rmlint documentation

Release

May 01, 2018

Contents

1 Installation **3**
- 1.1 Dependencies . . . 3
- 1.2 Compilation . . . 4

2 Gentle Guide to `rmlint` **5**
- 2.1 Beginner Examples . . . 5
- 2.2 Filtering input files . . . 6
- 2.3 Outputs & Formatters . . . 8
- 2.4 Paranoia mode . . . 10
- 2.5 Original detection / selection . . . 11
- 2.6 Finding duplicate directories . . . 12
- 2.7 Miscellaneous options . . . 14

3 Frequently Asked Questions **15**
- 3.1 `rmlint` finds more/less dupes than tool `X`! . . . 15
- 3.2 Can you implement feature `X`? . . . 15
- 3.3 I forgot to add some options before running on a large dataset. Do I need to re-run it? . . . 16
- 3.4 I have a very large number of files and I run out of memory and patience. . . . 16

4 Informative reference **17**
- 4.1 Developer's Guide . . . 17
- 4.2 Translating `rmlint` . . . 22
- 4.3 Benchmarks . . . 24
- 4.4 rmlint . . . 24

5 Authors **35**

6 License **37**

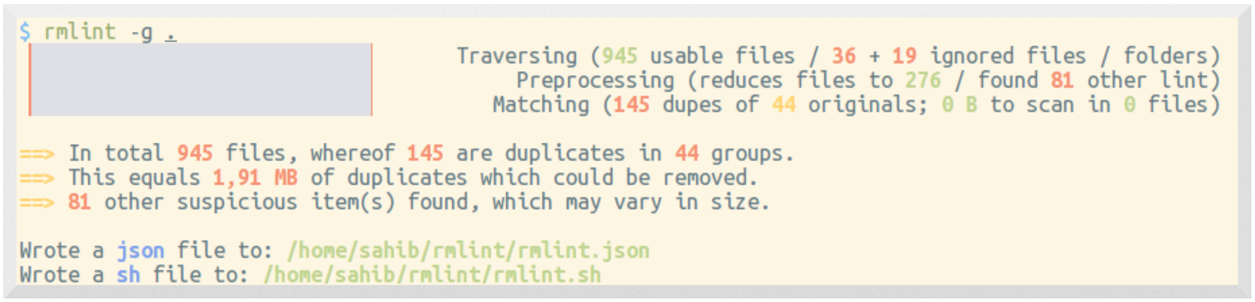
7 Donations **39**

`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.
- Flexible and easy commandline options.
- Choice of several hashes for hash-based deplicate detection
- Option for exact byte-by-byte comparison (only slightly slower).
- Numerous output options.
- Option to store time of last run; next time will only scan new files.
- Many options for original selection / prioritisation.
- Can handle very large file sets (millions of files).
- Colorful progressbar. ()



```
$ rmlint -g .
                              Traversing (945 usable files / 36 + 19 ignored files / folders)
                                  Preprocessing (reduces files to 276 / found 81 other lint)
                                 Matching (145 dupes of 44 originals; 0 B to scan in 0 files)

==> In total 945 files, whereof 145 are duplicates in 44 groups.
==> This equals 1,91 MB of duplicates which could be removed.
==> 81 other suspicious item(s) found, which may vary in size.

Wrote a json file to: /home/sahib/rmlint/rmlint.json
Wrote a sh file to: /home/sahib/rmlint/rmlint.sh
```

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:

CHAPTER 1

Installation

Many major Linux distribution might already package `rmlint` – but watch out for the version. This manual describes the rewrite of `rmlint` (i.e. version ≥ 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** ≥ 2.32 (general C Utility Library)

1.1.2 Soft dependencies:

- **libblkid** (detecting mountpoints)
- **libelf** (nonstripped binary detection)
- **libjson-glib** (parsing rmlint's own json as caching layer)

1.1.3 Build dependencies:

- **git** (version control)
- **scons** (build system)
- **sphinx** (manpage/documentation generation)
- **gettext** (support for localization)

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

- **Fedora:**

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

  There are also pre-built packages on Fedora Copr:

  ```
  $ dnf copr enable sahib/rmlint
  $ dnf install rmlint
  ```

  Those packages are built from master snapshots and might be slightly outdated.

- **ArchLinux:**

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

  There is also a PKGBUILD on the ArchLinux AUR:

  ```
  $ # Use your favourite AUR Helper.
  $ yaourt -S rmlint-git
  ```

  It is built from git master.

- **Ubuntu:**

  ```
  $ apt-get install git scons python3-sphinx python3-nose gettext build-essential
  $ apt-get installlibelf-dev libglib2.0-dev libblkid-dev libjson-glib libjson-glib-
  ↪dev
  ```

- **FreeBSD:**

  ```
  $ pkg install git scons py27-sphinx
  $ pkg install glib gettext libelf
  ```

  Also `rmlint` is maintained as port:

  ```
  $ cd /usr/ports/sysutils/rmlint && make install
  ```

Send us a note if you want to see your distribution here. The commands above install the full dependencies, therefore some packages might be stripped if you do not need the feature they enable. Only hard requirement is `glib`.

1.2 Compilation

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

```
$ # Omit -b develop if you want to build from the stable master
$ 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 --help` or `man 1 rmlint`.

CHAPTER 2

Gentle Guide to `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*[0]. 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:

```
$ rmlint some/path
# (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)
```

[0] You could say it should be named `findlint`.

```
```

On larger runs, it might be more preferable to show a progressbar instead of a long list of files. You can do this easily with the `-g` switch:

```
$ rmlint -g
```

2.2 Filtering input files

What if we do not want to check all files as dupes? `rmlint` has a good reportoire 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 progessively 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 supstantial 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 & Formatters

`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.

Note: If you just came here to learn how to print a nice progressbar: Just use the `-g` (`--progress`) option:

```
$ rmlint -g -VVV /usr  # -VVV is just to prevent pointless warnings
```

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,
  "progress": 100,
  "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 or just omit the colon which will print everything to `stdout`.

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 proram 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`. Addionally 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
remove_cmd '/home/user/test/b/file' # duplicate
remove_cmd '/home/user/test/a/two' # duplicate
remove_cmd '/home/user/test/a/file' # duplicate

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

Config values:

- *link*: Replace duplicate files with reflinks (if possible), hardlinks (if on same device as original) or with symbolic links (if not on same device as original).
- *cmd*: Provider a user specified command to execute on duplicates.
- *handler*: This option allows for more finegrained control. Please refer to the manpage here.

Example:

```
$ rmlint -o sh:stdout -o sh:rmlint.sh -c sh:link
...
echo  '/home/user/test/b/one' # original
cp_symlink '/home/user/test/b/file' '/home/user/test/b/one' # duplicate
$ ./rmlint.sh -d
Keeping: /home/user/test/b/one
Symlinking to original: /home/user/test/b/file
```

A safe user command example that just composes some string out of the original and duplicate path:

```
$ rmlint -o sh -c sh:cmd='echo "Stuff with" "$1" "->" "$2"'
```

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 intented 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.

fdupes A formatter that behaves similar to **fdupes(1)** - another duplicate finder. This is mostly indented for compatibility (e.g. scripts that relied on that format). Duplicate set of files are printed as block, each separated by a newline. Original files are highlighted in green (this is an addition). At the start and beginning a progressbar and summary is printed. The latter two are printed to `stderr`, while the parseable lines will be printed to `stdout`.

Consider using the far more powerful `json` output for scripting purposes.

2.4 Paranoia mode

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 probablilty of having a hash-collision is $\frac{1}{2^{160}}$ for the default 160-bit hash. Of course hash functions are not totally random, so the collision probability is slightly higher. Due to the "birthday paradox", this starts to become a real risk if you have more than about 2^{80} files.

If you're wary, you might want to make a bit more paranoid than the default. By default the `sha1` hash algorithm is used, which we consider a good trade-off 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 twice each).

Here's what they do in detail:

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

As you see, it just enables a certain hash algorithm. `--algorithm` changes the hash algorithm to someting more secure. 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). While this might sound slow it's often only a few seconds slower than the default behaviour.

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 1,000,000,000,000,000,000,000 different files all of the same file size. Bugs in `rmlint` are sadly (or happily?) more likely than hash collisions. See http://preshing.com/20110504/hash-collision-probabilities/ for discussion.

2.5 Original detection / selection

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.

By default, if you specify multiple paths in the rmlint command, the files in the first-named paths are treated as the originals. If there are two files in the same path, then the older one will be treated as the original. If they have the same modification time then it's just a matter of chance which one is selected as the original.

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:

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

The default setting is `-S pm`. 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 any `-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 duplicates 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 backupped 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`.

Warning: Also take note that `-D` will cause a higher memory footprint and might add a bit of processing time. This is due to the fact that all files need to be cached till the end and some other internal data structures need to be created.

2.7 Miscellaneous 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:

- Consecutive runs of `rmlint` can be speed up by using `--cache`.

  ```
  $ rmlint large_dataset/ -O json:cache.json --write-unfinished
  $ rmlint large_dataset/ -C cache.json
  ```

  Here, the second run should (or *might*) run a lot faster. But be sure to read the caveats stated in the manpage!

- `-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 all files except everything under .git or .svn folders
  $ find . -type d | grep -v '\(.git\|.svn\)' | rmlint - --hidden
  ```

  But you would have checked the output anyways, wouldn't you?

- 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 an 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 no 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
  ```

CHAPTER 3

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 `-pp` 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.

3.3 I forgot to add some options before running on a large dataset. Do I need to re-run it?

Probably. It's not as bad as it sounds though. Your filesystem is probably very good at caching.

Still there are some cases where re-running might take a long time, like running on network mounts. By default, `rmlint` writes a `rmlint.json` file along the `rmlint.sh`. This can be used to speed up the next run by passing it to `--cache`. It should be noted that using the cache file for later runs is discouraged since false positives will get likely if the data is changed in between. Therefore there will never be an "auto-pickup" of the cache file.

3.4 I have a very large number of files and I run out of memory and patience.

As a rule of thumb, `rmlint` will allocate *~150 bytes* for every file it will investigate. Additionally paths are stored in a patricia trie, which will compress paths and save memory therefore.

The memory peak is usually short after it finished traversing all files. For example, 5 million files will result in a memory footprint of roughly 1.0GB of memory in average.

If that's still not enough read on.

Some things to consider:

- Use `--with-metadata-cache` to swap paths to disk. When needed the path is selected from disk instead of keeping them all in memory. This lowers the memory footprint per file by a few bytes. Sometimes the difference may be very subtle since all paths in rmlint are stored by common prefix, i.e. for long but mostly identically paths the point after the difference is stored.

  This option will most likely only make sense if you files with long basenames. You might expect 10%-20% less memory as a rule of thumb.

- Use `--without-fiemap` on rotational disk to disable this optimization. With it enabled a table of the file's extents is stored to optimize disk access patterns. This lowers the memory footprint per file by around 50 bytes.
- Enable the progress bar with `-g` to keep track of how much data is left to scan.

Caveats:

- Some options like `-D` will not work well with `--with-metadata-cache` and use a fair bit of memory themselves. This is by the way they're working. Avoid them in this case. Also `--cache` might be not very memory efficient.
- The CPU usage might go up quite a bit, resulting longer runs.

Also:

`rmlint` have been successfully used on datasets of 5 million files. See this bug report for more information: #109.

If you have usage questions or find weird behaviour, you can also try to reach us via *IRC* in `#rmlint` on `irc.freenode.net`.

CHAPTER 4

Informative reference

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? You can run this command to make sure it is the same:

```
$ clang-format -style=file -i $(find lib src -iname '*.[ch]')
```

- Do all tests run? Go to the test documentation for more info. Also after opening the pull request, your code will be checked via TravisCI.
- Is your commit message descriptive? whatthecommit.com has some good examples how they should **not** look like.
- 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, since not every translator is necessarily a software developer.

4.1.4 Testsuite

`rmlint` has a not yet complete but quite powerful testsuite. It is not complete yet (and probably never will), but it's already an valueable boost of confidence in `rmlint's` correctness.

The tests are based on `nosetest` and are written in `python>=3.0`. Every testcase just runs the (previously built) `rmlint` binary a and parses it's json output. So they are technically blackbox-tests.

On every commit, those tests are additionally run on TravisCI.

Control Variables

The behaviour of the testsuite can be controlled by certain environment variables which are:

- `USE_VALGRIND`: Run each test inside of valgrind's memcheck. *(slow)*
- `PEDANTIC`: Run each test several times with different optimization options and check for errors between the runs. *(slow)*.
- `PRINT_CMD`: Print the command that is currently run.

Additionally slow tests can be omitted with by appending `-a '!slow'` to the commandline. More information on this syntax can be found on the nosetest documentation.

Before each release we call the testsuite (at least) like this:

```
$ sudo USE_VALGRIND=1 PRINT_CMD=1 PEDANTIC=1 nosetests-3.4 -s -a '!slow'
```

The `sudo` here is there for executing some tests that need root access (like the creating of bad user and group ids). Most tests will work without.

Coverage

To see which functions need more testcases we use `gcov` to detect which lines were executed (and how often) by the testsuite. Here's a short quickstart using `lcov`:

```
$ CFLAGS="-fprofile-arcs -ftest-coverage" LDFLAGS="-fprofile-arcs -ftest-coverage"␣
↪scons -j4 DEBUG=1
$ sudo USE_VALGRIND=1 PRINT_CMD=1 PEDANTIC=1 nosetests-3.4 -s -a '!slow'
$ lcov --capture --directory . --output-file coverage.info
$ genhtml coverage.info --output-directory out
```

The coverage results are updated from time to time here:

http://sahib.github.io/rmlint/gcov/index.html

Structure

```
tests
├── test_formatters   # Tests for output formatters (like sh or json)
├── test_options      # Tests for normal options like --merge-directories etc.
├── test_types        # Tests for all lint types rmlint can find
└── utils.py          # Common utilities shared amon tests.
```

Templates

A template for a testcase looks like this:

```
from nose import with_setup
from tests.utils import *

@with_setup(usual_setup_func, usual_teardown_func)
def test_basic():
    create_file('xxx', 'a')
    create_file('xxx', 'b')

    head, *data, footer = run_rmlint('-a city -S a')

    assert footer['duplicate_sets'] == 1
    assert footer['total_lint_size'] == 3
    assert footer['total_files'] == 2
    assert footer['duplicates'] == 1
```

Rules

- Test should be able to run as normal user.
- If that's not possible, check at the beginning of the testcase with this:

```
if not runs_as_root():
    return
```

- Regressions in `rmlint` should get their own testcase so they do not appear again.
- Slow tests can be marked with a slow attribute:

```
from nose.plugins.attrib import attr

@attr('slow')
@with_setup(usual_setup_func, usual_teardown_func)
def test_debian_support():
    assert random.choice([True, False]):
```

4.1.5 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`.

–libdir Some distributions like Fedora use separate libdirectories for 64/32 bit. If this happens, you should set the correct one for 64 bit with `--libdir=lib64`.

–without-libelf Do not link with `libelf`, which is needed for nonstripped binary detection.

–without-blkid Do not link with `libblkid`, which is needed to differentiate between normal rotational harddisks and non-rotational disks.

–without-json-glib Do not link with `libjson-glib`, which is needed to load json-cache files. Without this library a warning is printed when using `-C / --cache`.

–without-fiemap Do not attempt to use the `FIEMAP ioctl(2)`.

–without-gettext Do not link with `libintl` and do not compile any message catalogs.

–with-sse Allow the usage of `SSE 4.2` for CityHash if it is present. Binary packagers should not enable this to prevent crashes on hosts that do not support `SSE4.2`.

All `--without-*` options come with a `--with-*` option that inverses its effect. By default `rmlint` is built with all features available on the system, so you do not need to specify any `--with-*` option normally.

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 onlice 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.

dist Build a tarball suitable for release. Save it under `rmlint-$major-$minor-$patch.tar.gz`.

release Same as `dist`, but reads the `.version` file and replaces the current version in the files that are not built by *scons*.

4.1.6 Sourcecode layout

- All C-source lives in `lib`, the file names should be self explanatory.
- As an exception, the main lives in `src/rmlint.c`.
- 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`.

4.1.7 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 possiblye 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.

4.1.8 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.
- Create one hashing thread for each physical disk. This gives a big speedup if files are roughly evenly spread over multiple physical disks.

Subtle ones

- Check only executable files to be non-stripped binaries.
- Use `preadv(2)` based reading for small speeedups.
- 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 using the same algorithm as the incremental hash. The difference is that large chunks of the file are read and kept in memory instead of just keeping the hash in memory. This avoids the need for a two-pass algorithm (find matches using hashes then confirm via bytewise comparison). Each file is read once only. To our knowledge this is the first dupefinder which achieves bytewise comparison in O(N) time, even if there are large clusters of same-size files. The downside is that it is somewhat memory-intensive (the total memory used is set to 256 MB by default but can be configured by `--max-paranoid-mem` option.

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 Benchmarks

We will post some benchmark results here once the respective scripts are ready enough. Here, have some early one to see what they look like:

4.3.1 User benchmarks

If you like, you can add your own benchmarks below. Maybe include the following information:

- `rmlint --version`
- `uname -a` or similar.
- Hardware setup, in particular the filesystem.
- The summary printed by `rmlint` in the end.
- Did it match your expectations?

If you have longer output you might want to use a pastebin like gist.

4.4 rmlint

4.4.1 find duplicate files and other space waste efficiently

SYNOPSIS

rmlint [TARGET_DIR_OR_FILES ...] [//] [TAGGED_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 and directories.
- 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.
- `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/
# Actual user wanted to do this:
$ 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 true value.

`-z --perms[=[rwx]]` (default: ***no check*****)** Only look into file if it is readable, writable or executable by the current user. Which one of the can be given as argument as one of *rwx*.

If no argument is given, *"rw"* is assumed. Note that *r* does basically nothing user-visible since `rmlint` will ignore unreadable files anyways. It's just there for the sake of completeness.

By default this check is not done.

`-a --algorithm=name` (default: ***sha1*****)** 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 hash functions 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**.

`-g --progress / -G --no-progress` (default) Convenience shortcut for `-o progressbar -o summary -o sh:rmlint.sh`. It is recommended to run `-g` with `-VVV` to prevent the printing of warnings in between.

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

- **-p** is equivalent to **–algorithm=sha512**
- **-pp** 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-mem**.

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

- **-P** is equivalent to **–algorithm bastard**
- **-PP** is equivalent to **–algorithm spooky**

`-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 `--partial-hidden` and `-@` (`--see-symlinks`) for convenience.
- This feature might not deliver perfect result in corner cases.

- This feature might add some runtime.
- Consider using `-@` together with this option (this is the default).

`-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 --show-man` Show a shorter reference help text (`-h`) or this full man page (`-H`).

`--version` Print the version of rmlint. Includes git revision and compile time features.

Traversal Options

`-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` (default) / `-L --no-hardlinked` Wether to filter hardlinks after traversal (same *inode* and same *device*). If not filtered, there will be only one checksum built per hardlink cluster.

`-f --followlinks` / `-F --no-followlinks` / `-@ --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 an 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) / `--partial-hidden` Also traverse hidden directories? This is often not a good idea, since directories like `.git/` would be investigated. With `--partial-hidden` hidden are only shown if they're inside duplicate directories. Normal regular duplicates are not shown.

`-b --match-basename` / `-B --no-match-basename` (default) Only consider those files as dupes that have the same basename. See also `man 1 basename`. The comparison of the basenames is case-insensitive.

`-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`. The extension is compared case insensitive, so `.PNG` is the same as `.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. The comparison is also case-insensitive.

`-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: *pm*)

- **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

Alphabetical sort will only use the basename of the file and ignore it's case. 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.

Caching

`--xattr-read` / `--xattr-write` / `--xattr-clear` Read or write cached checksums from the extended file attributes. This feature can be used to speed up consecutive runs.

The same notes as in `--cache` apply.

NOTE: Many tools do not support extended file attributes properly, resulting in a loss of the information when copying the file or editing it. Also, this is a linux specific feature that works not on all filesystems and only if you write permissions to the file.

`-C --cache file.json` Read checksums from a *json* file. This *json* file is the same that is outputted via `-o json`, but you can also enrich the *json* with the checksums of sieved out files via `--write-unfinished`.

Usage example:

```
$ rmlint large_cluster/ -O json:cache.json -U   # first run.
$ rmlint large_cluster/ -C cache.json           # second run.
```

CAUTION: This is a potentially unsafe feature. The cache file might be changed accidentally, potentially causing `rmlint` to report false positives. As a security feature the *mtime* of each cached file is checked against the *mtime* of the time the checksum was created.

NOTE: The speedup you may experience may vary wildly. In some cases the parsing of the json file might take longer than the actual hashing. Also, the cached json file will not be of use when doing many modifications between the runs, i.e. causing an update of *mtime* on most files. This feature is mostly intended for large datasets in order to prevent the re-hashing of large files. If you want to ensure this, you can use `--size`.

`-U --write-unfinished` Include files in output that have not been hashed fully (i.e. files that do not appear to have a duplicate). This is mainly useful in conjunction with `--cache`. When re-running rmlint on a large dataset this can greatly speed up a re-run in some cases.

This option also applies for `--xattr-write`.

Rarely used, miscellaneous 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.

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

`-q --clamp-low=[fac.tor|percent%|offset]` (default: *0*) / `-Q --clamp-top=[fac.tor|percent%|offs`
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 `-q 0%` to `-Q 100%`, 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.

The shortcut `-q / -Q` can be easily remembered if you memorize the word `quantile` for it.

`--with-fiemap` (default) / `--without-fiemap` Enable or disable reading the file extents on rotational disk in order to optimize disk access patterns. Usually, this should be only disabled if you're low on memory since a table of extents have to be stored for every file. In exchange the IO speed will decrease. No extent data will be collected for non-rotational disks anyway.

`--with-metadata-cache` / `--without-metadata-cache` (default) Swap certain file metadata attributes onto disk in order to save memory. This can help to save memory for very big datasets (several million files) where storing the paths alone can eat up several GB RAM. Enabling swapping may cause slowdowns in exchange.

Sometimes the difference may be very subtle since all paths in rmlint are stored by common prefix, i.e. for long but mostly identically paths the point after the difference is stored.

This feature may not play nice with some other options, causing heavy load and long computations:

- The `--match-*` family of options (long)

- `--cache` might use more memory and takes longer.
- `--merge-directories` will not car about using the metadata cache yet.

Some of those restrictions might be removed in future `rmlint` versions.

The metadata cache will be stored in `$XDG_CACHE_HOME/rmlint/$pid`. If the cache cannot be created, `rmlint` falls back to no caching mode.

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 shell script. Sane defaults for most lint-types are set. This formatter is activated as default.

  Available options:

  – *cmd*: Specify a user defined command to run on duplicates. The command can be any valid `/bin/sh`-expression. The duplicate path and original path can be accessed via `"$1"` and `"$2"`. Not the actual command will be written to the script, but the content of the `user_command` function will be replaced with it.

  – *handler* Define a comma separated list of handlers to try on duplicate files in that given order until one handler succeeds. Handlers are just the name of a way of getting rid of the file and can be any of the following:

    * `reflink`: Try to reflink the duplicate file to the original. See also `--reflink` in `man 1 cp`. Fails if the filesystem does not support it.
    * `hardlink`: Replace the duplicate file with a hardlink to the original file. Fails if both files are not on the same partition.
    * `symlink`: Tries to replace the duplicate file with a symbolic link to the original. Never fails.
    * `remove`: Remove the file using `rm -rf`. (`-r` for duplicate dirs). Never fails.
    * `usercmd`: Use the provided user defined command (`-c sh:cmd=something`). Never fails.

    Default is `remove`.

  – *link*: Shortcut for `-c sh:reflink,hardlink,symlink`.

  – *hardlink*: Shortcut for `-c sh:hardlink,symlink`.

  – *symlink*: Shortcut for `-c sh:symlink`.

- `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:

  – *no_header=[true|false]:* Print the header with metadata.

  – *no_footer=[true|false]:* Print the footer with statistics.

  – *oneline=[true|false]:* Print one json document per line.

- `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 intented for complex use-cases 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**.

  See also: `-g` (`--progress`) for a convenience shortcut option.

  Available options:

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

  – *ascii:* Do not attempt to use unicode characters, which might not be supported by some terminals.

  – *fancy:* Use a more fancy style for the progressbar.
- `pretty`: Shows all found items in realtime 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.
- `fdupes`: Prints an output similar to the popular duplicate finder **fdupes(1)**. At first a progressbar is printed on **stderr.** Afterwards the found files are printed on **stdout;** each set of duplicates gets printed as a block separated by newlines. Originals are highlighted in green. At the bottom a summary is printed on **stderr**. This is mostly useful for scripts that are used to parsing this format. We recommend the `json` formatter for every other scripting purpose.

  Available options:

  – *omitfirst:* Same as the `-f / --omitfirst` option in `fdupes(1)`. Omits the first line of each set of duplicates (i.e. the original file.

  – *sameline:* Same as the `-1 / --sameline` option in `fdupes(1)`. Does not print newlines between files, only a space. Newlines are printed only between sets of duplicates.

EXAMPLES

This is a collection of common usecases and other tricks:

- Check the current working directory for duplicates.

  ```
  $ rmlint
  ```
- Reflink on btrfs, else try to hardlink duplicates to original. If that does not work, replace duplicate with a symbolic link:

  ```
  $ rmlint -c sh:link
  ```
- Inject user-defined command into shell script output:

  ```
  $ ./rmlint -o sh -c sh:cmd='echo "original:" "$2" "is the same as" "$1"'
  ```
- Quick re-run on large datasets:

  ```
  $ rmlint large_dir/ # First run; writes rmlint.json
  ```

  ```
  $ rmlint -C rmlint.json large_dir # Reads checksums from rmlint.json
  ```

- Search only for duplicates and duplicate directories

  ```
  $ rmlint -T df,dd .
  ```

- Compare files byte-by-byte in current directory:

  ```
  $ rmlint -pp .
  ```

- Find duplicates with same basename (but without extension):

  ```
  $ rmlint -e
  ```

- Do more complex traversal using `find(1)`.

  ```
  $ find /usr/lib -iname '*.so' -type f | rmlint - # find all duplicate .so
  files
  ```

  ```
  $ find ~/pics -iname '*.png' | ./rmlint - # compare png files only
  ```

- Limit file size range to investigate:

  ```
  $ rmlint -s 2GB # Find everything >= 2GB
  ```

  ```
  $ rmlint -s 0-2GB # Find everything < 2GB
  ```

- Only find writable and executable files:

  ```
  $ rmlint --perms wx
  ```

- Show a progressbar:

  ```
  $ rmlint -g
  ```

- Use *data* as master directory with all originals. Find only duplicates that are in *data* and *backup*. Do not delete any files in *data*:

  ```
  $ rmlint backup/ // data/ --keep-all-tagged --must-match-tagged
  ```

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.rtfd.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 http://rmlint.rtfd.org 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

The Changelog is also updated with new and futures features, fixes and overall changes.

CHAPTER 5

Authors

`rmlint` was and is written by:

Christopher Pahl	https://github.com/sahib	2010-2015
Daniel Thomas	https://github.com/SeeSpotRun	2014-2015

Additional thanks to:

- qitta (Ideas & Testing)
- vvs- (Scalability testing)
- 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*[0] 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.

[0] If it freed you from your beloved data: *Sorry.*†[0]