Package: darksoulsarmor (via r-universe)

November 6, 2026

Type Package

Title Explore and Optimize Armor Combinations

Version 2.0.2

Description This package contains exact stats for every armor piece in Dark Souls, at any upgrade level. It also has functions which produce a table of scored armor combinations, and the best value of a stat at each armor weight. This functionality is available in an interactive Shiny application.

License GPL (>= 2)

Encoding UTF-8

LazyData true

URL https://github.com/ricewhitlam/darksoulsarmor

BugReports https://github.com/ricewhitlam/darksoulsarmor/issues

Imports data.table, Rcpp, shiny, shinyWidgets, bslib, DT, shinybusy, plotly, writexl

Depends R (>= 4.3.0)

Suggests testthat (>= 3.0.0), knitr, rmarkdown, readxl, jsonlite

LinkingTo Rcpp

RoxygenNote 7.3.3

Config/testthat/edition 3

VignetteBuilder knitr

Config/pak/sysreqs cmake make libicu-dev libuv1-dev libssl-dev zlib1g-dev

Repository https://ricewhitlam.r-universe.dev

Date/Publication 2026-11-05 21:45:57 UTC

RemoteUrl https://github.com/ricewhitlam/darksoulsarmor

RemoteRef HEAD

RemoteSha 724e47fff920f6ca8803a77a3dcf126cb8428685

Contents

darksoulsarmor-package . . . 2
areas . . . 2
armor.application . . . 3
chest.data.unupgraded . . . 3
classes . . . 4
get.all.armor.combos . . . 4
get.armor.tradeoffs . . . 5
get.optimal.armor.combos . . . 6
get.tradeoff.efficiency . . . 10
hands.data.unupgraded . . . 11
head.data.unupgraded . . . 12
legs.data.unupgraded . . . 12
weapon.data . . . 13

Index **14**

`darksoulsarmor-package` *darksoulsarmor: Explore and Optimize Armor Combinations*

Description

This package contains exact stats for every armor piece in Dark Souls, at any upgrade level. It also has functions which produce a table of scored armor combinations, and the best value of a stat at each armor weight. This functionality is available in an interactive Shiny application.

Author(s)

Maintainer: Package Maintainer <package-maintainer@example.com>

See Also

`get.optimal.armor.combos`, `get.armor.tradeoffs`, `get.tradeoff.efficiency`, `get.all.armor.combos`, `armor.application`

`areas` *Character vector of areas in Dark Souls*

Description

Character vector of areas in Dark Souls

Format

This is not an official or complete listing of areas in the game. Rather, it is the underlying list which is used to indicate completed areas to `get.optimal.armor.combos`.

armor.application *Run an R Shiny application to Produce Optimized Armor Combinations*

Description

Runs a Shiny app for Dark Souls armor optimization. The sidebar sets the constraints and goals, the same settings as `get.optimal.armor.combos`. 'Refresh Armor Data' fills two tabs: Results, a table of the top-scoring combinations, and Trade-offs, a chart of the most of a chosen stat any combination can reach at every armor weight (see `get.armor.tradeoffs`). 'Download Armor Data' saves both, with their settings, as an Excel workbook. Some links are included in this app. To ensure that they work, launch the app in a browser.

Usage

```
armor.application(...)
```

Arguments

`...`	Arguments are passed to `shiny::runApp`.

Value

No value is returned - rather, a Shiny application is run.

Examples

```
## Not run:
armor.application()

## End(Not run)
```

chest.data.unupgraded *Table of all unupgraded chest armor pieces*

Description

Table of all unupgraded chest armor pieces

Format

See `head.data.unupgraded`

classes *Character vector of starting classes in Dark Souls*

Description

Character vector of starting classes in Dark Souls

Format

This is the underlying list which is used to indicate starting class to `get.optimal.armor.combos`.

get.all.armor.combos *Create a* `data.table` *of all possible Dark Souls armor combinations*

Description

Produces a table of every possible armor combination in Dark Souls. This table is large (>10M rows and 17 columns, nearly 2 GB). Only call this function if there is enough available RAM to support this. All relevant metrics are included. All metrics are aggregated as would be expected with perhaps one exception: Durability is aggregated by taking the minimum across all armor pieces. Stats are exact (what the game computes, in 32-bit floating point), not rounded to the one decimal place the game displays. For example, Black Sorcerer Hat's strike defense is 5.1499996, which the game shows as 5.1.

Usage

```
get.all.armor.combos(
 regular.level = c("+0", "+1", "+2", "+3", "+4", "+5", "+6", "+7", "+8", "+9", "+10")[1],
  twinkling.level = c("+0", "+1", "+2", "+3", "+4", "+5")[1]
)
```

Arguments

`regular.level`	A length 1 `character` indicating the upgrade level of armor pieces ascended via regular titanite. Options are `"+0"` thru `"+10"`. Each defense and resistance is computed from the piece's +0 value with the game's own upgrade rates, exactly as the game computes it. Defaults to `"+0"`.
`twinkling.level`	A length 1 `character` indicating the upgrade level of armor pieces ascended via twinkling titanite. Options are `"+0"` thru `"+5"`. Each defense and resistance is computed from the piece's +0 value with the game's own upgrade rates, exactly as the game computes it. Defaults to `"+0"`.

Value

A `data.table` with one row per combination of every piece at the selected upgrade levels; no filters apply. `HEAD`, `CHEST`, `HANDS`, `LEGS`; each defense and resistance, added up; `ARMOR_POISE`, the pieces' poise added up; `DURABILITY`, the lowest of the four; and `ARMOR_WEIGHT`, the armor's weight, added up in 32-bit floating point as the game does.

Examples

```
## Not run:
all.armor.combos <- get.all.armor.combos()

## End(Not run)
```

`get.armor.tradeoffs` *Find the best achievable value of a stat at each armor weight*

Description

For a range of armor-weight limits, finds the combination with the most of one stat - the score, Poise, or a single defense or resistance - that weighs no more than each limit, with every other setting (filters, upgrade levels, rings, minima, ...) as given. This traces the trade-off between armor weight and that stat: e.g. how much more Poise each extra unit of armor weight can buy, or the lightest armor that reaches a Poise breakpoint.

A limit applies to the armor's listed weight (its pieces' weights added up). Every combination must also allow the movement type, checked as the game checks it - with the weapons, rings and the Mask of the Father's equip load bonus, in 32-bit floating point - so near a movement type's line, a combination within the limit can still be ruled out. Among combinations tied on the stat, the best-scoring one is returned.

Usage

```
get.armor.tradeoffs(
  metric = "SCORE",
  weight.step = 0.1,
  min.armor.weight = 0,
  max.armor.weight = NULL,
  ...
)
```

Arguments

`metric`	A length 1 `character`: the stat to maximize. `"SCORE"` (the score built from `weights`, as in `get.optimal.armor.combos`), `"POISE"` (including the Wolf Ring's bonus if `wolf.ring = TRUE`), or one of `"PHYS_DEF"`, `"STRIKE_DEF"`, `"SLASH_DEF"`, `"THRUST_DEF"`, `"MAG_DEF"`, `"FIRE_DEF"`, `"LITNG_DEF"`, `"BLEED_RES"`, `"POIS_RES"`, `"CURSE_RES"`. Defaults to `"SCORE"`.

`weight.step` A length 1 positive `numeric`: the spacing of the armor-weight limits. Limits are the multiples of this size between `min.armor.weight` and `max.armor.weight`, plus those two limits themselves. Armor weights are multiples of 0.1, so steps below 0.1 add points but no detail, and `get.tradeoff.efficiency` is tuned for curves at 0.1. Defaults to `0.1`.

`min.armor.weight` A length 1 non-negative `numeric`: the smallest armor-weight limit, no larger than `max.armor.weight`. Defaults to `0`.

`max.armor.weight` A length 1 non-negative `numeric`, or `NULL`: the largest armor-weight limit. `NULL` (the default) uses the most armor weight the other settings allow - the movement type's line, with the Mask of the Father's bonus, less what the `weapons` weigh, rounded down to 0.1 - or, with `movement = "Poop"`, the heaviest possible armor.

`...` Any other arguments of `get.optimal.armor.combos` (except `max.table.size`), with the same meaning and defaults.

Value

A `list` holding (1) the arguments that defined the curve and (2) a `data.table` with one row per armor-weight limit (`ARMOR_WEIGHT_LIMIT`): the best achievable value of the metric (`BEST_VALUE`), and the combination achieving it - its `SCORE`, `SCORE_QUALITY`, `ARMOR_WEIGHT`, `TOTAL_POISE`, and pieces. Limits at which no combination satisfies the other settings have `NA` values. Values are exact, as in `get.optimal.armor.combos`.

Examples

```
poise.curve <- get.armor.tradeoffs(metric = "POISE", endurance.level = 40, movement = "Mid", weapons = c(right.1 =
```

`get.optimal.armor.combos`

Create a `data.table` *of optimized Dark Souls armor combinations*

Description

Produces a table of optimized armor combinations in Dark Souls. All relevant metrics are included, along with two score columns: `SCORE`, an unbounded standardized score (the one the table is sorted on - see `vignette("scoring")`), and `SCORE_QUALITY`, that score's exact rarity among every possible combination at the selected upgrade levels (regardless of the other filters) - e.g. `"Top 1 in 40"` when 1 in every 40 such combinations scores at least as well, or `"Bottom 1 in 40"` when 1 in every 40 scores at most as well. Combinations with equal scores are ordered lighter first, then more poise, then more durability. Stats are exact (what the game computes, in 32-bit floating point), not rounded to the one decimal place the game displays. For example, Black Sorcerer Hat's strike defense is 5.1499996, which the game shows as 5.1. The table can be tailored to satisfy various constraints.

Usage

```
get.optimal.armor.combos(
  max.table.size = 1000,
  starting.class = classes[1],
  areas.completed = areas,
  upgrade.types = c("Regular", "Twinkling", "None"),
  head.filter = head.data.unupgraded$ARMOR,
  chest.filter = chest.data.unupgraded$ARMOR,
  hands.filter = hands.data.unupgraded$ARMOR,
  legs.filter = legs.data.unupgraded$ARMOR,
 regular.level = c("+0", "+1", "+2", "+3", "+4", "+5", "+6", "+7", "+8", "+9", "+10")[1],
  twinkling.level = c("+0", "+1", "+2", "+3", "+4", "+5")[1],
  movement = c("Light", "Mid", "Fat", "Poop")[1],
 weapons = c(left.1 = "None", right.1 = "None", left.2 = "None", right.2 = "None"),
  endurance.level = 10,
  havel.ring = FALSE,
  favor.ring = FALSE,
  wolf.ring = FALSE,
  minima = c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),
  weights,
  stat.scale = "sd",
  weights.mode = "individual"
)
```

Arguments

`max.table.size`	A length 1 `numeric` indicating how large the produced table should be. Defaults to `1000`. Passed values will be clamped between 1 and 100,000,000 and cast to an integer.
`starting.class`	A length 1 `character` indicating the character's starting class, whose starting armor set is treated as available regardless of `areas.completed`. Defaults to `"Warrior"`. The vector of available classes is stored as `classes`.
`areas.completed`	A `character` vector indicating which areas have been completed. Defaults to all areas being complete. The vector of available areas is stored as `areas`.
`upgrade.types`	A `character` vector indicating which upgrade types to consider. There are three types available: `"Regular"` (armor pieces that upgrade with regular titanite), `"Twinkling"` (armor pieces that upgrade with twinkling titanite), and `"None"` (armor pieces that cannot be upgraded). Defaults to all types i.e. `c("Regular", "Twinkling", "None")`.
`head.filter`	A `character` vector indicating which head armor pieces should be included. Defaults to all pieces. A vector of available pieces can be accessed via `head.data.unupgraded$ARMOR`. If an empty vector is passed, it is assumed that no head armor will be worn.
`chest.filter`	Same as `head.filter` but for chest armor pieces. A vector of available pieces can be accessed via `chest.data.unupgraded$ARMOR`.
`hands.filter`	Same as `head.filter` but for hand armor pieces. A vector of available pieces can be accessed via `hands.data.unupgraded$ARMOR`.

`legs.filter` Same as `head.filter` but for leg armor pieces. A vector of available pieces can be accessed via `legs.data.unupgraded$ARMOR`.

`regular.level` A length 1 `character` indicating the upgrade level of armor pieces ascended via regular titanite. Options are `"+0"` thru `"+10"`. Each defense and resistance is computed from the piece's +0 value with the game's own upgrade rates, exactly as the game computes it. Defaults to `"+0"`.

`twinkling.level` A length 1 `character` indicating the upgrade level of armor pieces ascended via twinkling titanite. Options are `"+0"` thru `"+5"`. Each defense and resistance is computed from the piece's +0 value with the game's own upgrade rates, exactly as the game computes it. Defaults to `"+0"`.

`movement` A length 1 `character`: the heaviest movement type to allow, by the share of max equip load carried. Options are `"Light"` (weight at or below 25% of max equip load: light roll), `"Mid"` (at or below 50%: mid roll), `"Fat"` (at or below 100%: fat roll), and `"Poop"` (no equip-load limit - over 100%, the character can't roll and walks slowly). Defaults to `"Light"`.

`weapons` A `character` vector of the weapons equipped in the four weapon slots, by name (see `weapon.data`), or `"None"` for an empty slot. Either named by slot, in any order, leaving out empty slots - `left.1`, `right.1`, `left.2` and `right.2` for each hand's first and second slot, e.g. `c(left.1 = "Grass Crest Shield", right.1 = "Longsword")` - or an unnamed length 4 vector in exactly that order. Rings and ammunition weigh nothing, so these are everything the character carries besides armor. The game adds the four slots' weights in that order, one at a time in 32-bit floating point, then the armor's, which makes the total depend on which slot holds which weapon (with two or more talismans), so they're taken per slot rather than as a single weight. Only their weight is used. Defaults to every slot empty, as does an empty vector (`character(0)`).

`endurance.level` A length 1 `numeric` indicating the level of the character in the Endurance stat. This stat increases maximum equip load. Defaults to `10`. Passed values are clamped between 0 and 99 and rounded to an integer.

`havel.ring` A length 1 `logical` indicating whether Havel's Ring is equipped. This ring increases maximum equip load by 50%. Defaults to `FALSE`.

`favor.ring` A length 1 `logical` indicating whether the Ring of Favor and Protection is equipped. This ring increases maximum equip load by 20%. Defaults to `FALSE`.

`wolf.ring` A length 1 `logical` indicating whether the Wolf Ring is equipped. This ring increases Poise by 40. Defaults to `FALSE`.

`minima` A `numeric` indicating minimum allowable values for metrics. Either a named vector with any of the names PHYS_DEF, STRIKE_DEF, SLASH_DEF, THRUST_DEF, MAG_DEF, FIRE_DEF, LITNG_DEF, POISE, BLEED_RES, POIS_RES, CURSE_RES, DURABILITY, in any order (omitted metrics have no minimum, i.e. 0) - e.g. `c(POISE = 30, DURABILITY = 200)` - or an unnamed length 12 vector in exactly that order. Defaults to `c(0,0,0,0,0,0,0,0,0,0,0,0)`. Passed values are clamped between 0 and 999.

`weights` A `numeric` indicating weights for the scored metrics. With `weights.mode = "individual"`, either a named vector with any of the names PHYS_DEF,

STRIKE_DEF, SLASH_DEF, THRUST_DEF, MAG_DEF, FIRE_DEF, LITNG_DEF, BLEED_RES, POIS_RES, CURSE_RES, in any order (e.g. `c(PHYS_DEF = 2, MAG_DEF = 1)`), or an unnamed length 10 vector in exactly that order. With `weights.mode = "grouped"`, either a named vector with any of the names PHYSICAL_DEFENSES, ELEMENTAL_DEFENSES, RESISTANCES, or an unnamed length 3 vector in that order. With a named vector, omitted names get weight 0: they don't count toward the score at all. Required: there is no default. These weights are used in the calculation of a score. This score is then optimized across all possible armor combinations. Increasing the weight on a metric increases its importance to the final score. The weights do not need to sum to 1, but should all be nonnegative with positive sum.

`stat.scale` A length 1 `character` indicating what one unit of each stat counts as before it is weighted. `"sd"` (the default) indicates the stat's standard deviation across every armor combination, `"max"` indicates the stat's highest value across every armor combination, and `"raw"` indicates one unit of the stat with no adjustment.

`weights.mode` A length 1 `character` indicating what the weights apply to. `"individual"` (the default) indicates one weight per scored stat. `"grouped"` indicates one weight per group of stats: PHYSICAL_DEFENSES (PHYS_DEF, STRIKE_DEF, SLASH_DEF, THRUST_DEF), ELEMENTAL_DEFENSES (MAG_DEF, FIRE_DEF, LITNG_DEF) and RESISTANCES (BLEED_RES, POIS_RES, CURSE_RES). Each group counts as the average of its stats, and `stat.scale` applies to that average.

Value

A `list` holding (1) `args`, the arguments as validated and used: clamped and rounded where noted above, duplicates removed, an empty piece filter as its slot's "No ..." piece, and `weights` divided by their sum; and (2) `data`, a `data.table` with one row per combination, best first:

- `SCORE`: the score, with mean 0 and standard deviation 1 across every possible combination (see `vignette("scoring")`).
- `SCORE_QUALITY`: its exact rarity among every combination at the selected upgrade levels, e.g. `"Top 1 in 40"`.
- `HEAD`, `CHEST`, `HANDS`, `LEGS`: the pieces (`"No Head"` etc. for an empty slot).
- `PHYS_DEF` ... `CURSE_RES`: the set's totals at the selected upgrade levels.
- `DURABILITY`: the lowest durability among the four pieces.
- `ARMOR_POISE`: the pieces' poise added up. `TOTAL_POISE`: the same plus the Wolf Ring's 40 when it's equipped.
- `POISE_TIMER`: the poise recovery timer, in seconds: 5, less 10% for each piece with any poise (4.5 with one, about 3.28 with four).
- `ARMOR_WEIGHT`: the armor's weight, added up in 32-bit floating point as the game does.
- `TOTAL_WEIGHT`: everything carried, weapons then armor, added up as the game does.
- `EQUIP_LOAD`: the maximum equip load, with the rings and, when it's worn, the Mask of the Father's bonus.
- `PCT_LOAD`: `TOTAL_WEIGHT` divided by `EQUIP_LOAD` (0.25 is the Light line, 0.5 Mid, 1 Fat).

Examples

```
optimal.armor.combos <- get.optimal.armor.combos(endurance.level = 40, weapons = c(left.1 = "Grass Crest Shield",

## At least 30 poise, scored only on physical and magic defense (physical counting double)
poise.combos <- get.optimal.armor.combos(endurance.level = 40, movement = "Mid", minima = c(POISE = 30), weights =

## Physical defenses, elemental defenses and resistances weighted equally, each against its best possible total
grouped.combos <- get.optimal.armor.combos(endurance.level = 40, movement = "Mid", weights = c(PHYSICAL_DEFENSES =
```

`get.tradeoff.efficiency` *Describe where extra armor weight pays off*

Description

Describes a trade-off curve (from `get.armor.tradeoffs`) as a sequence of pieces, so that each step of the curve belongs to exactly one of them:

- **jumps**: single steps where a little more weight buys a lot;
- **lines**: regions with a steady rate, compared with the curve's average rate (its total gain divided by its total weight) - above it, extra weight buys more than average;
- **flats**: regions where extra weight buys nothing;

with a **boundary point** wherever two regions (lines or flats) meet.

Points with no set (`NA`) are left out. The range is the curve's highest value less its lowest, and the width its last weight less its first. A step smaller than a millionth of the range counts as no change: such steps are 32-bit rounding between near-identical sets.

The rules were tuned on curves every 0.1 weight, `get.armor.tradeoffs`' default; on a coarser curve each step merges several, so more of them read as jumps.

Jumps. A step is a jump if it rises at least a share of the range, and is worth at least an amount of weight at the rate around it: its rise divided by the rate of the rest of the gain within 4.5 weight either side of it, leaving out any jumps already chosen. So a big step in a steep stretch, whose neighbours rise as fast, isn't a jump; the stretch is a steep line. Jumps are chosen biggest first, up to a maximum; equal steps are taken together or not at all (a group that would go past the maximum is left out, along with everything smaller).

Flats. A run of steps with no gain at least a share of the width long, longest first, up to a maximum, equal lengths taken together or not at all.

Lines. Each stretch between the jumps and flats is fitted with the best 1, 2, 3 or 4 least-squares lines, each at least 2.5 weight wide, adding a line only if it brings the error (root mean square) down to at most 45% of what it was. A line's rate is its gain from the set before it to its last set, divided by the weight between them. A line with no gain is a flat.

Boundary points. Between two lines, the break moves to the biggest step within 1.25 weight of where the fit put it, which becomes the last step of the earlier line; from a line into a flat, it's the last

step before the flat; from a flat into a line, the first step after it. A single step between two flats is itself the boundary point between them.

Promoted jumps. While fewer jumps than the maximum are in use, a boundary point's step becomes a jump if it's among the curve's largest steps (as many as the maximum, equal ones together), rises at least half the share of the range a jump otherwise needs, and is worth as much weight; biggest first, the curve described again around each.

Usage

```
get.tradeoff.efficiency(curve, jumps = "some", flats = "some")
```

Arguments

curve	A curve from get.armor.tradeoffs (or its data): a table with ARMOR_WEIGHT_LIMIT and BEST_VALUE. Any other columns are ignored.
jumps	How many jumps to look for: "few" (at most 3, each rising at least 10% of the range and worth at least 1.6 weight), "some" (at most 6, 6%, 1.23 weight) or "many" (at most 8, 3%, 0.85 weight). Defaults to "some".
flats	How many flats to look for: "few" (at most 2, each at least 15% of the width), "some" (at most 3, 10%) or "many" (at most 5, 5%). Defaults to "some".

Value

A list holding (1) average.slope, the curve's total gain divided by its total weight, and (2) a data.table with one row per piece, in weight order: its TYPE ("jump", "line", "flat" or "boundary"); the weights it runs between (FROM, TO: from the set before it to its last set, so that jumps, lines and flats join end to end, a jump or boundary point being one step) and the curve's values there (START_VALUE, END_VALUE), with their difference (GAIN); for lines and flats, the rate as a multiple of the average (RATIO_TO_AVERAGE, NA when the average is 0) and whether it's above average (ABOVE_AVERAGE); and for boundary points, the rates before and after (RATIO_BEFORE, RATIO_AFTER). A boundary point's step also belongs to a line beside it, except for one between two flats.

Examples

```
score.curve <- get.armor.tradeoffs(weight.step = 0.1, endurance.level = 40, movement = "Fat", weights = c(16, 16, 1
efficiency <- get.tradeoff.efficiency(score.curve)
fewer.jumps <- get.tradeoff.efficiency(score.curve, jumps = "few")
```

---

hands.data.unupgraded *Table of all unupgraded hand armor pieces*

Description

Table of all unupgraded hand armor pieces

Format

See head.data.unupgraded

`head.data.unupgraded` *Table of all unupgraded head armor pieces*

Description

Table of all unupgraded head armor pieces

Format

ARMOR Name of armor piece

UPGRADE_TYPE One of `"Regular"` (upgrades with regular titanite), `"Twinkling"` (upgrades with twinkling titanite), or `"None"` (cannot be upgraded).

STARTING_CLASS If relevant, the starting class associated with the armor piece. Otherwise, `"N/A"`.

AREA_MATCH_TYPE, AREA_LIST Together, encode which areas make the piece available. Each holds one or more `"|"`-separated clauses combined with OR. A clause in `AREA_MATCH_TYPE` is `"ALWAYS"` (always available; its `AREA_LIST` clause is empty), `"ANY"` (available once at least one of its `";"`-separated `AREA_LIST` areas is completed), or `"ALL"` (available once every one of its areas is completed).

LINK A link to the armor piece on darksouls.wikidot.com.

PHYS_DEF, STRIKE_DEF, SLASH_DEF, THRUST_DEF, MAG_DEF, FIRE_DEF, LITNG_DEF, POISE, BLEED_ The value of the relevant metric at +0. Strike, slash and thrust defense are the game's exact 32-bit values (e.g. 5.1499996 where the game shows 5.1); the other stats are exact as listed.

STAM_MOD The additive modifier to the base stamina regeneration rate of 45 per second.

SOUND_MOD The multiplicative modifier to base character loudness.

`legs.data.unupgraded` *Table of all unupgraded leg armor pieces*

Description

Table of all unupgraded leg armor pieces

Format

See head.data.unupgraded

`weapon.data` *Table of every weapon, shield, bow and catalyst that can fill a weapon slot*

Description

Table of every weapon, shield, bow and catalyst that can fill a weapon slot

Format

WEAPON Name of the weapon.

WEIGHT Its weight, which infusions and upgrades don't change.

Weight is the only property the package uses: a weapon's other effects (e.g. the Gargoyle Tail Axe raising unarmored poison and bleed resistance) change the character, not its armor, so they are outside the armor stats the package computes.

Index

∗ **data**
areas, 2
chest.data.unupgraded, 3
classes, 4
hands.data.unupgraded, 11
head.data.unupgraded, 12
legs.data.unupgraded, 12
weapon.data, 13

areas, 2
armor.application, *2*, 3

chest.data.unupgraded, 3
classes, 4

darksoulsarmor
(darksoulsarmor-package), 2
darksoulsarmor-package, 2

get.all.armor.combos, *2*, 4
get.armor.tradeoffs, *2, 3*, 5, *10, 11*
get.optimal.armor.combos, *2–5*, 6, *6*
get.tradeoff.efficiency, *2*, *6*, 10

hands.data.unupgraded, 11
head.data.unupgraded, *3*, 12, *12*

legs.data.unupgraded, 12

weapon.data, *8*, 13