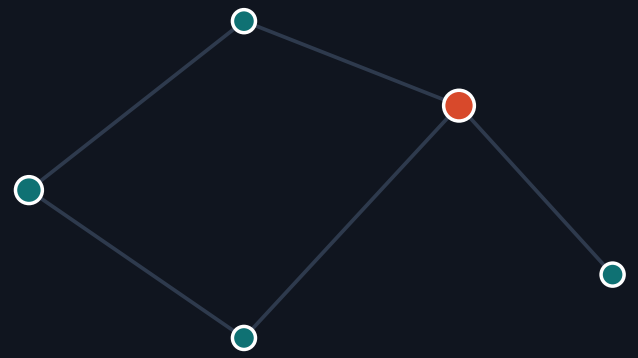
USER! 2026 · WARSAW · PACKAGE DEVELOPMENT

What If We Break Base R?

Bug propagation detection across CRAN



Szymon Maksymiuk · Warsaw University of Technology

THE BLIND SPOT

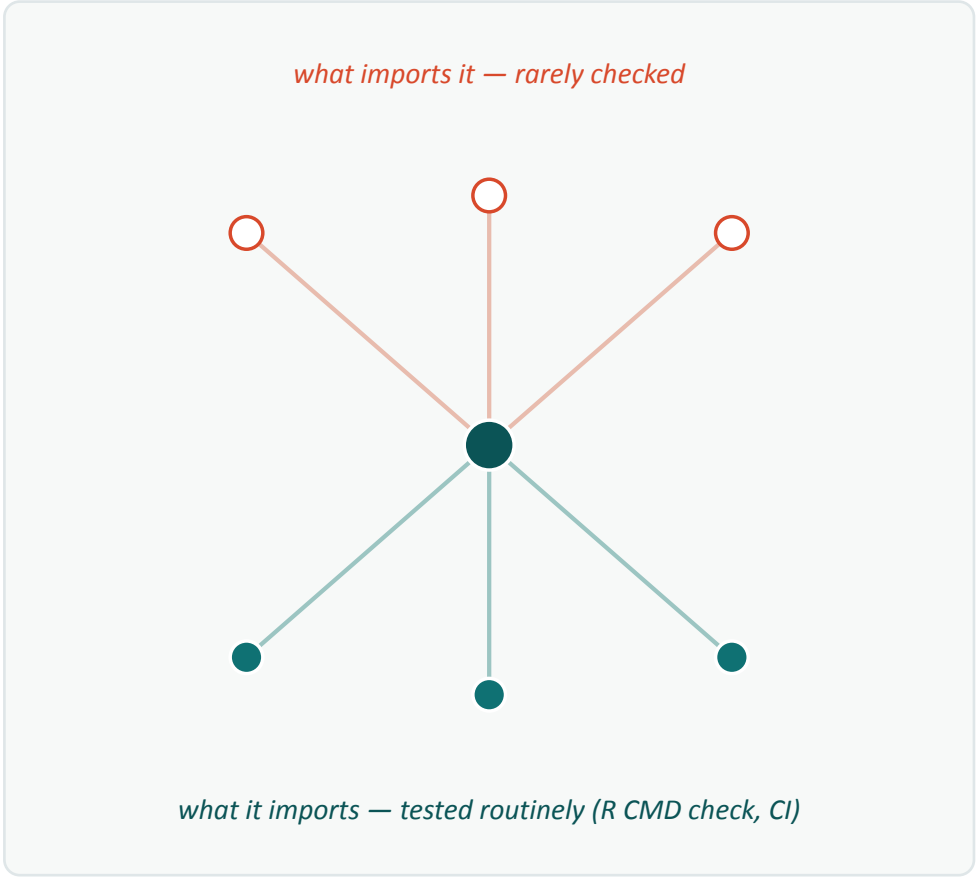
We test what we import — not who imports us

Every package is checked against the packages it depends on, we do that every time we run check. The other direction gets far less attention: if I break something in my package, would anyone downstream actually notice?

CRAN team does run reverse-dependency checks around releases. But it's rarely part of a maintainer's day-to-day workflow. Especially when they are not part of any organization.

npm, 2016. Removing left-pad — roughly a dozen lines of code — stopped builds across a large part of the JavaScript world. Nobody had asked what would happen if it went away.

CRAN reviews every package added and enforces strict dependency rules. Whether that's enough for a package network to be resilient?



THE EXPERIMENT

Make one base function slightly wrong. See what happens.

1 **Modify.** In a custom, local R session, one base function — say mean() — gets either a small Gaussian noise added to its output or, in the second variant, is made to fail outright..

2 **Check.** R CMD check runs over the packages that use that function, directly or indirectly, through their own dependencies — the same checks these packages run every day.

3 **Compare.** Results are compared against a run with unmodified R. Any difference — a failing test, a broken example, failure in installation, any meaningful difference in output counts as “detected”.

Everything runs locally, in disposable sessions. CRAN's infrastructure and published packages are never touched — the aim is to describe how detection currently behaves, not to infer conclusions about R packages quality.

METHOD OVERVIEW

One procedure, applied independently to thirteen functions

1

Select function

simple, numeric/logical, high-usage

→

2

Build network

top users + reverse deps, cap 1,100

→

3

Check (baseline)

R CMD check, unmodified R

→

4

Inject change

Gaussian noise, or forced error

→

5

Check (modified)

same packages, same checks

→

6

Compare & map

compare results

Repeated independently for each of the 13 functions. Each run produces its own package network and result set.

Selection criteria and the resulting function set

Candidates needed to be **widely used**, return a **single numeric or logical value**, and tolerate a controlled perturbation without changing object structure. Usage was estimated by parsing direct function calls across all CRAN packages (access date 20.05.2025).

`c()` 1,533,623 calls

`list()` 1,409,047 calls

CRAN's two most-used functions by this measure — neither is a candidate, since there is no well-defined notion of a "slightly wrong" concatenation or list construction.

Category	Functions	Network size range
Aggregate	sum, max, min, mean, var, median	481 – 1,035 packages
Analytic	log, sqrt, exp, abs, cos	265 – 1,098 packages
Logical	any, all	422 – 1,005 packages

Noise for logical functions is defined as a hard value flip rather than a randomised perturbation (see slide 13 for the empirical justification).

Bounding the reverse-dependency network (Algorithm 1)

```
Require: f (function), limit
i ← 1;  out ← ∅

while i ≤ Size(pkgs):
    deps ← reverse deps of pkgs[1..i]
    new_out ← out ∪ deps
    if Size(new_out) ≤ limit:
        out ← new_out
        i ← i + 1
    else:
        break

return out
```

Packages are ranked by direct usage of f. The candidate set grows outward through reverse dependencies until adding the next layer would exceed the size cap.

limit = 1,100 — chosen to balance network size against computation time

10,082 package-slots checked in total, across the 13 networks

Fun fact - Resulting graph would be a DAG, had circular Suggests dependencies been disallowed

Modifying a base function without rebuilding R

Base packages are built in as part of R itself, so rebuilding R for each of the 13 functions was not practical. Instead, the function binding is replaced at runtime:

```
min_noise <- function(..., na.rm = FALSE) {
  .Primitive("min")(..., na.rm = na.rm) + rnorm(1, 1, 0.5)
}

base <- getNamespace("base")
unlockBinding("min", base)
environment(min_noise) <- base
assign("min", min_noise, base)
```

Every R package, including base, has a namespace environment. Unlocking one binding and reassigning it in the namespace changes behaviour for the rest of that session without the need for rebuild or reinstall. Blunt brute force, but works!

Not every function tolerated this equally well: perturbing the + operator destabilised the session entirely, wgile max() required a small adjustment inside the checking code itself (it is used internally by R's own tooling).

Making the modification visible to R CMD check

1. .Rprofile runs at the start of every new R session, except in --vanilla mode — this is how the modification is applied automatically to new sessions.

2. R CMD check spawns each package check in an isolated subprocess that, by design, ignores startup files (for reproducibility).

3. The same .Rprofile that applies the modification also patches the check machinery for this one experiment, so the spawned subprocess picks up both.

This is a narrowly-scoped, temporary bypass used only within this sandboxed experiment — definitely not a general guide how to update base R behaviour.

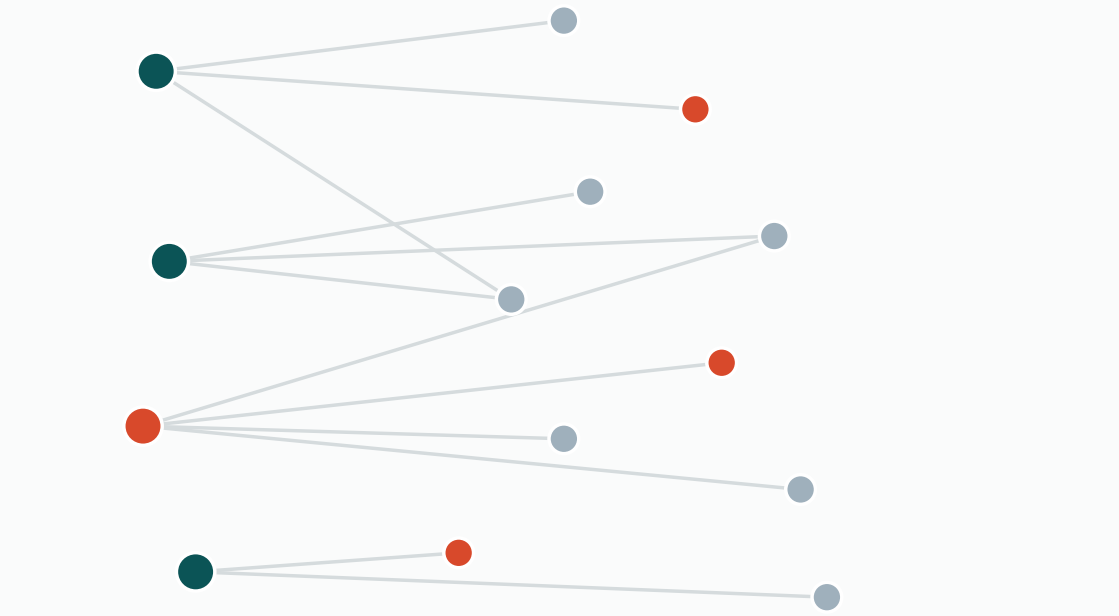
Also required

A modified copy of rcmdcheck (a dependency of checked), since it includes environment-modification safeguards similar to base R's — disabled here for the same sandboxed purpose.

The max() case

R's own checking tools call max() internally, so the checking code specifically was pointed at the untouched primitive.

Reverse-dependency networks have real internal structure



sqrt() — NETWORK SUMMARY

Packages in network	1,098
Mean degree	2.93
Network centralization	0.091
Flagged — noise	22.8%
Flagged — error	54.3%
Connected components for “broken” network	114 – 117
Mean degree “broken”	1.33 – 2.08
Packages with tests	38.5%

STEP 4 — RUNNING IT AT SCALE

Orchestrating roughly 40,000 R CMD check runs

13 × 2

functions × modification types (noise / forced error)

~40,300

individual R CMD check runs (10,082 slots × 2 × 2)

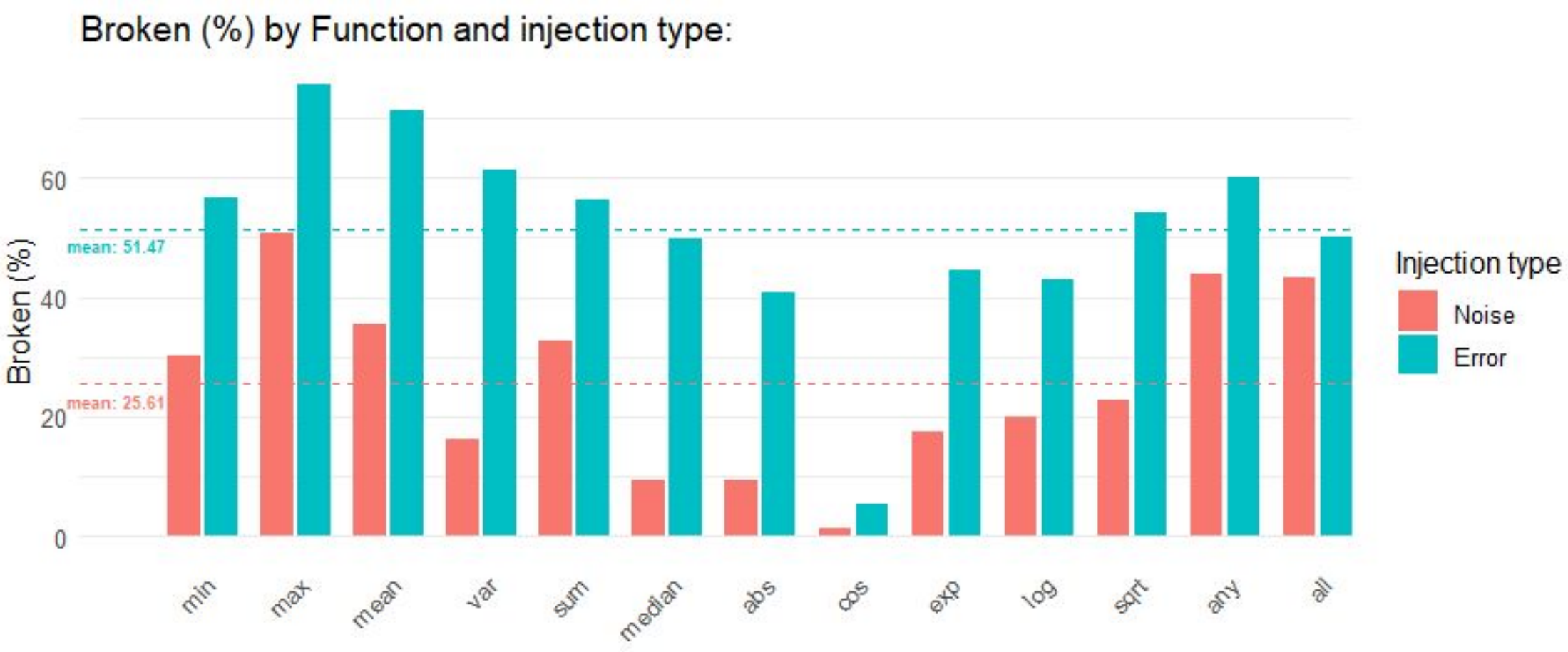
~500 GB

of raw check logs produced

Orchestration chain

`callr`	spawns an isolated “orchestrator” process with required base functions overridden
`checked`	runs and diffs R CMD check results across a package set (co-authored tool)
`callr`	spawns isolated R processes, each with a custom .Rprofile

Detection rates across the thirteen functions studied



Detection appears to vary by function category

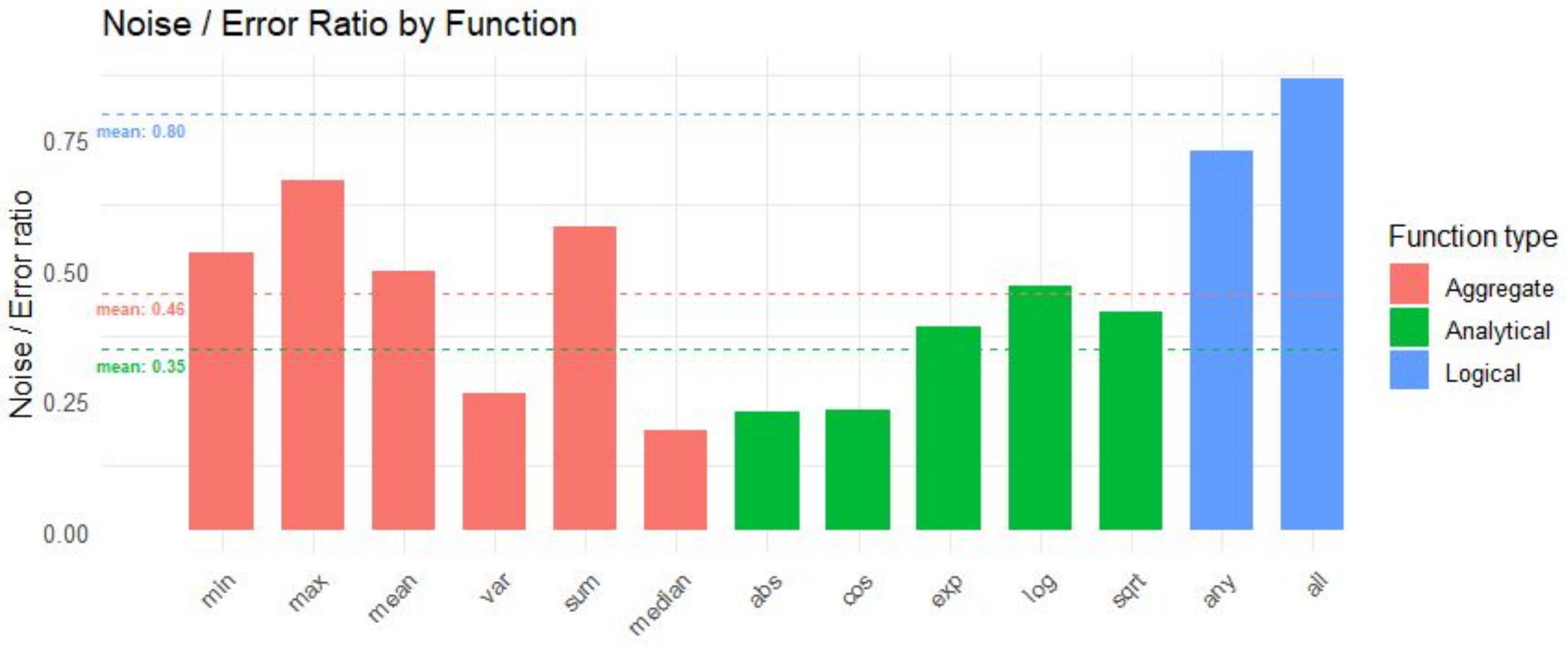
Category	Noise %	Error %	Ratio
Aggregate	29.1	61.8	0.46
Analytic	14.2	37.6	0.35
Logical	43.6	55.1	0.80

A tentative interpretation

Logical functions are often used directly as test assertions (e.g. expect_true()), where any deviation is binary. Numeric results can be absorbed by downstream arithmetic without necessarily failing a check. With only 13 functions, this is a plausible reading of the pattern rather than a confirmed mechanism.

`cos()` is an outlier (1.3% / 5.5%) despite a comparably large network (1,091 packages). We do not have a clear explanation for this from the data collected so far.

Detection appears to vary by function category



A few observations relevant to package development practice

1

Test coverage in these networks. On average, about 36% of packages in the studied networks include a dedicated test suite as part of their source; the remainder rely on examples or vignette code during R CMD check.

2

Assertion style. In a side-experiment on mean(), three noise magnitudes — centred at 0.1, 1, and 10 — were detected by the same set of packages. This is consistent with the predominance of exact-match assertions reported by Vidoni (2021), and suggests limited ability, from this measure alone, to infer severity.

3

Spread is not uniform. Connected-component counts in the affected subnetworks suggest detection does not spread evenly with distance from the perturbed function. Based on 13 functions that were tested.

4

Tooling reuse. The checked + callr pattern used here does not require rebuilding R and apart from the code itself does not require any setup. Could be used by anyone as part of their research into weird states of base R.

A percolation model of failure propagation

1 *Each dependency edge conducts with probability p (bond percolation)*

$$\mathbb{P}(A = a) = p^{|a|}(1-p)^{|E|-|a|}$$

2 *Detection given exposure occurs with sensitivity α*

$$\mathbb{P}(Y_v{=}1 \mid Z_v^A{=}1) = \alpha \qquad \mathbb{P}(Y_v{=}1 \mid Z_v^A{=}0) = 0$$

3 *Mean-field pseudo-likelihood, after a decorrelation approximation*

$$\widetilde{L}(p, \alpha; y) = \prod_{v \in V} \big(\alpha\, q_v(p)\big)^{y_v} \big(1 - \alpha\, q_v(p)\big)^{1-y_v}$$

4 *Message-passing (belief-propagation) update used to approximate qv(p)*

$$h_{u \to v}^{(k+1)} = (1 - s_u) \prod_{(w \to u) \in P(u \to v)} \big(1 - p + p\, h_{w \to u}^{(k)}\big)$$

$2^{|E|}$ subgraphs $\Rightarrow$ #P-complete

Exact evaluation sums over every possible "live-edge" pattern — approximated here with message-passing, borrowed from epidemic and statistical-physics models. Full derivation and per-function fitted values is a subject for a next useR (hopefully)..

NEXT STEPS

Final remarks

1 Broaden coverage

Apply the same pipeline to additional base R functions, and, where feasible, to other widely-imported infrastructure packages.

2 Cross-ecosystem comparison

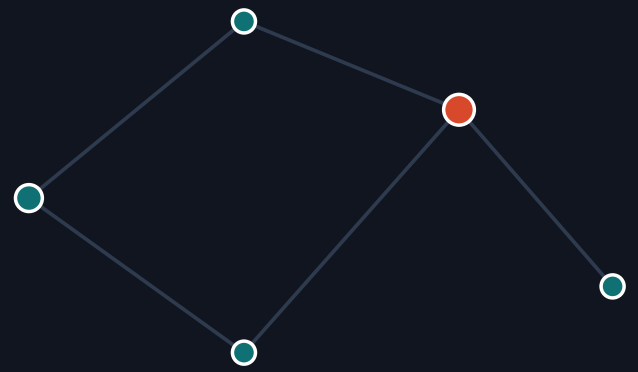
Compare against equivalent studies on PyPI and npm dependency graphs, where related work reports similar structural patterns (Decan et al. 2019; Gao et al. 2024; Mahon et al. 2025).

3 Community message

We should test more and better. Testing in R is improving with every year but the hard error results shows that there is still room for improvement. With how strict R is defined, we should be way more resilient and have better error capture ratio.

THANK YOU

Questions and discussion welcome.



Szymon Maksymiuk

Warsaw University of Technology

References

Decan, Mens & Grosjean (2019), Empirical Software Engineering 24(1).

Vidoni (2021), ICSE 2021.

Mora-Cantallops et al. (2020), J. Software: Evolution and Process 32(11).

Levin (2016), The Guardian, March 23.

Gao et al. (2024), ACM TOSEM 33(4). · Mahon, Hou & Yao (2025), arXiv preprint.