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 Eötvos Loránd University'

Alexandra Sarafoglou
University of Amsterdam'

Abstract

The abstract of the article.

Keywords: keywords, not capitalized, Java.

1. Blinding Data for Analysis with the vazul Package

1.1. 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.

1.2. 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 meaningful 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 true 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 true 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

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

Aspect	Masking	Scrambling
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

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 is preserved.

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)
```

1.3. 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)
```

1.4. 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)

          control         treatment           control         treatment
"masked_group_01" "masked_group_02" "masked_group_01" "masked_group_02"
          control
"masked_group_01"
```

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

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

   control  treatment    control  treatment    control
"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_13
```

```
3 country_group_03                  <NA>
4 country_group_03                  <NA>
5 country_group_03                  <NA>
6 country_group_03 denomination_group_07
```

```
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_01                  <NA>
2 country_group_01 denomination_group_14
3 country_group_01                  <NA>
4 country_group_01                  <NA>
5 country_group_01                  <NA>
6 country_group_01 denomination_group_03
```

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_", set_id = "LH") |>
+     names()

 [1] "subject"             "SexUnres_1"          "SexUnres_2"
 [4] "SexUnres_3"          "SexUnres_4_r"        "SexUnres_5_r"
 [7] "masked_LH_02"        "masked_LH_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")
+     )
R>
R> names(masked_williams)
```

```
 [1] "subject"             "variable_set_A_03"   "variable_set_A_04"
 [4] "variable_set_A_02"   "variable_set_A_01"   "variable_set_A_05"
 [7] "variable_set_B_01"   "variable_set_B_02"   "Impul_3_r"
[10] "variable_set_C_01"   "variable_set_C_03"   "variable_set_C_05"
[13] "variable_set_C_06"   "variable_set_C_04"   "variable_set_C_02"
[16] "variable_set_D_02"   "variable_set_D_01"   "variable_set_E_02"
[19] "variable_set_E_01"   "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("variable_set_")) |>
+   factanal(factors = 5, rotation = "varimax") |>
+   loadings() |>
+   print(cutoff = 0.3, sort = TRUE)
```

```
Loadings:
                  Factor1 Factor2 Factor3 Factor4 Factor5
variable_set_A_03  0.606           0.516
```

```
variable_set_A_04  0.686
variable_set_A_02  0.700           0.493
variable_set_B_01  0.761
variable_set_C_01  0.900
variable_set_C_03  0.828
variable_set_C_05  0.842
variable_set_C_06  0.737
variable_set_C_04  0.871
variable_set_E_02  0.778
variable_set_A_01          0.739   0.303
variable_set_A_05          0.569                   0.394
variable_set_B_02          0.864
variable_set_C_02          0.687           0.591
variable_set_D_02          0.675
variable_set_D_01          0.838
variable_set_E_01          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.

1.5. 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]  7  5  6  9  3  4 10  2  8  1
```

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    40 Desperate
2 A16X5FB3HAFCKN    34 Desperate
3 A1E9D1OT9VJYDZ    33 Desperate
4 A16FPOYD7566WI    30 Hopeful
5 A11NOTVHWST7Y3    35 Hopeful
6 A3TDR6MXS6UO5Z    26 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  <-
+     marp |>
+     select(subject, country, rel_1:rel_3)
R>
R> tail(df)

      subject country     rel_1     rel_2 rel_3
10530   10530      US 0.8333333 0.7142857   1.0
10531   10531      US 0.0000000 0.0000000   0.5
10532   10532      US 0.0000000 0.0000000   0.0
10533   10533      US 0.1666667 1.0000000   1.0
10534   10534      US 0.0000000 0.0000000   0.0
10535   10535      US 0.8333333 0.7142857   1.0

R> scramble_variables(df, starts_with("rel_"), together = TRUE) |>
+     tail()

      subject country     rel_1     rel_2 rel_3
10530   10530      US 0.1666667 0.0000000   0.5
10531   10531      US 0.0000000 0.2857143   0.5
10532   10532      US 0.0000000 0.0000000   0.0
10533   10533      US 0.5000000 1.0000000   1.0
10534   10534      US 0.1666667 0.4285714   0.5
10535   10535      US 0.0000000 0.0000000   0.5
```

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(starts_with("rel_"), .groups = "country")

# A tibble: 10,535 x 5
   subject country   rel_1 rel_2 rel_3
     <int> <chr>     <dbl> <dbl> <int>
 1       1 Australia 0.857   1     197
 2       2 Australia 0.714   1     280
 3       3 Australia 1       0.5    99
 4       4 Australia 0.857   1     320
 5       5 Australia 0.714   0     374
 6       6 Australia 0       1     162
 7       7 Australia 0.143   1     389
 8       8 Australia 1       0.5   175
 9       9 Australia 0       0.5   346
10      10 Australia 0.143   0.5   147
# i 10,525 more rows
```

The `scramble_variables_rowwise()` function scrambles values within each row across specified columns. This is useful for scrambling repeated measures or item responses.

```
R> df <- data.frame(
+   item1 = c(1, 4, 7),
+   item2 = c(2, 5, 8),
+   item3 = c(3, 6, 9),
+   id = 1:3
+ )
R>
R> scramble_variables_rowwise(df, c("item1", "item2", "item3"))

  item1 item2 item3 id
1     3     1     2  1
2     6     4     5  2
3     8     7     9  3
```

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     3     1     2      20      10  1
2     5     4     6      50      40  2
3     7     8     9      70      80  3
```

Affiliation:

Tamás Nagy
'ELTE Eötvos 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'

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