Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

vazul: An R Package for Analysis Blinding

Tamás Nagy
ELTE Eötvös Loránd University

Márton Kovács
Independent researcher

Alexandra Sarafoglou
University of Amsterdam

Abstract

Analysis blinding is a methodological approach designed to protect the confirmatory status of an analysis by concealing crucial test-relevant aspects of the data from the analysts. This article introduces the vazul package for R, which provides a comprehensive suite of tools for implementing masking and scrambling techniques. We discuss the theoretical foundations of analysis blinding, particularly as a complement or alternative to preregistration. The package allows researchers to maintain analytic flexibility while safeguarding against bias. By using vazul, data managers can easily generate blinded datasets that preserve essential distributional properties while obscuring associations or labels that could lead to biased decisions. We provide detailed examples of vector-level, data-frame-level, and row-wise blinding operations, and demonstrate how the package handles complex data structures such as hierarchical or repeated-measures designs.

Keywords: analysis blinding, R package, bias control.

1. Introduction

In data analysis, researchers face the challenge of maintaining objectivity and minimizing bias. A central concern is the considerable analytic flexibility present at various stages of the workflow, including data preprocessing, variable selection, model specification, and statistical testing. This flexibility has been described as the "garden of forking paths" (**?**), meaning that possibly hundreds or thousands of plausible, non-redundant analytic strategies can be applied to the same dataset (**??**). Crucially, different analytic paths can yield different results, and analysts may, intentionally or not, gravitate toward those that align with their expectations. The result can be biased or even spurious findings (**???**), which have eroded trust in science and contributed to replication failures across medical sciences, psychology, economics, and computer science (**???????**).[1] To ensure the reliability of empirical findings, it is therefore essential to prevent the idiosyncrasies of the data from feeding back into the formulation of

[1] But see **?**, **?**, **?**, and **?** for critiques of how replication failure is measured and of the design and validity of

the hypotheses being tested (HARKing, **?**) or from prompting researchers to exploit their analytic freedom to accentuate desired outcomes (**?**). In response, researchers have proposed methodologies to limit the influence of bias in data analysis. One such method is analysis blinding, which supplies analysts with an altered version of the real data that conceals biasing features until the analysis plan is finalized.

Analysis blinding, or blind analysis (**?**), is a methodological approach designed to protect the confirmatory status of an analysis by concealing crucial test-relevant aspects of the data from the analysts—for example, by scrambling dependent variables or masking key labels. The procedure typically involves two independent parties: a data manager, who has full access to the raw data and is responsible for implementing the blinding steps, and an analyst, who is tasked with developing the analysis plan. After the data manager applies the blinding procedure, the analyst receives the resulting modified dataset: the blinded data.

Working with the blinded data, the analyst can explore and preprocess the dataset and develop an analysis plan without being influenced by whether a particular analytic choice produces the desired result, as any patterns in the blinded data are, by design, only a product of chance. Importantly, although the test-relevant elements are concealed, other meaningful characteristics remain intact, including demographic information, secondary variables, and the distributional properties of both dependent and independent variables. As a result, the analyst retains all information necessary to tailor the analytic approach to the specific, and potentially unexpected, features of the data. Once the analysis plan has been finalized using the blinded data, the analyst is granted access to the raw, unblinded dataset, and the confirmatory analysis can then be carried out strictly according to the pre-established plan.

Analysis blinding ties in with other methodologies to minimize bias in the research cycle, such as single-blind or double-blind experimental designs, where participants and/or experimenters are unaware of certain aspects of the study to prevent bias in data collection (**?**). Here, the analysts is intentionally kept unaware of certain aspects of the data to prevent bias in data analysis, which is why the methodology is also referred to as triple-blind (**?**).

Although many blinding techniques are conceptually straightforward, such as scrambling outcome variables or masking variable names and factor labels, their practical implementation can become technically challenging once real research designs and data structures are considered. For instance, in hierarchical models, researchers may need to scramble the outcome variable to destroy the effect of interest while still preserving information within grouping levels, such as country-level means in cross-cultural datasets (e.g., **?**). Similarly, in repeated-measures designs, condition labels may need to be masked so that each participant's condition structure is preserved while permuting the mapping of conditions across participants to obscure the true experimental effect. These complexities make manual analysis blinding difficult to carry out consistently and correctly. To lower barriers for data managers and facilitate the broader adoption of analysis blinding in research labs, dedicated software tools, such as R packages, are needed to implement these procedures reliably.

To meet this need, the **vazul** package in R (`https://CRAN.R-project.org/package=vazul`) offers flexible and reproducible tools for implementing analysis blinding. The package is named after Vazul, a Hungarian prince of the 11th century who was blinded on the orders of King Stephen I of Hungary to render him unfit for the throne. Analogously, analysis blinding temporarily renders data unfit for the (unintentional) exploitation of researcher degrees of

some replication attempts.

freedom. The R package provides functionality for data scrambling as well as data and variable masking. In the remainder of this article, we introduce the methodology in more detail, describe the functionalities of the **vazul** package, and illustrate its application through practical examples. We conclude with a brief discussion and outline future directions.

1.1. Analysis Blinding: Safeguarding Against Bias Without Preregistration

Originally, analysis blinding gained traction in astrophysics in the early 2000s (**?**) as a means of safeguarding analysts against bias. Since then, the methodology has also been advocated in the social and behavioral sciences as an alternative or complement to preregistration (**??????**). This interest partly reflects the fact that analysis blinding addresses several of preregistration's major shortcomings. Preregistration can restrict analysts so rigidly that they are unable to adapt their analyses to unexpected peculiarities in the data. Although deviations from preregistered plans are both possible and accepted when transparently disclosed in the final manuscript, such deviations run counter to the spirit of preregistration, as they reintroduce data-dependent decisions that may bias the results. At the same time, text-based preregistrations, that is, the description of analysis plans in a preregistration template, often fail to be "specific, precise, and exhaustive'' (**?**, p.2), which leaves substantial degrees of freedom unaccounted for.

In contrast to preregistration, analysis blinding allows researchers to maintain flexibility in their analysis plans, as they can explore the data without being constrained by a rigid preregistered protocol. This flexibility is especially important in analyses involving advanced statistical modeling or preprocessing, where not all decisions can be anticipated in advance and researchers must make data-dependent choices. For instance, many psychological constructs, such as well-being, are measured with multi-item scales whose factor structure must be examined before further analysis; whether the items load onto a single factor or multiple factors determines subsequent analytic steps, including whether composite scores can be computed or whether a more complex measurement model is required (see e.g., **?**). The flexibility gained by analysis blinding enables researchers to develop analysis strategies that are appropriately tailored to the specific characteristics of the dataset without the need to anticipate and specify all eventualities in advance. Consequently, researchers who analyze their data with analysis blinding are less likely to deviate from their developed analysis plan compared to researchers who preregister their analytic strategy in a text-based format (**?**).

Moreover, similar in spirit to approaches proposed in code-based preregistration (**??**), analysis blinding naturally produces analysis plans that are specific, precise, and exhaustive, because they are written directly in code rather than described verbally in preregistration templates. As with preregistration, analysis scripts based on the blinded data can be uploaded or attached to preregistration platforms (e.g., the Open Science Framework) to provide a transparent record of the planned analysis before the data are unblinded. In short, analysis blinding can serve as a valuable alternative or additional safeguard to preregistered analysis plans: it effectively protects against bias while providing analysts with the flexibility needed to develop an analysis strategy that is optimally suited to their data.

2. Overview of the main functions

The package provides functions for two main types of analysis blinding:

1. **Masking**: Replaces original values with anonymous labels, completely hiding the original information.
2. **Scrambling**: Randomizes the order of existing values while preserving all original data content.

2.1. Choosing between masking and scrambling

When choosing between masking and scrambling for analysis blinding, the decision usually turns on what information must be concealed and what structure must be preserved for meaningful exploratory work. Masking is most appropriate when the core risk lies in revealing the identity or meaning of categorical variables—treatment groups, study arms, demographic categories, or experimental conditions. In these situations, replacing labels with arbitrary codes prevents analysts from inferring substantive content while still allowing them to run the full range of models that depend on category-level distinctions. Masking also accommodates cases where different variables require different relabeling schemes or prefixes, for example when several independent categorical factors need to remain distinguishable but substantively opaque. This approach is common in clinical trials, behavioral experiments, and preregistered confirmatory analyses where preserving the structure of factors is essential but knowledge of factor identity would bias analytic choices. Masking can be indispensable in situations where the variable names themselves carry substantive meaning. This occurs, for example, in network analysis, where variables become nodes with interpretable labels, or in factor-analytic work, where item names often hint at their psychological content. In such settings, simply scrambling values would not prevent analysts from inferring the underlying constructs, whereas masking removes this semantic cue entirely. The `vazul` package supports this use case directly by offering functionality for masking variable names alongside their values.

Scrambling becomes the better choice when the goal is to preserve the marginal distributions of numeric variables while severing their hypothesized associations. It is particularly useful in settings where analysts need access to realistic distributions—variance, skew—to make decisions about transformations, model families, or robustness checks, yet must remain blind to the relationship between predictors and outcomes. By permuting values within variables, scrambling prevents recognition of treatment effects, correlations, or temporal patterns, making it harder to inadvertently tune an analysis toward the desired result. It is often the preferred option in longitudinal studies, high-dimensional observational datasets where categorical relabeling would either destroy important numeric structure or be easy to reverse-engineer. Scrambling is generally the more versatile option, because it can be applied across variable types and adapts well to a wide range of study designs and analytic workflows. Its ability to break associations while preserving distributions makes it suitable for almost everything from tightly controlled experiments to complex observational datasets, including high-dimensional or mixed-format data.

Table 1: Comparison of masking and scrambling approaches for analysis blinding.

Aspect	**Masking**	**Scrambling**
Original values	Replaced with arbitrary codes	Preserved but permuted
Data distribution	Category counts preserved with different labels	Marginal distributions fully preserved
Associations between variables	Preserved	Broken (correlations, treatment–outcome links removed)
Risk addressed	Hiding the identity or meaning of categories or variable names	Preventing recognition of true relationships or effects
Best suited for	Categorical variables; variables whose names convey substantive meaning	Numeric or categorical variables; mixed-type data
Typical use cases	Clinical trials, behavioral experiments, preregistered analyses, network analysis, factor analysis	Longitudinal data, observational datasets, robustness checks
When essential	When variable names or category labels carry substantive information	When analysts need realistic numeric structure but must remain blind to associations
Flexibility across research settings	Limited by categorical/semantic constraints	Broadly applicable to most data types and study designs

In the `vazul` package, each approach is available at three levels:

- **Vector level**: `mask_labels()` and `scramble_values()` - operate on single vectors
- **Data frame level**: `mask_variables()` and `scramble_variables()` - operate on columns in a data frame
- **Row-wise level**: `mask_variables_rowwise()` and `scramble_variables_rowwise()` - operate within rows across columns

The design of these three levels reflects the common ways researchers interact with data in R. Vector-level functions are the primary building blocks, designed for simple pipelines where a researcher might want to blind a single outcome or a specific grouping factor. These functions are particularly useful when working with **dplyr**'s mutate() or within custom functions where a specific variable needs to be isolated and transformed without affecting the rest of the environment. By providing these atomic operations, **vazul** ensures that the core blinding logic remains accessible even for non-standard data structures.

Moving to the data frame and row-wise levels, the package addresses more complex research designs. Data frame-level functions allow for bulk operations, which are essential for datasets with dozens of variables requiring consistent masking or independent scrambling. Perhaps most importantly, the row-wise operations solve a frequent headache in repeated-measures and longitudinal research. In these designs, the "unit" of analysis is often spread across multiple columns (e.g., independent varibles in factorial designs); the row-wise functions ensure that the relationship within a participant's data is handled correctly—either by scrambling values across those specific time points for each person or by applying a consistent mask that hides the identity of the measurement while preserving the participant-level structure.

All of the functions require a data frame or vector as input and return a modified version with masked or scrambled values. The original data structure, including class attributes and metadata, is carefully preserved to ensure that the blinded data can be passed directly into existing analysis scripts or visualization packages without requiring additional type conversions.

Table 2: Overview of vazul functions for masking and scrambling data

Function	Level	Purpose
`mask_labels()`	Vector	Replace categorical values with anonymous labels
`mask_variables()`	Data frame	Mask multiple columns
`mask_variables_rowwise()`	Row-wise	Consistent masking within rows
`mask_names()`	Variable names	Mask column names
`scramble_values()`	Vector	Randomize value order
`scramble_variables()`	Data frame	Scramble multiple columns
`scramble_variables_rowwise()`	Row-wise	Scramble values within rows

```
R> library(vazul)
R> library(dplyr)
R>
R> set.seed(123)
```

2.2. Included datasets

The `vazul` package includes two research datasets for demonstration and practice. The `marp` dataset contains cross-national survey data on religiosity, while the `williams` dataset contains experimental data from a stereotyping study.

```
R> data(marp)
R> data(williams)
```

2.3. Masking functions

Masking functions replace categorical values with anonymous labels. This is useful when you want to hide the original information, such as treatment conditions or group assignments. Masking variables is useful when there are a limited number of unique values. The `mask_labels()` function takes a character or factor vector and replaces each unique value with a randomly assigned masked label. The function preserves factor structure when the input is a factor. After masking, each unique value receives a unique masked label, the same original value always maps to the same masked label, and the assignment of masked labels to original values is randomized.

```
R> # Create a simple treatment vector
R> treatment <- c("control", "treatment", "control", "treatment", "control")
R>
R> # Mask the labels
R> mask_labels(treatment)

[1] "masked_group_01" "masked_group_02" "masked_group_01" "masked_group_02"
[5] "masked_group_01"
```

It is possible to customize the prefix used for masked labels:

```
R> mask_labels(treatment, prefix = "group_")

[1] "group_01" "group_02" "group_01" "group_02" "group_01"
```

The `mask_variables()` function extends the masking functionality to multiple columns in a data frame simultaneously. It is possible to use tidyelect helpers to select columns.

```
R> marp |>
+     select(rel_1:rel_9, country, denomination) |>
+     mask_variables(c("country", "denomination")) |>
+     head()

      rel_1     rel_2 rel_3 rel_4     rel_5     rel_6     rel_7 rel_8 rel_9
1 0.0000000 0.0000000   0.5     0 0.1666667 0.5000000 0.5000000  0.00  0.50
2 0.8333333 0.7142857   1.0     1 0.5000000 0.5000000 0.3333333  0.50  0.25
3 0.0000000 0.0000000   0.5     0 0.1666667 0.0000000 0.1666667  0.00  0.00
4 0.0000000 0.0000000   0.0     0 0.5000000 0.6666667 0.3333333  0.25  0.00
5 0.0000000 0.0000000   0.0     0 0.1666667 0.0000000 0.3333333  0.00  0.00
6 0.5000000 0.0000000   0.5     1 0.6666667 0.0000000 1.0000000  0.25  0.00
          country          denomination
1 country_group_03                 <NA>
2 country_group_03 denomination_group_10
3 country_group_03                 <NA>
4 country_group_03                 <NA>
5 country_group_03                 <NA>
6 country_group_03 denomination_group_13

R> marp |>
+     select(rel_1:rel_9, country, denomination) |>
+     mask_variables(where(is.character)) |>
+     head()

      rel_1     rel_2 rel_3 rel_4     rel_5     rel_6     rel_7 rel_8 rel_9
1 0.0000000 0.0000000   0.5     0 0.1666667 0.5000000 0.5000000  0.00  0.50
2 0.8333333 0.7142857   1.0     1 0.5000000 0.5000000 0.3333333  0.50  0.25
```

```
3 0.0000000 0.0000000   0.5     0 0.1666667 0.0000000 0.1666667  0.00  0.00
4 0.0000000 0.0000000   0.0     0 0.5000000 0.6666667 0.3333333  0.25  0.00
5 0.0000000 0.0000000   0.0     0 0.1666667 0.0000000 0.3333333  0.00  0.00
6 0.5000000 0.0000000   0.5     1 0.6666667 0.0000000 1.0000000  0.25  0.00
          country         denomination
1 country_group_13                 <NA>
2 country_group_13 denomination_group_05
3 country_group_13                 <NA>
4 country_group_13                 <NA>
5 country_group_13                 <NA>
6 country_group_13 denomination_group_19
```

By default, each column gets its own set of masked labels with the column name as prefix. When `across_variables = TRUE`, all selected columns share the same mapping. This can be useful when the same conditions appear in multiple columns.

```
R> df <- data.frame(
+   pre_condition = c("A", "B", "C", "D"),
+   post_condition = c("B", "D", "D", "C"),
+   id = c(1, 2, 3, 4)
+ )
R>
R> mask_variables(df, c("pre_condition", "post_condition"),
+                               across_variables = TRUE)

    pre_condition  post_condition id
1 masked_group_02 masked_group_04  1
2 masked_group_04 masked_group_01  2
3 masked_group_03 masked_group_01  3
4 masked_group_01 masked_group_03  4
```

The `mask_variables_rowwise()` function applies consistent masking within each row across multiple columns. This is useful in case when of categrical data that is repeated across columns, such as treatment conditions or item responses.

```
R> df <- data.frame(
+   treat_1 = c("control", "treatment_1", "treatment_2"),
+   treat_2 = c("treatment_1", "treatment_2", "control"),
+   treat_3 = c("treatment_2", "control", "treatment_1"),
+   treat_4 = c("treatment_2", "control", "treatment_1"),
+   id = 1:3
+ )
R>
R> mask_variables_rowwise(df, starts_with("treat_"))

          treat_1         treat_2         treat_3         treat_4 id
1 masked_group_03 masked_group_02 masked_group_01 masked_group_01  1
```

```
2 masked_group_02 masked_group_01 masked_group_03 masked_group_03  2
3 masked_group_01 masked_group_03 masked_group_02 masked_group_02  3
```

Masking variable names

The `mask_names()` function allows users to rename variables in a dataset to anonymous, generic labels. This is especially useful in analyses where variable names carry substantive meaning (e.g., questionnaire items, network nodes, test-items), and one wants to prevent analysts from recognizing which item is which during preprocessing or exploratory phases.

The `mask_names()` function takes a data frame and one or more variable sets, which can be specified either with tidyselect expressions (e.g., `starts_with("Q")`) or as character vectors of column names. The `prefix` argument sets the base for the masked names, while `set_id` allows customizing the identifier for each set; if left `NULL`, letters A, B, C... are used automatically.

```
R> williams |>
+     mask_names(starts_with("Impuls"), prefix = "masked_") |>
+     names()

 [1] "subject"              "SexUnres_1"           "SexUnres_2"
 [4] "SexUnres_3"           "SexUnres_4_r"         "SexUnres_5_r"
 [7] "masked_02"            "masked_01"            "Impul_3_r"
[10] "Opport_1"             "Opport_2"             "Opport_3"
[13] "Opport_4"             "Opport_5"             "Opport_6_r"
[16] "InvEdu_1_r"           "InvEdu_2_r"           "InvChild_1"
[19] "InvChild_2_r"         "age"                  "gender"
[22] "ecology"              "duration_in_seconds" "attention_1"
[25] "attention_2"
```

By applying `mask_names()`, the selected variables will be renamed to a pattern such as `variable_set_A_01`, `variable_set_A_02`, etc.—where the letter (e.g., "A") denotes the first block of masked variables. Because the renaming order is randomized, an analyst cannot reliably guess which original variable corresponds to which masked label. Below is a minimal example illustrating how to use `mask_names()` in practice.

```
R> names(williams)

 [1] "subject"              "SexUnres_1"           "SexUnres_2"
 [4] "SexUnres_3"           "SexUnres_4_r"         "SexUnres_5_r"
 [7] "Impuls_1"             "Impuls_2_r"           "Impul_3_r"
[10] "Opport_1"             "Opport_2"             "Opport_3"
[13] "Opport_4"             "Opport_5"             "Opport_6_r"
[16] "InvEdu_1_r"           "InvEdu_2_r"           "InvChild_1"
[19] "InvChild_2_r"         "age"                  "gender"
[22] "ecology"              "duration_in_seconds" "attention_1"
[25] "attention_2"
```

```
R> masked_williams <-
+     williams |>
+     mask_names(
+         starts_with("SexUnres"),
+         starts_with("Impuls"),
+         starts_with("Opport"),
+         starts_with("InvEdu"),
+         starts_with("InvChild"),
+         prefix = "masked_variable_"
+     )
R>
R> names(masked_williams)
```

```
 [1] "subject"            "masked_variable_14" "masked_variable_06"
 [4] "masked_variable_03" "masked_variable_08" "masked_variable_15"
 [7] "masked_variable_17" "masked_variable_01" "Impul_3_r"
[10] "masked_variable_07" "masked_variable_10" "masked_variable_05"
[13] "masked_variable_09" "masked_variable_02" "masked_variable_13"
[16] "masked_variable_16" "masked_variable_11" "masked_variable_04"
[19] "masked_variable_12" "age"                "gender"
[22] "ecology"            "duration_in_seconds" "attention_1"
[25] "attention_2"
```

Once masked, one can proceed with exploratory analyses (e.g. factor analysis, network analysis) on the masked variables, without knowing which original items correspond to which columns. For instance:

```
R> masked_williams |>
+   select(starts_with("masked_variable_")) |>
+   factanal(factors = 5, rotation = "varimax") |>
+   loadings() |>
+   print(cutoff = 0.3, sort = TRUE)
```

```
Loadings:
                   Factor1 Factor2 Factor3 Factor4 Factor5
masked_variable_14  0.606           0.516
masked_variable_06  0.686
masked_variable_03  0.700           0.493
masked_variable_17  0.761
masked_variable_07  0.900
masked_variable_10  0.828
masked_variable_05  0.842
masked_variable_09  0.737
masked_variable_02  0.871
masked_variable_04  0.778
masked_variable_08          0.739   0.303
```

```
masked_variable_15         0.569                    0.394
masked_variable_01         0.864
masked_variable_13         0.687            0.591
masked_variable_16         0.675
masked_variable_11         0.838
masked_variable_12         0.882

               Factor1 Factor2 Factor3 Factor4 Factor5
SS loadings      6.166   4.248   0.813   0.489   0.334
Proportion Var   0.363   0.250   0.048   0.029   0.020
Cumulative Var   0.363   0.613   0.660   0.689   0.709
```

Because the column names are anonymized, any decisions about factor inclusion or rotation are made blind to the substantive meaning of items, reducing the risk of interpretive bias.

In short, `mask_names()` supports naming-based blinding by decoupling analysis from semantic content, while preserving the full data structure needed for legitimate exploratory workflows.

2.4. Scrambling functions

Scrambling functions randomize the order of values while preserving all original data content. This approach maintains the data distribution while breaking the connection between observations and their original values. The `scramble_values()` function randomly reorders the elements of a vector. The vector can contain numeric, character, or factor data. Scrambling is useful when you want to preserve the original values but eliminate any correspondence between observations and their values.

```
R> # Numeric data
R> numbers <- 1:10
R> scramble_values(numbers)

 [1]  8  5  7  3  4  6  2  1 10  9
```

The `scramble_variables()` function extends the scrambling functionality to data frames. It allows scrambling multiple columns simultaneously, with options for independent scrambling, joint scrambling, and within-group scrambling. Columns can be selected using tidyselect helpers.

```
R> df <-
+     williams |>
+     select(subject, age, ecology) |>
+     head()
R>
R> scramble_variables(df, c("age", "ecology"))

# A tibble: 6 x 3
  subject          age ecology
  <chr>          <dbl> <chr>
1 A3OMP4LXV4MIFD    33 Desperate
2 A16X5FB3HAFCKN    30 Desperate
3 A1E9D1OT9VJYDZ    34 Hopeful
4 A16FPOYD7566WI    40 Desperate
5 A11NOTVHWST7Y3    26 Hopeful
6 A3TDR6MXS6UO5Z    35 Desperate
```

By default, columns are scrambled independently of each other. When `together = TRUE`, the selected columns are scrambled as a unit, preserving row-level relationships:

```
R> df <-
+     data.frame(x = 1:6,
+                y = letters[1:6],
+                group = c("A", "A", "A", "B", "B", "B")
+                )
R> df
```

```
  x y group
1 1 a     A
2 2 b     A
3 3 c     A
4 4 d     B
5 5 e     B
6 6 f     B
```

```
R> scramble_variables(df, c("x", "y"), together = TRUE)
```

```
  x y group
1 1 a     A
2 4 d     A
3 6 f     A
4 2 b     B
5 5 e     B
6 3 c     B
```

Scrambling can be done in groups using the `.groups` parameter. This ensures that values are only scrambled within their original groups.

```
R> df |>
+     scramble_variables(c("x", "y"), .groups = "group")
```

```
# A tibble: 6 x 3
      x y     group
  <int> <chr> <chr>
1     3 c     A
```

```
2     1 b     A
3     2 a     A
4     5 d     B
5     4 f     B
6     6 e     B
```

The `scramble_variables_rowwise()` function scrambles values within each row across specified columns. This is useful for scrambling repeated measures or item responses. Within each row, the values are shuffled among the item columns. You can scramble multiple sets of columns independently. The function can use tidyselect helpers.

```
R> df2 <- data.frame(
+   day_1 = c(1, 4, 7),
+   day_2 = c(2, 5, 8),
+   day_3 = c(3, 6, 9),
+   score_a = c(10, 40, 70),
+   score_b = c(20, 50, 80),
+   id = 1:3
+ )
R>
R> scramble_variables_rowwise(df2, starts_with("day_"), c("score_a", "score_b"))

  day_1 day_2 day_3 score_a score_b id
1    20     1     2      10       3  1
2     4    40     5       6      50  2
3    80     8     7      70       9  3
```

3. Summary

The introduction of **vazul** fills a notable gap in the R ecosystem, as no dedicated package currently exists to specifically address the diverse requirements of analysis blinding. While some basic blinding tasks—such as simple vector permutations—can be achieved using base R or **tidyverse** functions like sample() or mutate(), these manual approaches quickly become error-prone as study designs grow in complexity. When dealing with multiple independent variables, hierarchical data structures, or the need for consistent masking across several columns, the risk of "breaking" the data structure or accidentally unblinding the analyst increases significantly. By providing a unified and tested framework, **vazul** abstracts these technical intricacies, allowing researchers to focus on the conceptual rigor of their analysis plans rather than the underlying data manipulation logic.

Furthermore, **vazul** lowers the barrier to entry for researchers who may not have extensive programming experience, effectively making robust bias-control techniques accessible to "newbies" and seasoned analysts alike. Beyond standalone use in R scripts, the package's modular design makes it an ideal candidate for integration into other statistical software platforms. For instance, the inclusion of **vazul** as a backend for the JASP graphical interface could provide a point-and-click solution for analysis blinding, further facilitating the adoption of these best practices across the social and behavioral sciences.

4. Acknowledgements

Tamás Nagy was supported by the University Excellence Fund of Eötvös Loránd University, Budapest, Hungary (ELTE), and the János Bolyai research fellowship of the Hungarian Academy of Sciences. Alexandra Sarafoglou was supported by an 2024 Ammodo Science Award.

Affiliation:

Tamás Nagy
ELTE Eötvös Loránd University
Institute of Psychology,
ELTE Eötvös Loránd University,
Budapest, Hungary
E-mail: nagy.tamas@ppk.elte.hu

Alexandra Sarafoglou
University of Amsterdam
Department of Psychology,
University of Amsterdam,
Amsterdam, The Netherlands
E-mail: a.s.g.sarafoglou@uva.nl

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd