

---

# rmlint documentation

*Release*

May 01, 2018



---

## Contents

---

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	Dependencies . . . . .	3
1.2	Compilation . . . . .	4
<b>2</b>	<b>Tutorial</b>	<b>5</b>
2.1	Beginner Examples . . . . .	5
2.2	Filtering . . . . .	6
2.3	Outputs . . . . .	7
2.4	Paranoia . . . . .	10
2.5	Original detection . . . . .	11
2.6	Finding duplicate directories . . . . .	12
2.7	Misc options . . . . .	13
<b>3</b>	<b>Frequently Asked Questions</b>	<b>15</b>
3.1	<code>rmlint</code> finds more/less dupes than tool <b>X</b> ! . . . . .	15
3.2	Can you implement feature <b>X</b> ? . . . . .	15
<b>4</b>	<b>Informative reference</b>	<b>17</b>
4.1	Developer's Guide . . . . .	17
4.2	Translating <code>rmlint</code> . . . . .	20
4.3	<code>rmlint</code> . . . . .	21
<b>5</b>	<b>Authors</b>	<b>29</b>
<b>6</b>	<b>License</b>	<b>31</b>
<b>7</b>	<b>Donations</b>	<b>33</b>



---

# [rm]lint

**rmlint** finds space waste and other broken things on your filesystem and offers to remove it. It is able to find:

- Duplicate files & directories.
- Nonstripped Binaries
- Broken symlinks.
- Empty files.
- Recursive empty directories.
- Files with broken user or group id.

**Key Features:**

- Extremely fast.
- Exchangeable hashing algorithm.
- Numerous output formats.
- Easy commandline interface.
- Possibility to update files with newer mtime.
- Many options for originaldetection.

Although **rmlint** is easy to use, you might want to read these chapters first. They show you the basic principles and most of the advanced options:



Many major Linux distribution might already package `rmlint` – but watch out for the version. This manual describes the rewrite of `rmlint` (i.e. version  $\geq 2$ ). Old versions before this might contain bugs, have design flaws or might eat your hamster. We recommend using the newest version.

If there is no package yet or you want to try a development version, you gonna need to compile `rmlint` from source.

## 1.1 Dependencies

### 1.1.1 Hard dependencies:

- **glib**  $\geq 2.32$  (general C Utility Library)
- **libblkid** (detecting mountpoints)
- **libelf** (nonstripped binary detection)

### 1.1.2 Build dependencies:

- **git** (version control)
- **scons** (build system)
- **sphinx** $\geq 3.0$  (manpage/documentation generation)

Here's a list of readily prepared commands for known distributions:

- **Fedora:**

```
$ yum -y install git scons python3-sphinx gettext
$ yum -y install glib2-devel libblkid-devel elfutils-libelf-devel
```

- **ArchLinux:**

```
$ pacman -S git scons python-sphinx
$ pacman -S glib2 libutil-linux elfutils
```

- **Ubuntu:**

```
$ apt-get install git scons python3-sphinx python3-nose gettext
$ apt-get install libelf-dev libglib2.0-dev libblkid-dev
```

Send us a note if you want to see your distribution here.

## 1.2 Compilation

Compilation consists of getting the source and translating it into a usable binary:

```
$ git clone -b develop https://github.com/sahib/rmlint.git
$ cd rmlint/
$ scons config          # Look what features scons would compile
$ scons DEBUG=1 -j4     # For releases you can omit DEBUG=1
$ sudo scons DEBUG=1 --prefix=/usr install
```

Done!

You should be now able to see the manpage with `rmlint -h` or `man 1 rmlint`.



Welcome to the Tutorial of `rmlint`.

We use a few terms that might not be obvious to you at first, so we gonna explain them to you here.

**Duplicate** A file that has the same hash as another file.

**Original** In a group of *duplicates*, one file is said to be the original file, from which the copies where created. This might or might not be true, but is an helpful assumption when deleting files.

## 2.1 Beginner Examples

Let's just dive in into some examples:

```
$ rmlint
```

This simply scans your current working directory for lint and reports them in your terminal. Note that **nothing will be removed** (even if it prints `rm`).

Despite it's name, `rmlint` just finds suspicious files, but never modifies the filesystem itself<sup>0</sup>. Instead it gives you detailed reports in different formats to get rid of them yourself. These reports are called *outputs*. By default a shellscript will be written to `rmlint.sh` that contains readily prepared shell commands to remove duplicates and other finds,

So for the above example, the full process would be:

```
`-FF` together with this option. rmlint
# (wait for rmlint to finish running)
$ gedit rmlint.sh
# (or any editor you prefer... review the content of rmlint.sh to
# check what it plans to delete; make any edits as necessary)
$ ./rmlint.sh
# (the rmlint.sh script will ask for confirmation, then delete the
# appropriate lint, then delete itself)
```

---

<sup>0</sup> You could say it should be named `findlint`.

---

## 2.2 Filtering

What if we do not want to check all files as dupes? `rmlint` has a good repertoire of options to select only certain files. We won't cover all options, but the useful ones. If those options do not suffice, you can always use external tools to feed `rmlint`'s `stdin`:

```
$ find pics/ -iname '*.png' | rmlint -
```

### 2.2.1 Limit files by size using `--size`

```
# only check files between 20 MB and 1 Gigabyte:
$ rmlint --size 20M-1G
# short form (-s) works just as well:
$ rmlint -s 20M-1G
# only check files bigger than 4 kB:
$ rmlint -s 4K
# only check files smaller than 1234 bytes:
$ rmlint -s 0-1234
```

Valid units include:

K,M,G,T,P for powers of 1000

KB, MB, GB etc for powers of 1024

If no units are given, `rmlint` assumes bytes.

### 2.2.2 Limit files by their basename

By default, `rmlint` compares file contents, regardless of file name. So if `afile.jpg` has the same content as `bfile.txt` (which is unlikely!), then `rmlint` will find and report this as a duplicate. You can speed things up a little bit by telling `rmlint` not to try to match files unless they have the same or similar file names. The three options here are:

```
-b (--match-basename)
-e (--match-extension)
-i (--match-without-extension).
```

Examples:

```
# Find all duplicate files with the same basename:
$ rmlint -b some_dir/
ls some_dir/one/hello.c
rm some_dir/two/hello.c
# Find all duplicate files that have the same extension:
$ rmlint -e some_dir/
ls some_dir/hello.c
```

```
rm some_dir/hello_copy.c
# Find all duplicate files that have the same basename:
# minus the extension
$ rmlint -e some_dir/
ls some_dir/hello.c
rm some_dir/hello.bak
```

### 2.2.3 Limit files by their modification time

This is an useful feature if you want to investigate only files newer than a certain date or if you want to progressively update the results, i.e. when you run `rmlint` in a script that watches a directory for duplicates.

The most obvious way is using `-N` (`--newer-than=<timestamp>`):

```
# Use a Unix-UTC Timestamp (seconds since epoch)
$ rmlint -N 1414755960%

# Find all files newer than file.png
$ rmlint -N $(stat --print %Y file.png)

# Alternatively use a ISO8601 formatted Timestamp
$ rmlint -N 2014-09-08T00:12:32+0200
```

If you are checking a large directory tree for duplicates, you can get a substantial speedup by creating a timestamp file each time you run `rmlint`. To do this, use command line options: `-n` (`--newer-than-stamp`) and `-O stamp:stamp.file` (we'll come to outputs in a minute): Here's an example for incrementally scanning your home folder:

```
# First run of rmlint:
$ rmlint /home/foobar -O stamp:/home/foobar/.rmlint.stamp
ls /home/foobar/a.file
rm /home/foobar/b.file

# Second run, no changes:
$ rmlint /home/foobar -n /home/foobar/.rmlint.stamp
<nothing>

# Second run, new file copied:
$ cp /home/foobar/a.file /home/foobar/c.file
$ rmlint /home/foobar -n /home/foobar/.rmlint.stamp
ls /home/foobar/a.file
rm /home/foobar/b.file
rm /home/foobar/c.file
```

Note that `-n` updates the timestamp file each time it is run.

## 2.3 Outputs

`rmlint` is capable to create it's reports in several output-formats. Actually if you run it with the default options you already see two of those formatters: Namely `pretty` and `summary`.

Formatters can be added via the `-O` (`--add-output`) switch. The `-o` (`--output`) instead clears all defaults first and does the same as `-O` afterwards.

Here's an example:

```
$ rmlint -o json:stderr
```

Here you would get this output printed on `stderr`:

```
[{
  "description": "rmlint json-dump of lint files",
  "cwd": "/home/user/",
  "args": "rmlint -o json:stderr"
},
{
  "type": "duplicate_file",
  "path": "/home/user/test/b/one",
  "size": 2,
  "inode": 2492950,
  "disk_id": 64771,
  "is_original": true,
  "mtime": 1414587002
},
... snip ...
{
  "aborted": false,
  "total_files": 145,
  "ignored_files": 9,
  "ignored_folders": 4,
  "duplicates": 11,
  "duplicate_sets": 2,
  "total_lint_size": 38
}]
```

You probably noticed the colon in the commandline above. Everything before it is the name of the output-format, everything behind is the path where the output should land. Instead of an path you can also use `stdout` and `stderr`, as we did above.

Some formatters might be configured to generate subtly different output using the `-c` (`--config`) command. Here's the list of currently available formatters and their config options:

**json** Outputs all finds as a json document. The document is a list of dictionaries, where the first and last element is the header and the footer respectively, everything between are data-dictionaries. This format was chosen to allow application to parse the output in realtime while `rmlint` is still running.

The header contains information about the program invocation, while the footer contains statistics about the program-run. Every data element has a type which identifies it's lint type (you can lookup all types [here](#)).

**Config values:**

- `use_header=[true|false]`: Print the header with metadata.
- `use_footer=[true|false]`: Print the footer with statistics.

**sh** Outputs a shell script that has default commands for all lint types. The script can be executed (it is already `chmod +x`'d by `rmlint`). By default it will ask you if you really want to proceed. If you do not want that you can pass the `-d`. Additionally it will delete itself after it ran, except you passed the `-x` switch.

It is enabled by default and writes to `rmlint.sh`.

Example output:

```
$ rmlint -o sh:stdout
#!/bin/sh
# This file was autowritten by rmlint
# rmlint was executed from: /home/user/
# You command line was: ./rmlint -o sh:rmlint.sh

# ... snip ...

echo '/home/user/test/b/one' # original
rm -f '/home/user/test/b/file' # duplicate
rm -f '/home/user/test/a/two' # duplicate
rm -f '/home/user/test/a/file' # duplicate

if [ -z $DO_REMOVE ]
then
    rm -f 'rmlint.sh';
fi
```

**Config values:**

- `use_ln=[true|false]`: Replace duplicate files with symbolic links (if on different device as original) or with hardlinks (if on same device as original).
- `symlinks_only=[true|false]`: Always use symbolic links with `use_ln`, never hardlinks.

**Example:**

```
$ rmlint -o sh:stdout -o sh:rmlint.sh -c sh:use_ln=true -c sh:symlinks_
↪only=true
...
echo '/home/user/test/b/one' # original
ln -s '/home/user/test/b/file' # duplicate
$ ./rmlint.sh -d
/home/user/test/b/one
```

**py** Outputs a python script and a JSON document, just like the **json** formatter. The JSON document is written to `.rmlint.json`, executing the script will make it read from there. This formatter is mostly intended for complex usecases where the lint needs special handling. Therefore the python script can be modified to do things standard `rmlint` is not able to do easily. You have the full power of the Python language for your task, use it!

**Example:**

```
$ rmlint -o py:remover.py
$ ./remover.py --dry-run # Needs Python3
Deleting twins of /home/user/sub2/a
Handling (duplicate_file): /home/user/sub1/a
Handling (duplicate_file): /home/user/a

Deleting twins of /home/user/sub2/b
Handling (duplicate_file): /home/user/sub1/b
```

**csv** Outputs a csv formatted dump of all lint files. It looks like this:

```
$ rmlint -o csv -D
type,path,size,checksum
emptydir,"/home/user/tree2/b",0,00000000000000000000000000000000
duplicate_dir,"/home/user/test/b",4,f8772f6fda08bbc826543334663d6f13
duplicate_dir,"/home/user/test/a",4,f8772f6fda08bbc826543334663d6f13
```

```
duplicate_dir, "/home/user/tree/b", 8, 62202a79add28a72209b41b6c8f43400
duplicate_dir, "/home/user/tree/a", 8, 62202a79add28a72209b41b6c8f43400
duplicate_dir, "/home/user/tree2/a", 4, 311095bc5669453990cd205b647a1a00
```

**Config values:**

- `use_header=[true|false]`: Print the column name headers.

**stamp** Outputs a timestamp of the time `rmlint` was run.

**Config values:**

- `iso8601=[true|false]`: Write an ISO8601 formatted timestamps or seconds since epoch?

**pretty** Prettyprints the finds in a colorful output supposed to be printed on `stdout` or `stderr`. This is what you see by default.

**summary** Sums up the run in a few lines with some statistics. This enabled by default too.

**progressbar** Prints a progressbar during the run of `rmlint`. This is recommended for large runs where the `pretty` formatter would print thousands of lines.

**Config values:**

- `update_interval=number`: Number of files to wait between updates. Higher values use less resources.

## 2.4 Paranoia

Let's face it, why should you trust `rmlint`?

Technically it only computes a hash of your file which might, by it's nature, collide with the hash of a totally different file. If we assume a *perfect* hash function (i.e. one that distributes it's hash values perfectly even over all possible values), the probability of having a hash-collision is  $\frac{1}{2^{128}}$  for the default 128-bit hash. Of course hash functions are not totally random, so the collision probability is slightly higher.

If you're wary you might want to make a bit more paranoid than it's default. By default the `spooky` hash algorithm is used, which we consider a good tradeoff of speed and accuracy. `rmlint`'s paranoia level can be easily inc/decreased using the `-p` (`--paranoid`) / `-P` (`--less-paranoid`) option (which might be given up to three times each).

Here's what they do in detail:

- `-p` is equivalent to `--algorithm=bastard`
- `-pp` is equivalent to `--algorithm=sha512`
- `-ppp` is equivalent to `--algorithm=paranoid`

As you see, it just enables a certain hash algorithm. `--algorithm` changes the hash algorithm to something more secure. `bastard` is a 256bit hash that consists of two 128bit subhashes (`murmur3` and `city` if you're curious). One level up the well-known `sha512` (with 512bits obviously) is used. Another level up, no hash function is used. Instead, files are compared byte-by-byte (which guarantees collision free output).

There is a bunch of other hash functions you can lookup in the manpage. We recommend never to use the `-P` option.

---

**Note:** Even with the default options, the probability of a false positive doesn't really start to get significant until you have around 1,000,000,000,000,000,000 files all of the same file size. Bugs in `rmlint` are sadly (or happily?) more likely than hash collisions.

---

## 2.5 Original detection

As mentioned before, `rmlint` divides a group of dupes in one original and clones of that one. While the chosen original might not be the one that was there first, it is a good thing to keep one file of a group to prevent dataloss.

The way `rmlint` chooses the original can be driven by the `-S` (`--sortcriteria`) option.

Here's an example:

```
# Normal run:
$ rmlint
ls c
rm a
rm b

# Use alphabetically first one as original
$ rmlint -S a
ls a
rm b
rm c
```

Alphabetically first makes sense in the case of backup files, ie **a.txt.bak** comes after **a.txt**.

Here's a table of letters you can supply to the `-S` option:

<b>m</b>	keep lowest mtime (oldest)	<b>M</b>	keep highest mtime (newest)
<b>a</b>	keep first alphabetically	<b>A</b>	keep last alphabetically
<b>p</b>	keep first named path	<b>P</b>	keep last named path

The default setting is `-S m` – which takes the oldest file, determined by it's modification time. Multiple sort criteria can be specified, eg `-S mpa` will sort first by mtime, then (if tied), based on which path you specified first in the `rmlint` command, then finally based on alphabetical order of file name. Note that "original directory" criteria (see below) take precedence over the `-S` options.

### 2.5.1 Flagging original directories

But what if you know better than `rmlint`? What if your originals are in some specific path, while you know that the files in it are copied over and over? In this case you can flag directories on the commandline to be original, by using a special separator (`//`) between the duplicate and original paths. Every path after the `//` separator is considered to be "tagged" and will be treated as an original where possible. Tagging always takes precedence over the `-S` options above.

```
$ rmlint a // b
ls b/file
rm a/file
```

If there are more than one tagged files in a duplicate group then the highest ranked (per `-S` options) will be kept. In order to never delete any tagged files, there is the `-k` (`--keep-all-tagged`) option. A slightly more esoteric option is `-m` (`--must-match-tagged`), which only looks for duplicates where there is an original in a tagged path.

Here's a real world example using these features: I have an portable backup drive with some old backups on it. I have just backed up my home folder to a new backup drive. I want to reformat the old backup drive and use it for something else. But first I want to check that there are no "originals" on the drive. The drive is mounted at `/media/portable`.

```
# Find all files on /media/portable that can be safely deleted:
$ rmlint -km /media/portable // ~
# check the shellscript looks ok:
$ less ./rmlint.sh
# run the shellscript to delete the redundant backups
$ ./rmlint.sh
# run again (to delete empty dirs)
$ rmlint -km /media/portable // ~
$ ./rmlint.sh
# see what files are left:
$ tree /media/portable
# recover any files that you want to save, then you can safely reformat the drive
```

In the case of nested mountpoints, it may sometimes makes sense to use the opposite variations, `-K` (`--keep-all-untagged`) and `-M` (`--must-match-untagged`).

## 2.6 Finding duplicate directories

---

**Note:** `--merge-directories` is still an experimental option that is non-trivial to implement. Please double check the output and report any possible bugs.

---

As far as we know, `rmlint` is the only duplicate finder that can do this. Basically, all you have to do is to specify the `-D` (`--merge-directories`) option and `rmlint` will cache all duplicate until everything is found and then merge them into full duplicate directories (if any). All other files are printed normally.

This may sound simple after all, but there are some caveats you should know of.

Let's create a tricky folder structure to demonstrate the feature:

```
$ mkdir -p fake/one/two/ fake/one/two_copy fake/one_copy/two fake/one_copy/two_copy
$ echo xxx > fake/one/two/file
$ echo xxx > fake/one/two_copy/file
$ echo xxx > fake/one_copy/two/file
$ echo xxx > fake/one_copy/two_copy/file
$ echo xxx > fake/file
$ echo xxx > fake/another_file
```

Now go run `rmlint` on it like that:

```
$ rmlint fake -D -S a
# Duplicate Directorie(s):
  ls -la /home/sahib/rmlint/fake/one
  rm -rf /home/sahib/rmlint/fake/one_copy
  ls -la /home/sahib/rmlint/fake/one/two
  rm -rf /home/sahib/rmlint/fake/one/two_copy

# Duplicate(s):
  ls /home/sahib/rmlint/fake/another_file
  rm /home/sahib/rmlint/fake/one/two/file
  rm /home/sahib/rmlint/fake/file

==> In total 6 files, whereof 5 are duplicates in 1 groups.
==> This equals 20 B of duplicates which could be removed.
```



As you can see it correctly recognized the copies as duplicate directories. Also, it did not stop at `fake/one` but also looked at what parts of this original directory could be possibly removed too.

Files that could not be merged into directories are printed separately. Note here, that the original is taken from a directory that was preserved. So exactly one copy of the `xxx-content` file stays on the filesystem in the end.

`rmlint` finds duplicate directories by counting all files in the directory tree and looking up if there's an equal amount of duplicate and empty files. If so, it tries the same with the parent directory.

Some file like hidden files will not be recognized as duplicates, but still added to the count. This will of course lead to unmerged directories. That's why the `-D` option implies the `-r` (`--hidden`) and `-l` (`--hardlinked`) option in order to make this convenient.

A note to symbolic links: The default behaviour is to not follow symbolic links, but to compare the link targets. If the target is the same, the link will be the same. This is a sane default for duplicate directories, since twin copies often are created by doing a backup of some files. In this case any symlinks in the backedup data will still point to the same target. If you have symlinks that reference a file in each respective directory tree, consider using `-f`.

**Warning:** Do *never ever* modify the filesystem (especially deleting files) while running with the `-D` option. This can lead to mismatches in the file count of a directory, possibly causing dataloss. You have been warned!

Sometimes it might be nice to only search for duplicate directories, banning all the sole files from littering the screen. While this will not delete all files, it will give you a nice overview of what you copied where.

Since duplicate directories are just a lint type as every other, you can just pass it to `-T: -T "none +dd"` (or `-T "none +duplicatedirs"`). There's also a preset of it to save you some typing: `-T minimaldirs`.

## 2.7 Misc options

If you read so far, you know `rmlint` pretty well by now. Here's just a list of options that are nice to know, but not essential:

- `-r` (`--hidden`): Include hidden files and directories - this is to save you from destroying git repositories (or similar programs) that save their information in a `.git` directory where `rmlint` often finds duplicates.

If you want to be safe you can do something like this:

```
$ find . | grep -v '\(.git\|.svn\)' | rmlint -
```

But you would have checked the output anyways?

- If something ever goes wrong, it might help to increase the verbosity with `-v` (up to `-vvv`).
- Usually the commandline output is colored, but you can disable it explicitly with `-w` (`--with-color`). If `stdout` or `stderr` is not a terminal anyways, `rmlint` will disable colors itself.
- You can limit the traversal depth with `-d` (`--max-depth`):

```
$ rmlint -d 0
<finds everything in the same working directory>
```

- If you want to prevent `rmlint` from crossing mountpoints (e.g. scan a home directory, but not the HD mounted in there), you can use the `-X` (`--no-crossdev`) option.
- It is possible to tell `rmlint` that it should not scan the whole file. With `-q` (`--clamp-low`) / `-Q` (`--clamp-top`) it is possible to limit the range to a starting point (`-q`) and end point (`-Q`). The point where to start might be either given as percent value, factor (percent / 100) or as an absolute offset.

If the file size is lower than the absolute offset, the file is simply ignored.

This feature might prove useful if you want to examine files with a constant header. The constant header might be different, i.e. by a different ID, but the content might be still the same. In any case it is advisable to use this option with care.

Example:

```
# Start hashing at byte 100, but not more than 90% of the filesize.  
$ rmlint -q 100 -Q.9
```

---

### Frequently Asked Questions

---

#### 3.1 `rmlint` finds more/less dupes than tool X!

Make sure that *none* of the following applies to you:

Both tools might investigate a different number of files. `rmlint` e.g. does not look through hidden files by default, while other tools might follow symlinks by default. Suspicious options you should look into are:

- `--hidden`: Disabled by default, since it might screw up `.git/` and similar directories.
- `--hardlinked`: Might find larger amount files, but not more lint itself.
- `--followlinks`: Might lead `rmlint` to different places on the filesystem.
- `--merge-directories`: pulls in both `--hidden` and `--hardlinked`.

If there's still a difference, check with another algorithm. In particular use `-ppp` to enable paranoid mode. Also make sure to have `-D (--merge-directories)` disabled to see the raw number of duplicate files.

Still here? Maybe talk to us on the [issue tracker](#).

#### 3.2 Can you implement feature X?

Depends. Go to to the [issue tracker](#) and open a feature request.

Here is a list of features where you probably have no chance:

- Port it to Windows.
- Find similar files like `ssdeep` does.
- Make a graphical user interface (totally okay as separate project, though).



These chapters are informative and are not essential for the average user. People that want to extend **rmlint** might want to read this though:

### 4.1 Developer's Guide

This guide is targeted to people that want to write new features or fix bugs in rmlint.

#### 4.1.1 Bugs

Please use the issue tracker to post and discuss bugs and features:

<https://github.com/sahib/rmlint/issues>

#### 4.1.2 Philosophy

We try to adhere to some principles when adding features:

- Try to stay compatible to standard unix' tools and ideas.
- Try to stay out of the users way and never be interactive.
- Try to make scripting as easy as possible.
- **Never** make `rmlint` modify the filesystem itself, only produce output to let the user easily do it.

Also keep this in mind, if you want to make a feature request.

#### 4.1.3 Making contributions

The code is hosted on GitHub, therefore our preferred way of receiving patches is using GitHub's pull requests (normal git pull requests are okay too of course).

---

**Note:** `origin/master` should always contain working software. Base your patches and pull requests always on `origin/develop`.

---

Here's a short step-by-step:

1. Fork it.
2. Create a branch from `develop`. (`git checkout develop && git checkout -b my_feature`)
3. Commit your changes. (`git commit -am "Fixed it all."`)
4. Check if your commit message is good. (If not: `git commit --amend`)
5. Push to the branch (`git push origin my_feature`)
6. Open a [Pull Request](#).
7. Enjoy a refreshing Tea and wait until we get back to you.

Here are some other things to check before submitting your contribution:

- Does your code look alien to the other code? Is the style the same?
- Do all tests run? (Simply run `nosetests` to find out) Also after opening the pull request, your code will be checked via [TravisCI](#).
- Is your commit message descriptive?
- Is `rmlint` running okay inside of `valgrind` (i.e. no leaks and no memory violations)?

For language-translations/updates it is also okay to send the `.po` files via mail at [sahib@online.de](mailto:sahib@online.de), since not every translator is necessarily a software developer.

## 4.1.4 Buildsystem Helpers

### Environement Variables

**CFLAGS** Extra flags passed to the compiler.

**LDFLAGS** Extra flags passed to the linker.

**CC** Which compiler to use?

```
# Use clang and enable profiling, verbose build and enable debugging
CC=clang CFLAGS='-pg' LDFLAGS='-pg' scons VERBOSE=1 DEBUG=1
```

### Variables

**DEBUG** Enable debugging symbols for `rmlint`. This should always be enabled during developement. Backtraces wouldn't be useful elsewhise.

**VERBOSE** Print the exact compiler and linker commands. Useful for troubleshooting build errors.

### Arguments

**-prefix** Change the installation prefix. By default this is `/usr`, but some users might prefer `/usr/local` or `/opt`.

**--actual-prefix** This is mainly useful for packagers. The `rmlint` binary knows where it is installed (which is needed to set e.g. the path to the gettext files). When installing a package, most of the time the build is installed to a local test environment first before being packed to `/usr`. In this case the `--prefix` would be set to the path of the temporary build env, while `--actual-prefix` would be set to `/usr`.

## Notable targets

**install** Install all program parts system-wide.

**config** Print a summary of all features that will be compiled and what the environment looks like.

**man** Build the manpage.

**docs** Build the online html docs (which you are reading now).

**test** Build the tests (requires `python` and `nosetest` installed). Optionally `valgrind` can be installed to run the tests through `valgrind`:

```
$ USE_VALGRIND=1 nosetests # or nosetests-3.3, python3 needed.
```

**xgettext** Extract a gettext `.pot` template from the source.

## 4.1.5 Sourcecode layout

- All C-source lives in `src`, the file names should be self explanatory.
- All documentation is inside `docs`.
- All translation stuff should go to `po`.
- All packaging should be done in `pkg/<distribution>`.
- Tests are written in Python and live in `tests`.

## 4.1.6 Hashfunctions

Here is a short comparasion of the existing `hashfunctions` in `rmlint` (linear scale). For reference: Those plots were rendered with [these](#) sources - which are very ugly, sorry.

If you want to add new hashfunctions, you should have some arguments why it is valueable and possibly even benchmark it with the above scripts to see if it's really that much faster.

Also keep in mind that most of the time the hashfunction is not the bottleneck.

## 4.1.7 Optimizations

For sake of overview, here is a short list of optimizations implemented in `rmlint`:

### Obvious ones

- Do not compare each file with each other by content, use a hashfunction to reduce comparison overhead drastically (introduces possibility of collisions though).
- Only compare files of same size with each other.

- Use incremental hashing, i.e. hash block-wise each size group and stop as soon a difference occurs or the file is read fully.

### Subtle ones

- Check only executable files to be non-stripped binaries.
- Use `preadv(2)` based reading for small speedups.
- Every thread in `rmlint` is shared, so only few calls to `pthread_create` are made.

### Insane ones

- Check the device ID of each file to see if it on a rotational (normal hard disks) or on a non-rotational device (like a SSD). On the latter the file might be processed by several threads.
- Use `fiemap ioctl(2)` to analyze the harddisk layout of each file, so each block can read it in *perfect* order on a rotational device.
- Use a common buffer pool for IO buffers.
- Use only one hashsum per group of same-sized files.
- Implement paranoia check as hash sum, so large chunks of the file are read and compared at one time. The total memory used for this can be configured by `--max-paranoid-ram`.

## 4.2 Translating `rmlint`

Rudimentary support for internationalization is provided via `gettext`.

### 4.2.1 Adding new languages

```
# Fork a new .po file from the po-template (here swedish):
$ msginit -i po/rmlint.pot -o po/se.po --locale se --no-translator

# Edit the po/se.po file, the format is self describing
$ vim po/se.po

# .po files need to be compiled, but that's handled by scons already.
$ scons
$ scons install

# You should see your changes now:
$ LANG=se ./rmlint
```

If you'd like to contribute your new translation you want to do a pull request (if you really dislike that, you may also send the translation to us via mail). [Here](#) is a small introduction on Pull Requests.

### 4.2.2 Updating existing languages



```
# Edit the file to your needs:
$ vim po/xy.po

# Install:
$ scons install

# Done
$ LANG=xy ./rmlint
```

### 4.2.3 Marking new strings for translations

If you want to mark strings in the C-code to be translated, you gonna need to mark them so the `xgettext` can find it. The latter tool goes through the source and creates a template file with all translations left out.

```
/* Mark the string with the _() macro */
fprintf(out, _("Stuff is alright: %s\n"), (alright) ? "yes" : "no");
```

It gets a little harder when static strings need to be marked, since they cannot be translated during compile time. You have to mark them first and translate them at a later point:

```
static const char * stuff = _N("Hello World");

void print_world(void) {
    printf("World is %s\n", _(stuff));
}
```

After you're done with marking the new strings, you have to update the template:

```
# scons can do this for you already:
$ scons xgettext
```

You need to add the new strings to the existing translations now:

```
$ msgmerge po/de.po po/rmlint.pot > po/de_new.po
$ EDITOR po/de_new.po # check if everything was merged alright.
$ mv po/de_new.po po/de.po
```

After that you can translate the new strings and proceed like in the upper steps.

## 4.3 rmlint

### 4.3.1 find duplicate files and other space waste efficiently

#### SYNOPSIS

```
rmlint [TARGET_DIR_OR_FILES ...] [/] [TARGET_DIR_OR_FILES ...] [-] [OPTIONS]
```

#### DESCRIPTION

`rmlint` finds space waste and other broken things on your filesystem and offers to remove it. Types of waste include:

- Duplicate files.

- Nonstripped Binaries (Binaries with debug symbols).
- Broken links.
- Empty files and directories.
- Files with broken user or group id.

In order to find the lint, `rmlint` is given one or more directories to traverse. If no directories or files were given, the current working directory is assumed. `rmlint` will take care of things like filesystem loops and symlinks during traversing.

Found duplicates are divided into the original and duplicates. Original are what `rmlint` thinks to be the file that was first there. You can drive the original detection with the `-S` option. If you know which path contains the originals you can prefix the path with `//`,

**Note:** `rmlint` will not delete any files. It only produces executable output for you to remove it.

## OPTIONS

### General Options

**-T --types="description" (default: defaults)** Configure the types of lint `rmlint` is supposed to find. The *description* string enumerates the types that shall be investigated, separated by a space, comma or semicolon (actually more separators work). At the beginning of the string certain groups may be specified.

- `all`: Enables all lint types.
- `defaults`: Enables all lint types, but `nonstripped`.
- `minimal`: `defaults` minus `emptyfiles` and `emptydirs`.
- `minimaldirs`: `defaults` minus `emptyfiles`, `emptydirs` and `duplicates`, but with `duplicatedirs`.
- `none`: Disable all lint types.

All following lint types must be one of the following, optionally prefixed with a `+` or `-` to select or deselect it:

- `badids,bi`: Find bad UID, GID or files with both.
- `badlinks,bl`: Find bad symlinks pointing nowhere.
- `emptydirs,ed`: Find empty directories.
- `emptyfiles,ef`: Find empty files.
- `nonstripped,ns`: Find nonstripped binaries. (**Warning:** slow)
- `duplicates,df`: Find duplicate files.
- `duplicatedirs,dd`: Find duplicate directories.

**Warning:** It is good practice to enclose the description in quotes. In obscure cases argument parsing might fail in weird ways:

```
# -ed is recognized as -e and -d here, -d takes "-s 10M" as parameter.
# This will fail to do the supposed, finding also files smaller than 10M.
$ rmlint -T all -ef -ed -s10M /media/music/
```

**-o --output=spec / -O --add-output=spec (default: -o sh:rmlint.sh -o pretty:stdout -o summary:stdout)**

Configure the way rmlint outputs it's results. You link a formatter to a file through `spec`. A file might either be an arbitrary path or `stdout` or `stderr`. If file is omitted, `stdout` is assumed.

If this options is specified, rmlint's defaults are overwritten. The option can be specified several times and formatters can be specified more than once for different files.

**--add-output** works the same way, but does not overwrite the defaults. Both **-o** and **-O** may not be specified at the same time.

For a list of formatters and their options, look at the **Formatters** section below.

**-c --config=spec [=value] (default: none)** Configure a formatter. This option can be used to fine-tune the behaviour of the existing formatters. See the **Formatters** section for details on the available keys.

If the value is omitted it is set to a truthy value.

**:-a --algorithm=name (default: spooky):**

Choose the hash algorithm to use for finding duplicate files. The following well-known algorithms are available:

**spooky, city, murmur, md5, sha1, sha256, sha512.**

If not explicitly stated in the name the hashfunctions use 128 bit. There are variations of the above functions:

- **bastard:** 256bit, half seeded **city**, half **murmur**.
- **city256, city512, murmur256, murmur512:** Slower variations with more bits.
- **spooky32, spooky64:** Faster version of **spooky** with less bits.
- **paranoid:** No hash function, compares files byte-by-byte.

**-v --loud / -V --quiet** Increase or decrease the verbosity. You can pass these options several times. This only affects rmlint's logging on `stderr`, but not the outputs defined with **-o**.

**-p --paranoid / -P --less-paranoid (default)** Increase the paranoia of rmlints internals. Both options can be specified up to three times. They do not do any work themselves, but set some other options implicitly as a shortcut.

- **-p** is equivalent to **--algorithm=bastard**
- **-pp** is equivalent to **--algorithm=sha512**
- **-ppp** is equivalent to **--algorithm=paranoid**

The last one is not a hash function in the traditional meaning, but performs a byte-by-byte comparison of each file. See also **--max-paranoid-ram**.

For the adventurous, it is also possible to decrease the default paranoia:

- **-P** is equivalent to **--algorithm spooky64**
- **-PP** is equivalent to **--algorithm spooky32**
- **-PPP** is equivalent to **--algorithm debian\_random**

*This is really not recommended though.*

**-D --merge-directories ([experimental] default: disabled)** Makes rmlint use a special mode where all found duplicates are collected and checked if whole directory trees are duplicates. This is an **HIGHLY EXPERIMENTAL FEATURE** and was/is tricky to implement right. Use with caution.

You always should make sure that the investigated directory is not modified during rmlint or it's removal scripts run.

Output is deferred until all duplicates were found. Sole duplicate groups are printed after the directories.

**--sortcriteria** applies for directories too, but 'p' or 'P' (path index) has no defined (useful) meaning. Sorting takes only place when the number of preferred files in the directory differs.

*Notes:*

- This option pulls in `-r` (`--hidden`) and `-l` (`--hardlinked`) for convenience.
- This feature might not deliver perfect result in corner cases.
- This feature might add some runtime.
- Consider using `-FF` together with this option (this is the default).

**-q --clamp-low=[fac.tor|percent%|offset] (default: 0) / -Q --clamp-top=[fac.tor|percent%|offset]**

The argument can be either passed as factor (a number with a `.` in it), a percent value (suffixed by `%`) or as absolute number or size spec, like in `--size`.

Only look at the content of files in the range of from `low` to (including) `high`. This means, if the range is less than `-q0%` to `-Q100%`, than only partial duplicates are searched. If the actual file size would be 0, the file is ignored during traversing. Be careful when using this function, you can easily get dangerous results for small files.

This is useful in a few cases where a file consists of a constant sized header or footer. With this option you can just compare the data in between. Also it might be useful for approximate comparison where it suffices when the file is the same in the middle part.

**-u --max-paranoid-ram=size** Apply a maximum number of bytes to use for **-paranoid**. The `size-description` has the same format as for **-size**.

**-w --with-color (default) / -W --no-with-color** Use color escapes for pretty output or disable them. If you pipe *rmlints* output to a file `-W` is assumed automatically.

**-h --help / -H --version** Show this manual or print the version string.

## Traversal Options

**-t --threads=N\*\* (default: 16)** The number of threads to use during file tree traversal and hashing. *rmlint* probably knows better than you how to set the value.

**-s --size=range (default: all)** Only consider files in a certain size range. The format of *range* is *min-max*, where both ends can be specified as a number with an optional multiplier. The available multipliers are:

- *C* ( $1^1$ ), *W* ( $2^1$ ), *B* ( $512^1$ ), *K* ( $1000^1$ ), *KB* ( $1024^1$ ), *M* ( $1000^2$ ), *MB* ( $1024^2$ ), *G* ( $1000^3$ ), *GB* ( $1024^3$ ),
- *T* ( $1000^4$ ), *TB* ( $1024^4$ ), *P* ( $1000^5$ ), *PB* ( $1024^5$ ), *E* ( $1000^6$ ), *EB* ( $1024^6$ )

The size format is about the same as *dd(1)* uses. Example: **"100KB-2M"**.

It's also possible to specify only one size. In this case the size is interpreted as "up to this size".

**-d --max-depth=depth (default: INF)** Only recurse up to this depth. A depth of 1 would disable recursion and is equivalent to a directory listing.

**-l --hardlinked / -L --no-hardlinked (default)** By default `rmlint` will not allow several files with the same *inode* and therefore keep only one of them in it's internal list. If `-l` is specified the whole group is reported instead.

**-f --followlinks / -F --no-followlinks / -FF --see-symlinks (\*\*default\*\*)** Follow symbolic links? If file system loops occur `rmlint` will detect this. If `-F` is specified, symbolic links will be ignored completely, if the `-F` is specified once more `rmlint` will see symlinks and treats them like small files with the path to their target in them. The latter is the default behaviour, since it is a sensible default for `--merge-directories`.

**Note:** Hardlinks are always followed, but it depends on `-L` how those are handled.

**-x --crossdev (default) / -X --no-crossdev** Do cross over mount points (`-x`)? Or stay always on the same device (`-X`)?

**-r --hidden / -R --no-hidden (default)** Also traverse hidden directories? This is often not a good idea, since directories like `.git/` would be investigated.

**-b --match-basename / -B --no-match-basename (default)** Only consider those files as dupes that have the same basename. See also `man 1 basename`.

**-e --match-with-extension / -E --no-match-with-extension (default)** Only consider those files as dupes that have the same file extension. For example two photos would only match if they are a `.png`.

**-i --match-without-extension / -I --no-match-without-extension (default)** Only consider those files as dupes that have the same basename minus the file extension. For example: `banana.png` and `banana.jpeg` would be considered, while `apple.png` and `peach.png` won't.

**-n --newer-than-stamp=<timestamp\_filename> / -N --newer-than=<iso8601\_timestamp\_or\_unix>** Only consider files (and their size siblings for duplicates) newer than a certain modification time (*mtime*). The age barrier may be given as seconds since the epoch or as ISO8601-Timestamp like `2014-09-08T00:12:32+0200`.

`-n` expects a file from where it can read the timestamp from. After `rmlint` run, the file will be updated with the current timestamp. If the file does not initially exist, no filtering is done but the stampfile is still written.

`-N` in contrast takes the timestamp directly and will not write anything.

If you want to take **only** the files (and not their size siblings) you can use `find(1)`:

```
• find -mtime -1 | rmlint - # find all files younger than a day
```

*Note:* you can make `rmlint` write out a compatible timestamp with:

```
• -O stamp:stdout # Write a seconds-since-epoch timestamp to
  stdout on finish.
• -O stamp:stdout -c stamp:iso8601 # Same, but write as ISO8601.
```

## Original Detection Options

**-k --keep-all-tagged / -K --keep-all-untagged (default)** Don't delete any duplicates that are in original paths. (Paths that were named after `//`).

**Note:** for lint types other than duplicates, `--keep-all-tagged` option is ignored.

**-m --must-match-tagged / -M --must-match-untagged (default)** Only look for duplicates of which one is in original paths. (Paths that were named after `//`).

**-S --sortcriteria=criteria (default: *m*)**

- **m**: keep lowest mtime (oldest) **M**: keep highest mtime (newest)
- **a**: keep first alphabetically **A**: keep last alphabetically
- **p**: keep first named path **P**: keep last named path

One can have multiple criteria, e.g.: `-S am` will choose first alphabetically; if tied then by mtime.

**Note:** original path criteria (specified using `//`) will always take first priority over `-S` options.

## FORMATTERS

- **csv**: Format all found lint as comma-separated-value list.

Available options:

- `no_header`: Do not write a first line describing the column headers.

- **sh**: Format all found lint as shellscript. Sane defaults for most lint-types are set. This formatter is activated as default.

Available options:

- `use_ln`: Instead of just deleting duplicates remove them and replace them with hardlinks (if they are on the same partition) or with symlinks if they're on different devices.
- `symlinks_only`: Only relevant with `use_ln`, always use symbolic links, never use hardlinks.

- **json**: Print a JSON-formatted dump of all found reports. Outputs all finds as a json document. The document is a list of dictionaries, where the first and last element is the header and the footer respectively, everything between are data-dictionaries.

Available options:

- `use_header=[true|false]`: Print the header with metadata.
- `use_footer=[true|false]`: Print the footer with statistics.

- **py**: Outputs a python script and a JSON document, just like the **json** formatter. The JSON document is written to `.rmlint.json`, executing the script will make it read from there. This formatter is mostly intended for complex usecases where the lint needs special handling. Therefore the python script can be modified to do things standard `rmlint` is not able to do easily.

- **stamp**:

Outputs a timestamp of the time `rmlint` was run.

Available options:

- `iso8601=[true|false]`: Write an ISO8601 formatted timestamps or seconds since epoch?

- **progressbar**: Shows a progressbar. This is meant for use with **stdout** or **stderr**.

Available options:

- `update_interval=number`: Number of files to wait between updates. Higher values use less resources.

- **pretty**: Shows all found items in realtimes nicely colored. This formatter is activated as default.
- **summary**: Shows counts of files and their respective size after the run. Also list all written files.

## EXAMPLES

- `rmlint`  
Check the current working directory for duplicates.
- `find ~/pics -iname '*.png' | ./rmlint -`  
Read paths from *stdin* and check all png files for duplicates.
- `rmlint files_backup // files --keep-all-tagged --must-match-tagged`  
Check for duplicate files between the current files and the backup of it. Only files in *files\_backup* would be reported as duplicate. Additionally, all reported duplicates must occur in both paths.

## PROBLEMS

1. **False Positives:** Depending on the options you use, there is a very slight risk of false positives (files that are erroneously detected as duplicate). Internally a hashfunctions is used to compute a *fingerprint* of a file. These hashfunctions may, in theory, map two different files to the same fingerprint. This happens about once in  $2^{64}$  files. Since `rmlint` computes at least 3 hashes per file and requires them to be the same size, it's very unlikely to happen. If you're really wary, try the `-paranoid` option.
2. **File modification during or after rmlint run:** It is possible that a file that `rmlint` recognized as duplicate is modified afterwards, resulting in a different file. This is a general problem and cannot be solved from `rmlint`'s side alone. You should **never modify the data until “rmlint” and the shellscript has been run through**. Careful persons might even consider to mount the filesystem you are scanning read-only.

## SEE ALSO

- *find(1)*
- *rm(1)*

Extended documentation and an in-depth tutorial can be found at:

- <http://rmlint.rtfid.org>

## BUGS

If you found a bug, have a feature requests or want to say something nice, please visit <https://github.com/sahib/rmlint/issues>.

Please make sure to describe your problem in detail. Always include the version of `rmlint` (`--version`). If you experienced a crash, please include at least one of the following information with a debug build of `rmlint`:

- `gdb --ex run -ex bt --args rmlint -vvv [your_options]`
- `valgrind --leak-check=no rmlint -vvv [your_options]`

You can build a debug build of `rmlint` like this:

- `git clone git@github.com:sahib/rmlint.git`
- `cd rmlint`
- `scons DEBUG=1`
- `sudo scons install # Optional`

## **LICENSE**

`rmlint` is licensed under the terms of the GPLv3.

See the COPYRIGHT file that came with the source for more information.

## **PROGRAM AUTHORS**

`rmlint` was written by:

- Christopher <sahib> Pahl 2010-2014 (<https://github.com/sahib>)
- Daniel <SeeSpotRun> T. 2014-2014 (<https://github.com/SeeSpotRun>)

Also see the THANKS file for other people that helped us.

If you consider a donation you can use *Flattr* or buy us a beer if we meet:

<https://flattr.com/thing/302682/libglyr>



## CHAPTER 5

---

### Authors

---

**rmlint** was and is written by:

<i>Christopher Pahl</i>	<a href="https://github.com/sahib">https://github.com/sahib</a>	2010-2014
<i>Daniel Thomas</i>	<a href="https://github.com/SeeSpotRun">https://github.com/SeeSpotRun</a>	2014-2014

Additional thanks to:

- [qitta](#) (Ideas & Testing)
- [dieterbe](#) (Bugs & -D was his idea)
- [My cats](#).
- Attila Toth
- All sane bugreporters (there are not many)
- All packagers, porters and patchers.



## CHAPTER 6

---

### License

---

**rmlint** is licensed under the terms of [GPLv3](#).




## CHAPTER 7

---

### Donations

---

If you think rmlint saved<sup>\*0</sup> you some serious time and/or space, you might consider a donation. You can donate either via *Flattr* or via *PayPal*: 

Or just buy us a beer if we ever meet. Nice emails are okay too.

---

<sup>0</sup> If it freed you from your beloved data: *Sorry.*<sup>†0</sup>