages. It hosts 16K packages with 6 new ones submitted daily [Ligges 2017]. Packages are authored by experienced developers and abide by well-formedness rules automatically checked on each commit. Each package comes with sample data. There are three sources of runnable code in a package: *tests, examples*, and *vignettes* – respectively, unit tests, code snippets from the documentation, and long-form use-cases written in RMarkdown. Examples and vignettes can be turned into scripts by extracting the relevant code. From 15,401 packages we extracted 240,327 scripts that contain 4,599,196 lines of code (1.5M in examples, 507.6K in vignettes and 2.6M in tests).

Kaggle. The Kaggle website (kaggle.com) is an online platform for data science. It allows users to submit and compete to solve problems. Solutions, called *kernels*, are uploaded as scripts or notebooks. While the quality of code is not uniform, each kernel is runnable, and maps to ones script. Input data is provided. We obtained 10,270 kernels. Since Kaggle is not curated, we used SHA-1 hashes to identify and remove 2,339 duplicate entries. The remaining 7,931 kernels are unique solutions to 219 competitions. They contain 665K lines of R code. Only 74 kernels call `eval`.

3.2 Pipeline

The analysis presented in this has been automated by a pipeline that acquires packages, extracts scripts, executes them, traces their behavior, and summarizes observations. Figure 2 shows the main steps along with their running time, data size, and number of elements manipulated. Timings are from a cluster of three 2.3GHz Intel Xeon 6140 servers, with 72 cores and 256GB of RAM, and shared OCFS network storage. The pipeline steps are:

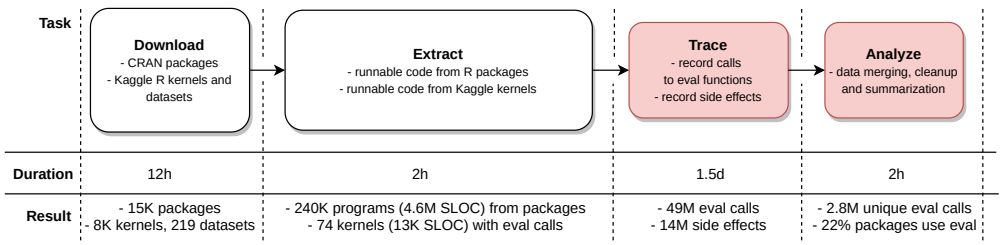


Fig. 2. Pipeline

(1) *Download.* Packages are downloaded from CRAN. For scripts, a web crawler retrieves code, and the Kaggle command-line tool gets data. Installation is complicated by native dependencies, which are not adequately documented and thus hard to resolve automatically.

(2) *Extract.* Given installed packages, the *genthat* tool extracts all runnable code snippets and turns each of these into a self-standing program [Krikava and Vitek 2018]. Some Kaggle kernels are already scripts and nothing more needs to be done. Kernels released as notebooks are processed by knitr which extracts runnable code. The body of each program is instrumented with calls to our dynamic analyzer to ensure that we only record calls to `eval` from the code of interest and not from bootstrapping or execution harness operations.
(3) *Trace.* Each program executes using our dynamic analysis tool, a heavily instrumented interpreter that captures calls to `eval` and many other fine-grained runtime events. Packages are run twice, once to capture `eval` calls originating from package code and a second time to capture calls coming from the base libraries. To avoid any interference, each program is run in its own process with the GNU R compiler turned off to avoid recording its execution.
(4) *Analyze.* Finally, the analysis output is merged, cleaned, and summarized in a post-processing phase driven by series of R scripts. The summarized data is then analyzed in RMarkdown notebooks to gather insights. All figures and numbers appearing in the paper are generated automatically. Figures are produced in PDF by ggplot2; numbers are exported as LaTeXmacros.

The pipeline runs in parallel [Tange 2018] orchestrated by a Makefile. Servers have identical environments thanks to docker images with all dependencies installed.

3.3 Dynamic Analysis

The dynamic analysis is performed by *R-dyntrace*, a modified R virtual machine based on GNU R 4.0.2 that exposes low-level callbacks for a variety of runtime events [Goel and Vitek 2019]. The tracer registers callbacks to all `eval` variants and a few additional functions to assist in locating the origin of `eval` arguments. For example, we taint the results of calls to `parse` and `match.call` (reflects on current call). We also capture calls to R API for dynamic code loading. Next, we subscribe to the events related to a variable definition and assignment, allowing us to record side effects that happen in environments while evaluating code in `eval`. One challenge was that R only provides source references for block surrounded by braces. Thus, a function whose body is eval(x) would no usable debug information, while the same expression surrounded by braces would be fine. This is unfortunately not easily fixed. We extended the dynamic analysis tool to attach synthetic source code references to all `eval` call sites by traversing ASTs. The tracer is implemented as an R package in 3.2K C++ and 1.3K of R code. For performance reasons, most of the tracing is done in C++. While in theory, the implementation is relatively straightforward, not so in practice:

— The lazy evaluation makes it challenging to analyze function arguments while tracing as prematurely forcing a promise might have dangerous consequences.
— While the R interpreter is implemented in C, most core functionality is in R. For example, package loading is implemented in R using `eval`. This makes it hard in the tracer to separate the `eval` that are essential to user programs from the accidental ones that are products of the way R implements its basic operations. This is even more so for the side-effects analysis.
— In the dynamic analysis, we run all the *runnable* code we obtained from CRAN and Kaggle—*i.e.*, actual code written by people with highly varying expertise in R and programming in general. This exercises a lot of the corner cases of the highly underspecified R behavior.

Limitations. Even with the extension described above, there are 42.1K `eval` calls without source references. This occurs when `eval` is passed as an argument to a higher-order function or when the `eval` call originates from native code. However, these missing source references account for a meager 0.09% of all calls; they are unlikely to affect our results. Other limitations are that we ignore calls to the native `eval` and to the alternate `rlang::tidy_eval` uses native `eval` internally. None of these limitations should invalidate our conclusions.

4 USAGE METRICS

This section focuses on CRAN packages and reports statistics about the usage of `eval`. We use the word *site* to an occurrence of a call site to the `eval` function in the source code and *call* to denote an observed invocation of the `eval` function.

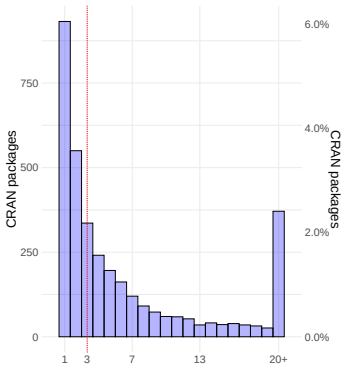


Fig. 3. CRAN `eval` call sites

CRAN. There are 38,619 `eval` sites in 3,488 packages. The proportion of packages calling `eval` is 22.6%. Over half of these packages have fewer than 3 sites, and with the exception VGAM, which has 2,376 sites, all packages contain fewer than 800 sites (*cf.* Fig 3, which shows a histogram of sites per package). These sites appear in 15,532 functions (2.5% of all functions in CRAN). For dynamic analysis, we run 240,327 programs extracted from 15,401 packages. Any run that does not exercise `eval` is discarded. This left 98,656 runs from 3,488 packages. There were 49,296,059 calls in 2,482 packages originating from 17,613 unique sites. In terms of coverage, the data exercised 52.9% of sites, a coverage similar to the code coverage metric for the packages, which is 51.7%. The fact that not all sites are exercised can be chalked down to incomplete tests and occasional analysis failures (3% of programs crashed or timed out). Fig 6 shows the number of sites that were exercised; coverage is unequal. Figure 4 summarizes the frequency of dynamic calls to `eval`, with the left column being the number of calls and the right,

#calls	#pck	#calls	#pck
1 – 10	744	10K – 100K	117
11 – 100	796	100K – 1M	29
101 – 1K	522	1M – 10M	5
1K – 10K	268	10M – 100M	1

Fig. 4. Call frequency

	eval	evalq	eval .parent	local
Static sites	36,241	207	1,673	250
Exercised sites	16,412	8	1,064	129
Invocations	48.7M	1.1K	569.9K	39.5K

Fig. 5. Variants

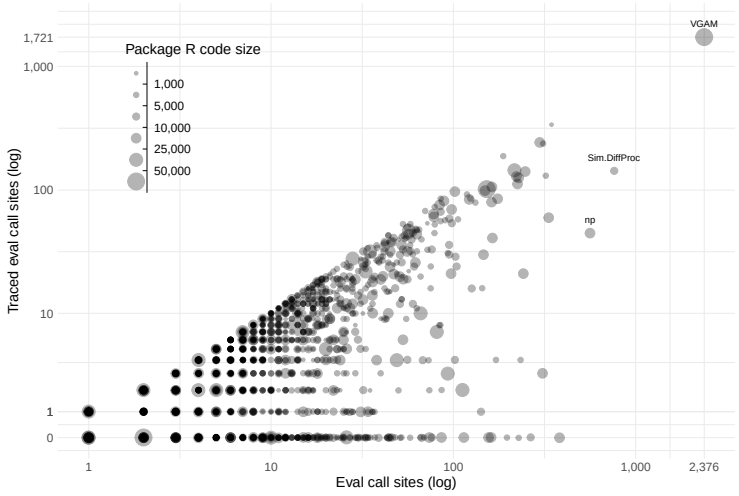


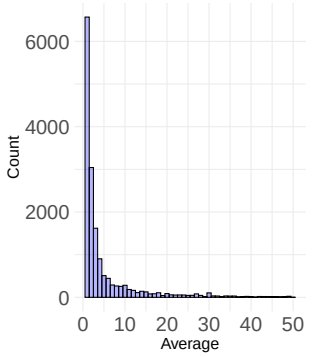
Fig. 6. `eval` call sites coverage of the 3,488 packages.

the number of packages in that range. There are 1,540 packages with low `eval` frequency, fewer than 100 calls, and a smaller, but still significant, number of packages, 420 to be precise, that use `eval` more than 1,000 times. Package ggplot2 makes 24,485,531 calls and thus accounts for over half of the observations. Figure 5 summarizes the use of variants of `eval`. For each of the four variants, the first row shows the number of sites in the corpus (*static*), the second row is the number of sites that were encountered during analysis (*exercised*), the last row is the number of calls (*invocations*). The overwhelming majority of sites and calls are to `eval` itself, eval.parent is rare, and both evalq and local are barely used at all. The difference between sites and exercised sites underscores the limits of code coverage.

Figure 7(a) shows normalized call counts per site; on the left are the average number of calls from a given site and given run, on the right are the counts of sites that fall in that range. For instance, 38 sites are invoked 2,000+ times per run. Larger numbers suggest loops or recursive contexts – this seems to be the exception as most evals, 16,274, are exercised 50 times or less. Figure 7(b) zooms in on low frequency sites. The x-axis shows normalized calls, and the y-axis is the number of sites for that value. Most low-frequency evals are invoked only once; about half as many are invoked twice; after that, the frequency quickly drops.

#calls	#sites	#calls	#sites
0 – 50	16,274	501 – 1000	115
51 – 100	429	1001 – 1500	65
101 – 250	369	1501 – 2000	29
251 – 500	181	2001 – 3000	38

(a) All



(b) Small

Fig. 7. Normalized calls

`Eval` takes any value, but if its argument is not an expression, it is returned unchanged. Expressions account for 90.2% of arguments. More specifically, 53.2% are `symbols` (single variables such as `x`), 26.6% are language objects (function calls), and 10.4% are expression objects (lists of expressions). Further inspection reveals that most symbols, 93.1% to be exact, come from a single site in the ggplot2 package and have the value _inherit.[3]

To estimate how much executable code is injected through `eval`, we measure the number of nodes in the expressions; for example, `x+1` counts as 3. We also measured string lengths of unparsed expressions, but these measurements were dominated by the size of data objects which could range in the MBs. The median argument size is 1 (due to the symbols), and the average is 4.5 nodes. The largest `eval` input observed is 63,265 – a significant chunk of code. Figure 8 shows the distribution of sizes for arguments of fewer or equal to 25 nodes. The x-axis is the size of arguments in number of nodes, and the y-axis is the count of arguments with that size. The size drops rapidly, with few observations larger than 15 nodes. The long tail is omitted for legibility.

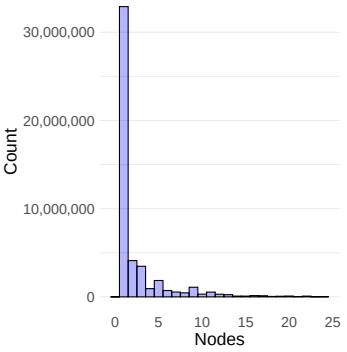
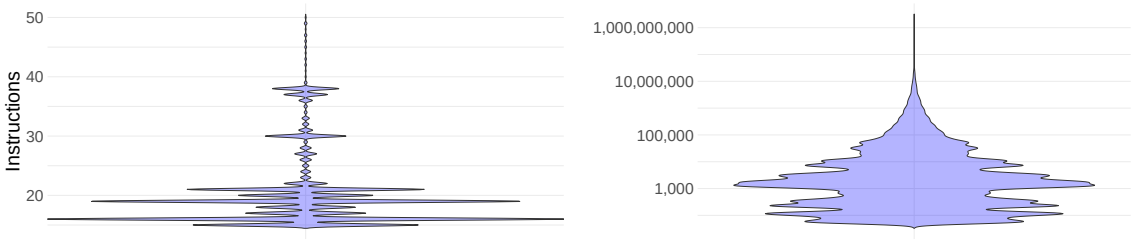


Fig. 8. Loaded code

To estimate the work performed in evals, we count instructions executed by the interpreter. Most invocations perform relatively little work, with 89.8% of evals executing 50 or fewer instructions. The

[3]This models inheritance in ggproto, one of the many object-oriented systems in R, used in the ggplot2 graphics library.

violin plot of Figure 9(a) corresponds to evals executing ≤ 50 instructions; it is dominated by trivial symbol lookups. Figure 9(b) has the work-intensive evals, which go all the way to 2.1G instructions.



(a) Small (b) Large

Fig. 9. Instructions per call

Base. We recorded 15,441,438 `eval` calls in the 13 base libraries. This comes from running 10% randomly selected programs from the 240.3K extracted programs from CRAN packages, covering 49.5% of the 342 sites. Most of the calls (95.3%) are to the eval function, and most arguments (96.4%) are expressions. However, unlike CRAN, the majority (92.7%) of the arguments are language objects. The median argument size is 4, which is more than for CRAN and the maximum size is 60,105, less than for CRAN. Small instruction counts (< 50) amount to 87.6% of calls. A single site in the match.arg function is responsible for 75.5% of all recorded calls. This function provides a convenient way for argument verification using partial matching. It is heavily relied on in R functions that use string arguments to parameterize function behavior, common practice in R. For example, in a body of `center<-function(x,type=c("mean","median","trimmed"))` one can get the value of type parameter using `type<-match.arg(type)`. A call `center(x,"trim")` will match type to `"trimmed"`. The challenge dealing with Base is that every program uses it. It can thus generate extreme amounts of data, but the data is quite predictable. From now on, we only mention Base when it surprises.

Kaggle. In total, 63,221 `eval` were recorded, all to `eval` function. Out of 220 sites from 74 scripts, only 61 sites in 26 scripts were hit; 30 scripts failed to run, and the others did not exercise the `eval` sites. This is partially expected as Kaggle code does not need to abide by any checks. Upon a manual inspection, we observed that indeed, the failing scripts were of poor quality, often not finished, using misspelled package names, hard-coded file paths, or accessing missing files. Most call sites are invoked once. Only one site is called 34,091 times. Expressions amount for 94.1% of arguments, there are very few symbols (0.1%). Most expressions result from calls to parse; thus, most evals start with strings. The median argument size is 7, which makes sense, as few arguments are only symbols. The largest argument is 71. The distribution of instructions per eval is similar to CRAN; 41.6% of evals execute fewer than 50 instructions. Manual inspection of `eval` usage in Kaggle suggests that it is consistent with the data obtained for CRAN. We do not discuss it further.

Discussion. `Eval` is central to R's implementation as it is omnipresent in the Base library. That library is, in turn, used by every single package. To avoid Base dominating the result, we excluded calls to it from the results. CRAN Packages, which are typically developed by experienced programmers, make regular and varied use of `eval`. They represent our most interesting data set as these packages result from over 20 years of contributions by thousands of authors. The code in packages is well maintained and relatively well tested. Finally, Kaggle scripts are often written by less sophisticated users and likely perform more straightforward tasks, thus having a lesser need for `eval`, and its uses originate from strings as these are easier to manipulate for end-users.

5 A TAXONOMY OF EVAL

The previous section gave a quantitative view of `eval` usage; we now try to elucidate *what* it does.

5.1 The expression in eval

The expressions passed to `eval` vary widely. In order to categorize them, let us use a minimization function $min(e)$ which for a given expression e returns a normal form that abstracts incidental details allowing the reader to focus on the structure of the evaluated code. The minimization function performs constant folding of arithmetic and string expressions for base operators, e.g. $min(1 + 1) = V$, value simplification, $min(c(1, 2, 3 + 2)) = V$, variable absorption, $min(x + y) = X$, function absorption, $min(g(f(x), h(z))) = F(F(X))$ and a number of other simplifications. Table 1 gives the 10 most frequent forms; #sites and %sites are, respectively, the number and ratio of sites receiving arguments of that form; #packages is the number of packages with that form; #operations is the median number of instructions performed by the interpreter; and %envir is the ratio of sites that evaluate in a function environment. The example column shows one sample expression e that normalizes to the particular form.

$min(e)$	#sites	%sites	#packages	#operations	%envir	example
X	4,739	27%	1,021	2	48%	y+1
F(F(X))	3,752	21%	854	241	79%	gbov(mean(x), a-1)
V	2,842	16%	791	1	70%	c(42,21,0)
F(X)	2,195	12%	772	24	68%	seq_len(iters)
$	1,728	10%	510	7	79%	DF$B
model.frame	1,263	7%	510	2K	46%	model.frame(formula = Z ~ U)
F()	1,136	6%	475	5	78%	rgamma(3, 2, n = 10L)
FUN	1,132	6%	129	2	96%	function(x, y) x + 3 * y
<-	864	5%	275	4	79%	x[1, 2:3, 2:3] <- value
BLOCK	729	4%	149	16.4K	84%	{ write.csv(iris, tf) ; file.size(tf) }

Table 1. Minimized expressions

We detail these forms and discuss their implication for the behavior of `eval`.

$\boxed{min(e) = V}$

Expressions that represent values are frequently passed to `eval`– they occur in 17% of sites. The majority of those, 74.5%, are inline constants (integer or double vectors). The rest trivially evaluate to a value, *e.g.*, 1+1.[4] In our corpus, 1,409 call sites only ever see a simple value, such as:

```
f1 ← eval(paste("A~",paste(paste(names(X[,-1])),collapse="+")))
```

where the argument to `eval` is a string which `eval` simply returns. Of twenty randomly selected evals, 65% of them need `eval` for some inputs. Either we are dealing with a value that needs to be constructed dynamically, or the value is a default case that sometimes is replaced by a more interesting expression. This form is usually evaluated in few interpreter steps; in fact, the median is only 1. The environment in which they evaluate is mostly irrelevant.

$\boxed{min(e) = X}$

Variable lookups are the most common form; they are found in 28% of the sites. This form includes simple variable reads, *e.g.*, x, which are 50.6% of X. The form also subsumes V, so it includes a mixture

[4] True as long as base functions such as + are not redefined. They typically are not, but this is a limitation of this categorization.

of arithmetic expressions, *e.g.*, x+y+1. The operations allowed are limited to built-in arithmetics. It is noteworthy that, while most Xs evaluate in a single step, the variable can be bound to a promise, and accessing it may trigger evaluation of that promise, thus resulting in an arbitrary amount of computation. The median number of interpreted operations is 2, suggesting that it is not the common case. Lookups are often evaluated in constructed environments; as few as 48% of these expressions are evaluated in a function environment.

$min(e) = \$$

This form extends X to include lookup with the dollar operator, *e.g.*, x$f, and vector indexing, *e.g.*, x[42] or x[[24]]. As with X, we allow arithmetics and values in this form. Lookup occurs in 10% of the sites. The interpreter evaluates 5 operations on average; the minimum is 3 operations. This is typically used in a function environment, 79% of the time to be precise.

$min(e) = <-$

This form includes both assignment operators, the direct assignment <-, and assignment to the parent environment <<-, the assign function and the $ form. Assignments occur in 5% of the sites. They represent the most obvious source of side effects. The median count is 4 (min=3).

$min(e) = F()$

This form captures simple function calls, *e.g.*, f(2), with neither variables nor assignments. More specifically, it allows for variables in the function position but not in arguments. Usually, looking up function names does not trigger computation, but that is not a given if the function is returned by a promise or if the function name is shadowed by a promise containing a value, then the computation will occur. This form occurs in 6% of sites and typically does not perform much work.

$min(e) = F(X)$ $min(e) = F(F(X))$

These forms allow for function calls whose arguments may include variable references. The latter allows nested calls. Assignments are excluded from this form. They occur in, respectively, 14% and 20% of the sites in the corpus. Together they are the most frequent forms. The median numbers of interpreter steps are, respectively, 24 and 241. Also, 68% and 79% of these expressions run in function environments.

$min(e) = FUN$

This form captures expressions that define functions, *e.g.*, function(x) x+1, and do nothing else. FUN occurs in 6.4% of sites. Evaluating a function definition is done in 2 interpreter steps; the data does not record the work performed by the interpreter when the generated functions are eventually run. In addition to FUN, 10% of sites have function definitions nested in other expressions. This gives an idea of the use of higher-order functions.

$min(e) = BLOCK$

This form captures multi-statement code blocks, which occur in only 4% of sites. These are huge expressions; we do not inspect the contents of the blocks. The median number of executed operations is 16.4K. They typically run in function environments.

$min(e) = model.frame$

The model.frame function returns a dataframe resulting from fitting the model described in a given formula. This form subsumes F(F(X)), FUN, and assignments. It is the single most popular function invoked from `eval`; it occurs in 7% of the sites. Each call does quite a lot of work with a 2K instructions median.

Consistency. It is interesting how many different forms any given site sees. The more forms, the harder it is to characterize behavior at that site. Luckily, 87.4% of sites only see a single form. In other words, a compiler could correctly predict the *shape* of an expression by looking just at the first call in 87.4% of the sites. Few sites are highly polymorphic (8 or more different forms). These include the pipe operator of the magrittr package, used to compose functions.

Discussion. The variety of uses of `eval` is evidenced by the number of different forms. In comparison, JavaScript `eval` usage was more straightforward and more predictable as reported by Meawad et al. [2012], with 98.7% of the sites with only one form. Nevertheless, simple forms dominate in R also, and there are many cases where `eval` could be replaced by less powerful constructs.

5.2 The environments of `eval`

Contrary to the Javascript `eval`, which does not specify the environment in which to evaluate, the second argument of `eval` in R, envir, is dedicated to that. This argument determines what is visible to the computation started by `eval` and the potential reach of its side-effects. Environments used by `eval` can be classified into the following four kinds:

— *Function:* environments for the local variables of some function currently active on the call stack. Obtained by calling parent.frame() or sys.frame().
— *Synthetic:* environments built from data structures such as lists, dataframes, or constructed explicitly with new.env, list2env, or as.environment. It also includes the empty environment.
— *Global:* environments in which scripts or interactive commands are evaluated.
— *Package:* environments of a loaded library.

As shown in Table 2, most calls evaluate in the scope of a function with global as a distant second. This means that most variable reads and most side-effects operate on local variables. But for which function? From the point of view of the caller of `eval`, Table 3 identifies that function by its offset on the call stack. Thus 0 is the direct caller, 1 is its parent, and so on. The data suggests that in 81% of cases, `eval` accesses the caller – this means the variables of the function where `eval` textually occurs are read and written to. It is noteworthy that 1.5% of sites evaluate three frames or above. This means that, in general, modular reasoning is impossible in R. To understand what any given code snippet may do requires fully understanding all the functions that may be called, transitively, from that snippet. The actions of `eval` happen at a distance. Finally, note that any site may see several kinds, but in 91% of the cases they have a single kind. It means that a compiler could predict the *kind* of environment after the first invocation.

Kind	#sites	%sites
Function	12,842	72.9%
Synthetic	3,506	19.9%
Global	3,015	17.1%
Package	77	0.4%

Table 2. Kinds per site

Offset	#sites	%sites
0	10.3K	81.5%
1	2K	15.5%
2	249	2%
≥ 3	193	1.5%

Table 3. Function offset

Parent	#sites	%sites
Function	2,497	71.2%
Package	1,362	38.8%
Global	378	10.8%
Empty	100	2.9%

Table 4. Wrapper envs.

#kinds	#sites	%sites
1	16K	91%
2	1.3K	7.6%
3	229	1.3%
4	11	0.06%

Table 5. Multiplicities

Global evals are likely split between intentional and accidental one. Direct references to the top-level, using globalenv() or .GlobalEnv, are rare; they occur in only 25 sites. Thus we suspect most uses of global are accidental. They arise from properties of our corpus, most scripts are top-level code snippets. Thus, global is often the caller of `eval` or close to it. The reason we make this point is that values stored in the global environment are visible to all functions and are not reclaimed by the garbage collector. So accidental uses may pollute that namespace.

Each synthetic environment has parent that is specified when calling new.env. When envir is a list or a data frame, `eval` uses its third argument (enclos). The parent is used to search for variables not found in its child (side-effects stay in the child). Table 4 shows parent kinds for synthetic environments. Most of them are functions, then come global and package.

Discussion. The data presented in this section is what one could expect. For *functions*, they represent the main user case for `eval`. Thus `eval` is really extending the behavior of that function by executing a dynamically selected piece of code in the function's scope. Typically, variables are read (mostly), written (less), but there are also some cases where new variables are injected or existing variables are deleted. Usually, this happens in the current function, but also in frames arbitrarily far up the call stack. One data point we lack is how the target environment was obtained. The expected case is that the body of a promise was modified and the resulting expression was evaluated in the same environment the promise originated from. Likely less frequent are cases where `eval` is provided the results of programmatically selecting some call stack. For *synthetics*, their relatively high frequency correlates to use cases where one wants to evaluate an expression in either a restricted environment or to use a data structure as an environment. For *global*, its relatively high frequency is likely an artifact of how the code of our corpus is is run. For *packages*, only 77 sites have this kind. This is probably for the best as mutating the bindings of a loaded package is improper and R tries to make it difficult.

5.3 The origins of `eval`

Where does the expression passed to `eval` come from? There are various means of creating that expression, which are associated with particular use cases. We classify them into three categories:

— *Constructed:* Expressions can be constructed by invoking the quote, enquote, expression functions. Function arguments are passed as promises, and substitute is used to retrieve the source expression associated with the promise.
— *Reflection:* This group corresponds to uses of match.call to reflectively capture the expression that invoked the current function.
— *String:* Expressions created from strings by invoking parse, str2expression, or str2lang.

Table 6 summarizes the expression provenance in our corpus. This data is obtained by dynamically tainting values as they are produced by the various sources. Due to technical reasons, there is some imprecision in the results. In particular, we are not able to classify all sites. Manual inspection of numerous examples suggests that errors are rare.

Strings could correlate with dynamic code loading. This is what the base functions source and sys.source do. We observed few calls (11 in total) that consume the result of calling parse on a file. Most of the calls build strings programmatically. We also identified one function invokeRestartInteractively that prompts the user for input, parses it, and passes it to `eval`. The use of strings seems to correlate with less sophisticated programmers; in Kaggle, 60.7% of sites use strings.

Origin	#sites	%sites
Constructed	7,901	44.9%
Reflection	4,524	25.7%
String	4,050	23%

Table 6. Provenance

Discussion. The origin data suggests that constructing expressions from strings is a minority of the use cases. Instead, the constructed category shows that most evals comes from code that was processed by the compiler, and may be slightly modified by the programmer before invoking `eval`. Both constructed and reflection categories roughly correspond to meta-programming. Some of these use cases could likely be replaced by macros if the R designers could be convinced to overcome their distaste for those.

5.4 The effects of eval

Eval may peform side-effects. From a compiler's perspective, we care about effects that can be observed — *i.e.*, variable definitions, updates, and removals that are visible after a call finishes. Knowing in which environments these side-effects happen, can help us determine how much of the compiler knowledge about the program will be invalidated. Our analysis records information about every environment. From the recorded data, we discard side-effects coming from unit testing frameworks.[5] They run their tests via eval, and thus everything becomes a side-effect.

From our 98K programs, we capture 14M side-effects from 2.9M eval calls (5.2%) in 3,091 sites (13.7%). Again, the challenge is to remove the accidental side-effects caused by the R virtual machine implementation which are not related to the user code. For example, the .Random.seed variable, which contains the state of the random number generator, is saved and restored from and to the global environment every time a statistical routine is called. Removing those leaves us with 7.9M side effects from 2.5M eval calls in 2,418 sites. These sites are part of 1,524 R functions (17.2% of all functions doing eval) in 694 packages. From these functions, 43 are responsible for 90% of side effects. Half of the side-effects come just from three functions: plyr::allocate_column (allocates space for a new data frame column), withr::execute_handlers (executes deferred expressions), and foreach::doSEQ (executes an expression on each element in a collection, possibly in parallel). Most of the eval sites (58.9%) do side effects in the environment specified by the envir parameter (*cf.* Section 2.3), 31.3% modifies other environments, and finally, 9.9% does both.

Table 7 shows environment kinds where side-effects happen. Function environments distinguish between *local*, the caller environment (offset 0), and *function* (offset >0). *Synthetic* represents constructed environments. *Object* is used to denote the environments attached to objects and classes in the S4 and R6 object systems. *Multiple* denotes cases where side-effects to more than one kind originate from one site. The table gives the number of call sites of eval (and the ratio) performing side-effect in a particular environment kind. The table also gives the number of functions in which these sites occur (and ratio).

Environments	#sites	%sites	#funs.	%funs.
Local	945	39%	269	25%
Synthetic	394	16%	248	23%
Object	240	10%	136	12%
Function	153	6%	62	6%
Global	106	4%	47	4%
Package	5	0.2%	0	0%
multiple	575	24%	332	30%

Table 7. Target environments for side-effects

Most sites, 76.2%, do all their side-effects consistently in one kind of environments. The same happens at the function level. Almost half of the sites (over a third of the functions) do side effects in either *Local* or *Object* environments. This gives a ray of hope for a hypothetical R compiler. Even though eval can do anything anywhere, the data suggests most effects are sane.

Table 8 shows recorded effects. There are over 5M updates. In terms of calls, we see assignments primarily and in terms of site definitions. This is expected. A subsequent eval call will turn a definition into an update. The most dangerous side effect is variable removal, as it means that after a call to eval some binding in some environments will disappear. While rare, this happens, but the vast majority comes from a single site (withr::execute_handlers). This makes sense as it is used to defer evaluation of an expression to after the function exit and thus used for clean up. It is used almost exclusively by the tidyselect package for removing the reference to the current quosure environment while interpreting a data frame column selectors.[6]

[5] In the corpus, we have RUnit, testthat, tinytest and unitizer testing frameworks.
[6] *cf.* https://tidyselect.r-lib.org/

Looking closely at the value types in variable updates, we observe that the majority of `eval` sites involve basic R vectors (53.8%) and lists (31%). From the compiler perspective, we would like to know how many sites change function bindings. In the corpus, this happens in only 62 sites; 75.8% of them do that in the local environment. We have manually inspected a few of these sites, but except for manual injection of parameters into a model.frame execution environment, we did not find a common use case.

Side effect	#events	%events	#calls	%calls	#sites	%sites
update	5,155,985	66%	2,674,085	61%	1,372	43%
definition	2,458,727	31%	1,455,335	33%	1,700	54%
removal	243,260	3%	237,711	5%	104	3%

Table 8. Types of `eval` side-effects

Discussion In JavaScript assignments in `eval` can happen in either local scope or less often, when called through an alias, in the global scope. In R, given the support for first-class environments, it can happen anywhere, making it `eval` more dangerous than it already is. However, the data suggest that first, side-effects from `eval` are not as widespread as in the case of JavaScript,[7] and that over half of them happen in a predictable environment.

6 USAGE OF EVAL

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. In this section, we focus on the *why*, giving real-world examples of the `eval` usage in the wild. All the examples are based on code from our corpus, often simplified to fit the space and increase clarity. For each example, we indicate the package and the function where it is located.

6.1 Shapes redux

In Table 1 we have shown the most common shapes of expressions passed to `eval`. Here, we illustrate them in concrete instances.

— *Variable lookup and values, and indexing* (X, V, $) constitute one of the most common use case for `eval`. Let's consider the subset function from the R base library, which allows, among others, to select variables in a data frame. For example, given a data frame df with three columns A,B,C, subset(df, sel=A) returns a subset of df with just the A column. The same would happen with subset(df, sel=1). However, thanks to `eval` it can also be also called with subset(df, sel=c(1,B,x)) returning a subset of the first column, column B and whatever index the variable x holds. It is implemented using: `eval(substitute(sel), nl, parent.frame())` The first argument returns the expression passed to sel, the second is the data frame in which this expression will be evaluated, and the last indicates which environment should be used to look up bindings not found in nl.
Another example is the most often called `eval` site in CRAN: `eval(`_inherit`, env)` defined in the find_super function in the ggproto object system. It looks up the variable '_inherit' to find the parent class. In the vast majority of cases, the variable is a symbol, so the `eval` seems superfluous. The `eval` is however necessary so one can specify parent classes using pkg_name::class_name, as the :: operator is a function call in R.

[7] [Richards et al. 2011] shows that in the *Interactive* scenario, `eval` in Javascript performs, store events can reach up to 40% of the events, and 7% to 8% of the `eval` do side effects in the global scope.

— *Assignments* (<-) appear in the cases where a developer wants to create or update a binding in a specific environment where either the name or the value come from some expression. Most cases originate from trivial code generation where the assign expression is assembled using parse or substitute, often in a loop. For example, the PopED::get_rse uses it to assign a number of variables in the current scope: `eval(parse(text=paste(default_args[[i]]), "←", i))`. There are also more sophisticated cases. The plyr::allocate_column function uses `eval` for a deferred assignment to fill missing values in a column. Package overture repeatedly evaluates R expressions capturing any assignments that happen in it as samples of Markov chains.

— *Function definition* (FUN) is mostly used in conjunction with code generation where new functions are synthesized using either parse or substitute. Some FFI frameworks use this to automatically generate R binding to native code. For example, Rcpp, which bridges R with C++, uses `eval` to generate R functions for C++ methods:
`eval(substitute(function(...) .CppObj$M(...), list(M=...)), env)`.

— *Function calls* (F(), F(X), F(F(X))) represent two very frequent patterns: *code generation* and *call reflection*. There are numerous cases for code generation. One occurring usage (present also in the Kaggle scripts) is to generate ad-hoc calls to libraries such as ggplot2 when one needs to parameterize data or aesthetics selection from a variable. For example `ggplot(df, aes(x=A, y=B))` maps X and Y axes in the plot to A and B columns in df. It might not be immediately clear[8] for a ggplot2 user how to modify the call to assign the axis columns from variables. Another case is the traditional use of `eval` as a mean to reduce boilerplate code; simple and repetitive code can easily be replaced with judicious use of `eval`. For example, the data.table package uses `eval` to call the options function with named arguments taken from a vector of strings:

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

Call reflection refers to the combination of `eval` with match.call, a function that returns the current call with all its arguments captured as expressions. This is used mostly in the following circumstances: (1) to record a call for later use in a possibly different environment, (2) pass most of the captured call arguments to another function, or (3) evaluate arguments of the function in different environments. This is widely used in relation to statistical modeling in combination with a model fitting function or with model.frame and related functions. For example the survival::coxph function[9] contains:

```
Call ←match.call()
tform ←Call[c(1,indx)] # only keep the arguments we wanted
tform[[1L]] ←quote(stats::model.frame) # change the function called
mf ←eval(tform, parent.frame())
```

— *Block* (BLOCK) can appear anywhere, but it is almost always the case that the block is only passed to `eval` rather than being directly part of in as in `eval({...})`. Essentially, the block denotes a fragment of a program to be evaluated in a particular way and in a particular environment. The latter is what makes it different from 0-argument closure.
There is a number of use cases: unit testing frameworks (*e.g.*, testthat, testit), code benchmarking (*e.g.*, rbenchmark, microbenchmark), running code in parallel (*e.g.*, foreach, doParallel), or deferring code execution (*e.g.*, withr).

[8] *cf.* https://stackoverflow.com/questions/22309285/how-to-use-a-variable-to-specify-column-name-in-ggplot
[9] Function fitting a Cox proportional hazards regression model.

6.2 Domain Specific Language

The ability to capture the current call as a language object (match.call()), extract the unevaluated expression from function arguments (substitute()) and `eval` with first class environment makes the R toolkit for building DSLs. There are numerous widely spread DSLs used in R. For example, data.table package defines a DSL for data frame manipulation using the subset operator [. This allows one to write compact queries such as flights[carrier == "AA", .N, by = origin][10] which will compute the number of flights done by American Airlines for each origin airport. Essentially, it uses substitute to capture the operand's expressions, transforms then appropriately and finally uses `eval` to execute the query.

Another example is the tidyselect package that defines a DSL for selecting columns in a data frame powering number of data manipulation operation in the data.table main competitor, the dply package. It uses eval to build an interpreter for evaluating column selectors such as select(df, starts_with("length") & !(name:mass)) which returns all columns in df that starts with the string length and is not between name and mass columns.

Another instance is the *symbolic differentiation support* provided by base R package *stats*, with two operators D and deriv. They support arithmetic operations and functions on real numbers such as sin, on expressions built with expression() or parse. In *MCMCglmm* (Monte Carl Markov Chain Generalised Linear Mixed Models), a Dtensor object is created: it contains expressions that are derived twice (with DD). To get a concrete value, it is evaluated in dataframe mu where symbolic variables are bound to actual numbers.

```
expr← expression(beta_1 + time*beta_2+u)
mu = data.frame(beta_1=0.5, beta_2=1, time=3, u=2.3)
D[i]← eval(DD(expr, name[unlist(comb.pos[i,])]), mu)
```

6.3 Unnecessary use of `eval`

Similarly to JavaScript, there are also unnecessary uses of `eval`. For example, the PerformanceAnalytics package contains a function chart.QQPlot that uses `eval` to resolve a string into function and another to call it and assign its results into a variable:

```
function (R, d="norm", dp, ...) {
  q.f ← eval(parse(text=paste("q",d,sep=""))); z ← NULL
  eval(parse(text=paste("z← q.f(",dp,",...)")))
}
```

In both cases, there is no need for `eval` and the function can be rewritten as

```
function (R, d="norm", dp, ...) {
  q.f ← get(paste0("q",d)); z ← q.f(dp, ...)
}
```

or even to a one-liner do.call(paste0("q",d), c(dp, list(...))).

The problem is that these cases are not easy to spot. Because of the limits of dynamic analysis, we only have a partial coverage and cannot determine if for example, `eval` sites classified as X, V will not be eventually called with an expression in a F(), F(X) or F(F(X)) category.

6.4 Discussion

`Eval` in R is used chiefly for meta-programming purposes and to access environments other than the local one. In Javascript, patterns are ad-hoc, such as json loading and parsing, and can often be

[10]The data.table subset operation is defined as DT[i, j, k], where j expression is used to calculate on a subset/reorder of rows from i grouped by k.

rewritten without `eval`, such as a function or method call, or an assignment to a local variable or an object property. For R, the data suggests that in some cases, the use of `eval` in the `X`, `V`, `$`, `<-`, `FUN` and the function calls related to code generation could replaced by more specific functions. For instance, variable lookup in any environment can be performed with `get`, and assignment in any environment, with `assign`. Building a call and executing can be done with `do.call`. Nevertheless, many of the observed `eval` usage in R, particularly, the ones related to statistical modeling and DSLs are more sophisticated than those reported for JavaScript.

Finally, let us mention an anecdotal use of `eval` as a means to bypass some checks performed by the R `CMD CHECK` tool. This tool performs a number of well-formedness rules before accepting a package to the CRAN repository. Many checks are performed statically by looking at syntactic constructs in the R code. Using `eval` allows one to bypass such checks and thus submit a package to CRAN that would otherwise be refused. For example, it checks that packages do not call the `unlockBinding` function.[11] Packages such as `data.table` circumvent such restrictions by using `eval` as the check is only performed statically.

7 CONCLUSION

In this paper, we provide a large-scale study of the use of the `eval` function in the R programming language. It is based primarily on a corpus of 240,327 scripts extracted from 15,401 R packages. As expected, `eval` is used mainly in libraries. We have found just a few `eval` call sites in scripts coming from a large corpus of Kaggle representing code written by R practitioners. We used an instrumented version of the R virtual machine, *R-dyntrace*, to capture all `eval` calls, including the relevant side effects that happen within these calls. Several insights can be learned from the data. First, `eval` is widely used in the R core packages that come bundled with the language. Because of that, there is hardly any R code that would not use it. `eval` is used to implement some basic functionality such as package loading. Putting the core packages aside, in CRAN, `eval` is used by 22.6% of the available 15,401 packages. The code executed by R `eval` has unusual expressive power. It can essentially modify anything anywhere, which is wary for both R developers and R compiler. The shapes of expressions passed to `eval` are extremely diverse, ranging from simple variable lookups to complex blocks or the creation of new functions. Luckily, most sites only see a single shape. `eval` unlashes its power and shows up its singularity compared to `eval` in other languages with the diversity of environments in which it can evaluate its expressions. While most of the time, the `eval` environment is the local environment, it is also often the global environment, a synthetic environment, or even a package environment. Luckily, as for the expressions passed to `eval`, the environment passed to `eval` for one given site is easily predictable with the first call to the site.

The expressive power of `eval` shines even more so in the case of side effects that can quite literally happen anywhere. The data, however, suggest that this is not the case. The majority of eval sites do not do any observable side effects, and when they do, it happens in a predictable environment.

The origins of the expression passed to `eval`, either constructed from language expressions, or stemming from reflection, or built by parsing strings. Reflection and constructed expressions dominate, which suggests the primary use of `eval` is meta-programming.

Similarly to [Richards et al. 2011], we started looking into evals in R with the ambition to replace them with other features that are more amenable to static analysis. The data, unfortunately, suggests that replacing `eval` is going to be a lengthy journey. Even if we disregard core packages as they are part of the language and the `eval` use is stable, in CRAN `eval` is used widely and in

[11] It allows one to open and modify loaded package namespaces.

more sophisticated ways than was the case in JavaScript. There the vast majority of sites fitted an extremely simple categorization based on simple regular expressions. Not so in R. While some usage, especially the one which calls `eval` with an expression parsed from string, can be categorized and often replaced by equivalent yet more disciplined and safer code, the case of `eval` coming from reflection is much harder.

Even though we reckon that `eval` has more expressive power than macros, as it can access runtime information such as its call stack, adding a macro system to R to replace `eval` for all meta-programming purposes would help static analysis as macros can be expanded at analysis-time.

REFERENCES

Vincenzo Arceri and Isabella Mastroeni. 2021. Analyzing Dynamic Code: A Sound Abstract Interpreter for *Evil* Eval. *ACM Trans. Priv. Secur.* 24, 2 (2021). https://doi.org/10.1145/3426470

Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. 2020. World Age in Julia: Optimizing Method Dispatch in the Presence of Eval. *Proc. ACM Program. Lang.* 4, OOPSLA (2020). https://doi.org/10.1145/3428275

Jeff Bezanson, Jiahao Chen, Ben Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. 2018. Julia: Dynamism and Performance Reconciled by Design. *Proc. ACM Program. Lang.* 2, OOPSLA (2018). https://doi.org/10.1145/3276490

Jeff Bezanson, Stefan Karpinski, Viral Shah, and Alan Edelman. 2012. Julia: A Fast Dynamic Language for Technical Computing. *CoRR* abs/1209.5145 (2012).

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/1985793.1985827

Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2013. How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. *Empir. Softw. Eng.* 18, 6 (2013). https://doi.org/10.1007/s10664-012-9203-2

Zhifei Chen, Wanwangying Ma, Wei Lin, Lin Chen, Yanhui Li, and Baowen Xu. 2018. A study on the changes of dynamic feature code when fixing bugs: towards the benefits and costs of Python dynamic features. *Sci. China Inf. Sci.* 61, 1 (2018). https://doi.org/10.1007/s11432-017-9153-3

Aske Simon Christensen, Anders Møller, and Michael Schwartzbach. 2003. Precise Analysis of String Expressions. In *Static Analysis Symposium (SAS)*. https://doi.org/10.1007/3-540-44898-5_1

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/512950.512973

Olivier Flückiger, Guido Chari, Jan Jecmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R melts brains: an IR for first-class environments and lazy effectful arguments. In *International Symposium on Dynamic Languages (DLS)*. https://doi.org/10.1145/3359619.3359744

Aviral Goel and Jan Vitek. 2019. On the design, implementation, and use of laziness in R. *Proc. ACM Program. Lang.* 3, OOPSLA (2019). https://doi.org/10.1145/3360579

Liang Gong. 2018. *Dynamic Analysis for JavaScript Code*. Ph.D. Dissertation. University of California, Berkeley. http://www.escholarship.org/uc/item/7n30n4kd

Ross Ihaka and Robert Gentleman. 1996. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics* 5, 3 (1996), 299–314. http://www.amstat.org/publications/jcgs/

Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/2338965.2336758

Filip Krikava and Jan Vitek. 2018. Tests from traces: automated unit test extraction for R. In *International Symposium on Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3213846.3213863

Sheng Liang and Gilad Bracha. 1998. Dynamic Class Loading in the Java Virtual Machine. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/286936.286945

Uwe Ligges. 2017. 20 Years of CRAN (Video on Channel9). In *Keynote at UseR!* https://channel9.msdn.com/Events/useR-international-R-User-conferences/useR-International-R-User-2017-Conference/KEYNOTE-20-years-of-CRAN

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015). https://doi.org/10.1145/2644805

John McCarthy. 1978. History of LISP. In *History of programming languages (HOPL)*. https://doi.org/10.1145/960118.808387

Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. "Eval begone!: semi-automated removal of eval from JavaScript programs". In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*.

https://doi.org/10.1145/2384616.2384660

Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-31057-7_6

R Core Team. 2017. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. https://www.R-project.org/

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval that Men Do: A Large-scale Study of the Use of Eval in JavaScript Applications. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.1007/978-3-642-22655-7_4

Gregor Richards, Sylvain Lesbrene, Brian Burg, and Jan Vitek. 2010. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*. https://doi.org/10.1145/1809028.1806598

Ole Tange. 2018. *GNU Parallel*. Ole Tange. https://doi.org/10.5281/zenodo.1146014

Beibei Wang, Lin Chen, Wanwangying Ma, Zhifei Chen, and Baowen Xu. 2015. An empirical study on the impact of Python dynamic features on change-proneness. In *International Conference on Software Engineering and Knowledge Engineering*. https://doi.org/10.18293/SEKE2015-097

Hadley Wickham. 2014. *Advanced R*. Chapman & Hall. https://doi.org/10.1201/b17487