restic

Release 0.5.0

Apr 17, 2017

Contents

CHAPTER 1

Installation

Packages

Mac OS X

If you are using Mac OS X, you can install restic using the homebrew packet manager:

```
$ brew tap restic/restic
$ brew install restic
```

archlinux

On archlinux, there is a package called `restic-git` which can be installed from AUR, e.g. with `pacaur`:

```
$ pacaur -S restic-git
```

Pre-compiled Binary

You can download the latest pre-compiled binary from the restic release page.

From Source

restic is written in the Go programming language and you need at least Go version 1.7. Building restic may also work with older versions of Go, but that's not supported. See the Getting started guide of the Go project for instructions how to install Go.

In order to build restic from source, execute the following steps:

```
$ git clone https://github.com/restic/restic
[...]

$ cd restic

$ go run build.go
```

You can easily cross-compile restic for all supported platforms, just supply the target OS and platform via the command-line options like this (for Windows and FreeBSD respectively):

```
$ go run build.go --goos windows --goarch amd64

$ go run build.go --goos freebsd --goarch 386
```

The resulting binary is statically linked and does not require any libraries.

At the moment, the only tested compiler for restic is the official Go compiler. Building restic with gccgo may work, but is not supported.

CHAPTER 2

Manual

Usage help

Usage help is available:

```
$ ./restic --help
restic is a backup program which allows saving multiple revisions of files and
directories in an encrypted repository stored on different backends.

Usage:
  restic [command]

Available Commands:
  backup        create a new backup of files and/or directories
  cat           print internal objects to stdout
  check         check the repository for errors
  find          find a file or directory
  forget        forget removes snapshots from the repository
  init          initialize a new repository
  key           manage keys (passwords)
  list          list items in the repository
  ls            list files in a snapshot
  mount         mount the repository
  prune         remove unneeded data from the repository
  rebuild-index build a new index file
  restore       extract the data from a snapshot
  snapshots     list all snapshots
  tag           modifies tags on snapshots
  unlock        remove locks other processes created
  version       Print version information

Flags:
      --json                   set output mode to JSON for commands that support it
      --no-lock                do not lock the repo, this allows some operations on␣
↪read-only repos
```

```
  -p, --password-file string   read the repository password from a file
  -q, --quiet                  do not output comprehensive progress report
  -r, --repo string            repository to backup to or restore from (default:
↪$RESTIC_REPOSITORY)

Use "restic [command] --help" for more information about a command.
```

Similar to programs such as `git`, restic has a number of sub-commands. You can see these commands in the listing above. Each sub-command may have own command-line options, and there is a help option for each command which lists them, e.g. for the `backup` command:

```
$ ./restic backup --help
The "backup" command creates a new snapshot and saves the files and directories
given as the arguments.

Usage:
  restic backup [flags] FILE/DIR [FILE/DIR] ...

Flags:
  -e, --exclude pattern         exclude a pattern (can be specified multiple times)
      --exclude-file string     read exclude patterns from a file
      --files-from string       read the files to backup from file (can be combined␣
↪with file args)
  -f, --force                   force re-reading the target files/directories.␣
↪Overrides the "parent" flag
  -x, --one-file-system         Exclude other file systems
      --parent string           use this parent snapshot (default: last snapshot in␣
↪the repo that has the same target files/directories)
      --stdin                   read backup from stdin
      --stdin-filename string   file name to use when reading from stdin
      --tag tag                 add a tag for the new snapshot (can be specified␣
↪multiple times)

Global Flags:
      --json                   set output mode to JSON for commands that support it
      --no-lock                do not lock the repo, this allows some operations on␣
↪read-only repos
  -p, --password-file string   read the repository password from a file
  -q, --quiet                  do not output comprehensive progress report
  -r, --repo string            repository to backup to or restore from (default:
↪$RESTIC_REPOSITORY)
```

Subcommand that support showing progress information such as `backup`, `check` and `prune` will do so unless the quiet flag `-q` or `--quiet` is set. When running from a non-interactive console progress reporting will be limited to once every 10 seconds to not fill your logs.

Additionally on Unix systems if `restic` receives a SIGUSR signal the current progress will written to the standard output so you can check up on the status at will.

Initialize a repository

First, we need to create a "repository". This is the place where your backups will be saved at.

Local

In order to create a repository at `/tmp/backup`, run the following command and enter the same password twice:

```
$ restic init --repo /tmp/backup
enter password for new backend:
enter password again:
created restic backend 085b3c76b9 at /tmp/backup
Please note that knowledge of your password is required to access the repository.
Losing your password means that your data is irrecoverably lost.
```

Other backends like sftp and s3 are *described in a later section* of this document.

Remembering your password is important! If you lose it, you won't be able to access data stored in the repository.

For automated backups, restic accepts the repository location in the environment variable `RESTIC_REPOSITORY`. The password can be read from a file (via the option `--password-file`) or the environment variable `RESTIC_PASSWORD`.

SFTP

In order to backup data via SFTP, you must first set up a server with SSH and let it know your public key. Passwordless login is really important since restic fails to connect to the repository if the server prompts for credentials.

Once the server is configured, the setup of the SFTP repository can simply be achieved by changing the URL scheme in the `init` command:

```
$ restic -r sftp:user@host:/tmp/backup init
enter password for new backend:
enter password again:
created restic backend f1c6108821 at sftp:user@host:/tmp/backup
Please note that knowledge of your password is required to access the repository.
Losing your password means that your data is irrecoverably lost.
```

You can also specify a relative (read: no slash (`/`) character at the beginning) directory, in this case the dir is relative to the remote user's home directory.

The backend config string does not allow specifying a port. If you need to contact an sftp server on a different port, you can create an entry in the `ssh` file, usually located in your user's home directory at `~/.ssh/config` or in `/etc/ssh/ssh_config`:

```
Host foo
    User bar
    Port 2222
```

Then use the specified host name `foo` normally (you don't need to specify the user name in this case):

```
$ restic -r sftp:foo:/tmp/backup init
```

You can also add an entry with a special host name which does not exist, just for use with restic, and use the `Hostname` option to set the real host name:

```
Host restic-backup-host
    Hostname foo
    User bar
    Port 2222
```

Then use it in the backend specification:

```
$ restic -r sftp:restic-backup-host:/tmp/backup init
```

Last, if you'd like to use an entirely different program to create the SFTP connection, you can specify the command to be run with the option `-o sftp.command="foobar"`.

REST Server

In order to backup data to the remote server via HTTP or HTTPS protocol, you must first set up a remote REST server instance. Once the server is configured, accessing it is achieved by changing the URL scheme like this:

```
$ restic -r rest:http://host:8000/
```

Depending on your REST server setup, you can use HTTPS protocol, password protection, or multiple repositories. Or any combination of those features, as you see fit. TCP/IP port is also configurable. Here are some more examples:

```
$ restic -r rest:https://host:8000/
$ restic -r rest:https://user:pass@host:8000/
$ restic -r rest:https://user:pass@host:8000/my_backup_repo/
```

If you use TLS, make sure your certificates are signed, 'cause restic client will refuse to communicate otherwise. It's easy to obtain such certificates today, thanks to free certificate authorities like Let's Encrypt.

REST server uses exactly the same directory structure as local backend, so you should be able to access it both locally and via HTTP, even simultaneously.

Amazon S3

Restic can backup data to any Amazon S3 bucket. However, in this case, changing the URL scheme is not enough since Amazon uses special security credentials to sign HTTP requests. By consequence, you must first setup the following environment variables with the credentials you obtained while creating the bucket.

```
$ export AWS_ACCESS_KEY_ID=<MY_ACCESS_KEY>
$ export AWS_SECRET_ACCESS_KEY=<MY_SECRET_ACCESS_KEY>
```

You can then easily initialize a repository that uses your Amazon S3 as a backend, if the bucket does not exist yet it will be created in the default location:

```
$ restic -r s3:s3.amazonaws.com/bucket_name init
enter password for new backend:
enter password again:
created restic backend eefee03bbd at s3:s3.amazonaws.com/bucket_name
Please note that knowledge of your password is required to access the repository.
Losing your password means that your data is irrecoverably lost.
```

It is not possible at the moment to have restic create a new bucket in a different location, so you need to create it using a different program. Afterwards, the S3 server (`s3.amazonaws.com`) will redirect restic to the correct endpoint.

For an S3-compatible server that is not Amazon (like Minio, see below), or is only available via HTTP, you can specify the URL to the server like this: `s3:http://server:port/bucket_name`.

Minio Server

Minio is an Open Source Object Storage, written in Go and compatible with AWS S3 API.

- Download and Install Minio Server.
- You can also refer to https://docs.minio.io for step by step guidance on installation and getting started on Minio Client and Minio Server.

You must first setup the following environment variables with the credentials of your running Minio Server.

```
$ export AWS_ACCESS_KEY_ID=<YOUR-MINIO-ACCESS-KEY-ID>
$ export AWS_SECRET_ACCESS_KEY= <YOUR-MINIO-SECRET-ACCESS-KEY>
```

Now you can easily initialize restic to use Minio server as backend with this command.

```
$ ./restic -r s3:http://localhost:9000/restic init
enter password for new backend:
enter password again:
created restic backend 6ad29560f5 at s3:http://localhost:9000/restic1
Please note that knowledge of your password is required to access
the repository. Losing your password means that your data is irrecoverably lost.
```

Password prompt on Windows

At the moment, restic only supports the default Windows console interaction. If you use emulation environments like MSYS2 or Cygwin, which use terminals like `Mintty` or `rxvt`, you may get a password error:

You can workaround this by using a special tool called `winpty` (look here and here for detail information). On MSYS2, you can install `winpty` as follows:

```
$ pacman -S winpty
$ winpty restic -r /tmp/backup init
```

Create a snapshot

Now we're ready to backup some data. The contents of a directory at a specific point in time is called a "snapshot" in restic. Run the following command and enter the repository password you chose above again:

```
$ restic -r /tmp/backup backup ~/work
enter password for repository:
scan [/home/user/work]
scanned 764 directories, 1816 files in 0:00
[0:29] 100.00%  54.732 MiB/s  1.582 GiB / 1.582 GiB  2580 / 2580 items  0 errors  ETA␣
↪0:00
duration: 0:29, 54.47MiB/s
snapshot 40dc1520 saved
```

As you can see, restic created a backup of the directory and was pretty fast! The specific snapshot just created is identified by a sequence of hexadecimal characters, `40dc1520` in this case.

If you run the command again, restic will create another snapshot of your data, but this time it's even faster. This is de-duplication at work!

```
$ restic -r /tmp/backup backup ~/shared/work/web
enter password for repository:
using parent snapshot 40dc1520aa6a07b7b3ae561786770a01951245d2367241e71e9485f18ae8228c
scan [/home/user/work]
scanned 764 directories, 1816 files in 0:00
```

```
[0:00] 100.00%  0B/s  1.582 GiB / 1.582 GiB  2580 / 2580 items  0 errors  ETA 0:00
duration: 0:00, 6572.38MiB/s
snapshot 79766175 saved
```

You can even backup individual files in the same repository.

```
$ restic -r /tmp/backup backup ~/work.txt
scan [~/work.txt]
scanned 0 directories, 1 files in 0:00
[0:00] 100.00%  0B/s  220B / 220B  1 / 1 items  0 errors  ETA 0:00
duration: 0:00, 0.03MiB/s
snapshot 31f7bd63 saved
```

In fact several hosts may use the same repository to backup directories and files leading to a greater de-duplication.

Please be aware that when you backup different directories (or the directories to be saved have a variable name component like a time/date), restic always needs to read all files and only afterwards can compute which parts of the files need to be saved. When you backup the same directory again (maybe with new or changed files) restic will find the old snapshot in the repo and by default only reads those files that are new or have been modified since the last snapshot. This is decided based on the modify date of the file in the file system.

You can exclude folders and files by specifying exclude-patterns. Either specify them with multiple `--exclude`s or one `--exclude-file`

```
$ cat exclude
# exclude go-files
*.go
# exclude foo/x/y/z/bar foo/x/bar foo/bar
foo/**/bar
$ restic -r /tmp/backup backup ~/work --exclude=*.c --exclude-file=exclude
```

Patterns use `` `filepath.Glob <https://golang.org/pkg/path/filepath/#Glob>`__ `` internally, see `` `filepath.Match <https://golang.org/pkg/path/filepath/#Match>`__ `` for syntax. Additionally `**` excludes arbitrary subdirectories. Environment-variables in exclude-files are expanded with `` `os.ExpandEnv <https://golang.org/pkg/os/#ExpandEnv>`__ ``.

By specifying the option `--one-file-system` you can instruct restic to only backup files from the file systems the initially specified files or directories reside on. For example, calling restic like this won't backup `/sys` or `/dev` on a Linux system:

```
$ restic -r /tmp/backup backup --one-file-system /
```

By using the `--files-from` option you can read the files you want to backup from a file. This is especially useful if a lot of files have to be backed up that are not in the same folder or are maybe pre-filtered by other software.

For example maybe you want to backup files that have a certain filename in them:

```
$ find /tmp/somefiles | grep 'PATTERN' > /tmp/files_to_backup
```

You can then use restic to backup the filtered files:

```
$ restic -r /tmp/backup backup --files-from /tmp/files_to_backup
```

Incidentally you can also combine `--files-from` with the normal files args:

```
$ restic -r /tmp/backup backup --files-from /tmp/files_to_backup /tmp/some_additional_
↪file
```

Reading data from stdin

Sometimes it can be nice to directly save the output of a program, e.g. `mysqldump` so that the SQL can later be restored. Restic supports this mode of operation, just supply the option `--stdin` to the `backup` command like this:

```
$ mysqldump [...] | restic -r /tmp/backup backup --stdin
```

This creates a new snapshot of the output of `mysqldump`. You can then use e.g. the fuse mounting option (see below) to mount the repository and read the file.

By default, the file name `stdin` is used, a different name can be specified with `--stdin-filename`, e.g. like this:

```
$ mysqldump [...] | restic -r /tmp/backup backup --stdin --stdin-filename production.
↪sql
```

Tags

Snapshots can have one or more tags, short strings which add identifying information. Just specify the tags for a snapshot with `--tag`:

```
$ restic -r /tmp/backup backup --tag projectX ~/shared/work/web
[...]
```

The tags can later be used to keep (or forget) snapshots.

List all snapshots

Now, you can list all the snapshots stored in the repository:

```
$ restic -r /tmp/backup snapshots
enter password for repository:
ID        Date                 Host    Tags   Directory
----------------------------------------------------------------------
40dc1520  2015-05-08 21:38:30  kasimir        /home/user/work
79766175  2015-05-08 21:40:19  kasimir        /home/user/work
bdbd3439  2015-05-08 21:45:17  luigi          /home/art
590c8fc8  2015-05-08 21:47:38  kazik          /srv
9f0bc19e  2015-05-08 21:46:11  luigi          /srv
```

You can filter the listing by directory path:

```
$ restic -r /tmp/backup snapshots --path="/srv"
enter password for repository:
ID        Date                 Host    Tags   Directory
----------------------------------------------------------------------
590c8fc8  2015-05-08 21:47:38  kazik          /srv
9f0bc19e  2015-05-08 21:46:11  luigi          /srv
```

Or filter by host:

```
$ restic -r /tmp/backup snapshots --host luigi
enter password for repository:
ID        Date                 Host    Tags   Directory
----------------------------------------------------------------------
```

```
bdbd3439  2015-05-08 21:45:17  luigi          /home/art
9f0bc19e  2015-05-08 21:46:11  luigi          /srv
```

Combining filters is also possible.

Restore a snapshot

Restoring a snapshot is as easy as it sounds, just use the following command to restore the contents of the latest snapshot to `/tmp/restore-work`:

```
$ restic -r /tmp/backup restore 79766175 --target ~/tmp/restore-work
enter password for repository:
restoring <Snapshot of [/home/user/work] at 2015-05-08 21:40:19.884408621 +0200 CEST>␣
↪to /tmp/restore-work
```

Use the word `latest` to restore the last backup. You can also combine `latest` with the `--host` and `--path` filters to choose the last backup for a specific host, path or both.

```
$ restic -r /tmp/backup restore latest --target ~/tmp/restore-work --path "/home/art"␣
↪--host luigi
enter password for repository:
restoring <Snapshot of [/home/art] at 2015-05-08 21:45:17.884408621 +0200 CEST> to /
↪tmp/restore-work
```

Manage repository keys

The `key` command allows you to set multiple access keys or passwords per repository. In fact, you can use the `list`, `add`, `remove` and `passwd` sub-commands to manage these keys very precisely:

```
$ restic -r /tmp/backup key list
enter password for repository:
 ID          User        Host        Created
----------------------------------------------------------------------
*eb78040b    username    kasimir   2015-08-12 13:29:57

$ restic -r /tmp/backup key add
enter password for repository:
enter password for new key:
enter password again:
saved new key as <Key of username@kasimir, created on 2015-08-12 13:35:05.316831933␣
↪+0200 CEST>

$ restic -r backup key list
enter password for repository:
 ID          User        Host        Created
----------------------------------------------------------------------
 5c657874    username    kasimir   2015-08-12 13:35:05
*eb78040b    username    kasimir   2015-08-12 13:29:57
```

Manage tags

Managing tags on snapshots is done with the `tag` command. The existing set of tags can be replaced completely, tags can be added to removed. The result is directly visible in the `snapshots` command.

Let's say we want to tag snapshot `590c8fc8` with the tags `NL` and `CH` and remove all other tags that may be present, the following command does that:

```
$ restic -r /tmp/backup tag --set NL,CH 590c8fc8
Create exclusive lock for repository
Modified tags on 1 snapshots
```

Note the snapshot ID has changed, so between each change we need to look up the new ID of the snapshot. But there is an even better way, the `tag` command accepts `--tag` for a filter, so we can filter snapshots based on the tag we just added.

So we can add and remove tags incrementally like this:

```
$ restic -r /tmp/backup tag --tag NL --remove CH
Create exclusive lock for repository
Modified tags on 1 snapshots

$ restic -r /tmp/backup tag --tag NL --add UK
Create exclusive lock for repository
Modified tags on 1 snapshots

$ restic -r /tmp/backup tag --tag NL --remove NL
Create exclusive lock for repository
Modified tags on 1 snapshots

$ restic -r /tmp/backup tag --tag NL --add SOMETHING
No snapshots were modified
```

Check integrity and consistency

Imagine your repository is saved on a server that has a faulty hard drive, or even worse, attackers get privileged access and modify your backup with the intention to make you restore malicious data:

```
$ sudo echo "boom" >> backup/index/
↪d795ffa99a8ab8f8e42cec1f814df4e48b8f49129360fb57613df93739faee97
```

In order to detect these things, it is a good idea to regularly use the `check` command to test whether everything is alright, your precious backup data is consistent and the integrity is unharmed:

```
$ restic -r /tmp/backup check
Load indexes
ciphertext verification failed
```

Trying to restore a snapshot which has been modified as shown above will yield the same error:

```
$ restic -r /tmp/backup restore 79766175 --target ~/tmp/restore-work
Load indexes
ciphertext verification failed
```

Mount a repository

Browsing your backup as a regular file system is also very easy. First, create a mount point such as `/mnt/restic` and then use the following command to serve the repository with FUSE:

```
$ mkdir /mnt/restic
$ restic -r /tmp/backup mount /mnt/restic
enter password for repository:
Now serving /tmp/backup at /tmp/restic
Don't forget to umount after quitting!
```

Mounting repositories via FUSE is not possible on Windows and OpenBSD.

Restic supports storage and preservation of hard links. However, since hard links exist in the scope of a filesystem by definition, restoring hard links from a fuse mount should be done by a program that preserves hard links. A program that does so is rsync, used with the option –hard-links.

Removing old snapshots

All backup space is finite, so restic allows removing old snapshots. This can be done either manually (by specifying a snapshot ID to remove) or by using a policy that describes which snapshots to forget. For all remove operations, two commands need to be called in sequence: `forget` to remove a snapshot and `prune` to actually remove the data that was referenced by the snapshot from the repository. This can be automated with the `--prune` option of the `forget` command, which runs `prune` automatically if snapshots have been removed.

Remove a single snapshot

The command `snapshots` can be used to list all snapshots in a repository like this:

```
$ restic -r /tmp/backup snapshots
enter password for repository:
ID        Date                 Host      Tags  Directory
----------------------------------------------------------------------
40dc1520  2015-05-08 21:38:30  kasimir         /home/user/work
79766175  2015-05-08 21:40:19  kasimir         /home/user/work
bdbd3439  2015-05-08 21:45:17  luigi           /home/art
590c8fc8  2015-05-08 21:47:38  kazik           /srv
9f0bc19e  2015-05-08 21:46:11  luigi           /srv
```

In order to remove the snapshot of `/home/art`, use the `forget` command and specify the snapshot ID on the command line:

```
$ restic -r /tmp/backup forget bdbd3439
enter password for repository:
removed snapshot d3f01f63
```

Afterwards this snapshot is removed:

```
$ restic -r /tmp/backup snapshots
enter password for repository:
ID        Date                 Host     Tags  Directory
----------------------------------------------------------------------
40dc1520  2015-05-08 21:38:30  kasimir        /home/user/work
79766175  2015-05-08 21:40:19  kasimir        /home/user/work
```

```
590c8fc8  2015-05-08 21:47:38  kazik          /srv
9f0bc19e  2015-05-08 21:46:11  luigi          /srv
```

But the data that was referenced by files in this snapshot is still stored in the repository. To cleanup unreferenced data, the `prune` command must be run:

```
$ restic -r /tmp/backup prune
enter password for repository:

counting files in repo
building new index for repo
[0:00] 100.00%  22 / 22 files
repository contains 22 packs (8512 blobs) with 100.092 MiB bytes
processed 8512 blobs: 0 duplicate blobs, 0B duplicate
load all snapshots
find data that is still in use for 1 snapshots
[0:00] 100.00%  1 / 1 snapshots
found 8433 of 8512 data blobs still in use
will rewrite 3 packs
creating new index
[0:00] 86.36%  19 / 22 files
saved new index as 544a5084
done
```

Afterwards the repository is smaller.

You can automate this two-step process by using the `--prune` switch to `forget`:

```
$ restic forget --keep-last 1 --prune
snapshots for host mopped, directories /home/user/work:

keep 1 snapshots:
ID        Date                 Host        Tags        Directory
----------------------------------------------------------------------
4bba301e  2017-02-21 10:49:18  mopped                  /home/user/work

remove 1 snapshots:
ID        Date                 Host        Tags        Directory
----------------------------------------------------------------------
8c02b94b  2017-02-21 10:48:33  mopped                  /home/user/work

1 snapshots have been removed, running prune
counting files in repo
building new index for repo
[0:00] 100.00%  37 / 37 packs
repository contains 37 packs (5521 blobs) with 151.012 MiB bytes
processed 5521 blobs: 0 duplicate blobs, 0B duplicate
load all snapshots
find data that is still in use for 1 snapshots
[0:00] 100.00%  1 / 1 snapshots
found 5323 of 5521 data blobs still in use, removing 198 blobs
will delete 0 packs and rewrite 27 packs, this frees 22.106 MiB
creating new index
[0:00] 100.00%  30 / 30 packs
saved new index as b49f3e68
done
```

Removing snapshots according to a policy

Removing snapshots manually is tedious and error-prone, therefore restic allows specifying which snapshots should be removed automatically according to a policy. You can specify how many hourly, daily, weekly, monthly and yearly snapshots to keep, any other snapshots are removed. The most important command-line parameter here is `--dry-run` which instructs restic to not remove anything but print which snapshots would be removed.

When `forget` is run with a policy, restic loads the list of all snapshots, then groups these by host name and list of directories. The policy is then applied to each group of snapshots separately. This is a safety feature.

The `forget` command accepts the following parameters:

- `--keep-last n` never delete the `n` last (most recent) snapshots
- `--keep-hourly n` for the last `n` hours in which a snapshot was made, keep only the last snapshot for each hour.
- `--keep-daily n` for the last `n` days which have one or more snapshots, only keep the last one for that day.
- `--keep-weekly n` for the last `n` weeks which have one or more snapshots, only keep the last one for that week.
- `--keep-monthly n` for the last `n` months which have one or more snapshots, only keep the last one for that month.
- `--keep-yearly n` for the last `n` years which have one or more snapshots, only keep the last one for that year.
- `--keep-tag` keep all snapshots which have all tags specified by this option (can be specified multiple times).

Additionally, you can restrict removing snapshots to those which have a particular hostname with the `--hostname` parameter, or tags with the `--tag` option. When multiple tags are specified, only the snapshots which have all the tags are considered.

All the `--keep-*` options above only count hours/days/weeks/months/years which have a snapshot, so those without a snapshot are ignored.

Let's explain this with an example: Suppose you have only made a backup on each Sunday for 12 weeks. Then `forget --keep-daily 4` will keep the last four snapshots for the last four Sundays, but remove the rest. Only counting the days which have a backup and ignore the ones without is a safety feature: it prevents restic from removing many snapshots when no new ones are created. If it was implemented otherwise, running `forget --keep-daily 4` on a Friday would remove all snapshots!

Debugging

The program can be built with debug support like this:

```
$ go run build.go -tags debug
```

Afterwards, extensive debug messages are written to the file in environment variable `DEBUG_LOG`, e.g.:

```
$ DEBUG_LOG=/tmp/restic-debug.log restic backup ~/work
```

If you suspect that there is a bug, you can have a look at the debug log. Please be aware that the debug log might contain sensitive information such as file and directory names.

The debug log will always contain all log messages restic generates. You can also instruct restic to print some or all debug messages to stderr. These can also be limited to e.g. a list of source files or a list of patterns for function

names. The patterns are globbing patterns (see the documentation for `path.Glob <https://golang.org/pkg/path/#Glob>`__), multiple patterns are separated by commas. Patterns are case sensitive.

Printing all log messages to the console can be achieved by setting the file filter to `*`:

```
$ DEBUG_FILES=* restic check
```

If you want restic to just print all debug log messages from the files `main.go` and `lock.go`, set the environment variable `DEBUG_FILES` like this:

```
$ DEBUG_FILES=main.go,lock.go restic check
```

The following command line instructs restic to only print debug statements originating in functions that match the pattern `*unlock*` (case sensitive):

```
$ DEBUG_FUNCS=*unlock* restic check
```

Under the hood: Browse repository objects

Internally, a repository stores data of several different types described in the design documentation. You can `list` objects such as blobs, packs, index, snapshots, keys or locks with the following command:

```
$ restic -r /tmp/backup list snapshots
d369ccc7d126594950bf74f0a348d5d98d9e99f3215082eb69bf02dc9b3e464c
```

The `find` command searches for a given pattern in the repository.

```
$ restic -r backup find test.txt
debug log file restic.log
debug enabled
enter password for repository:
found 1 matching entries in snapshot␣
↪196bc5760c909a7681647949e80e5448e276521489558525680acf1bd428af36
  -rw-r--r--   501    20      5 2015-08-26 14:09:57 +0200 CEST path/to/test.txt
```

The `cat` command allows you to display the JSON representation of the objects or its raw content.

```
$ restic -r /tmp/backup cat snapshot␣
↪d369ccc7d126594950bf74f0a348d5d98d9e99f3215082eb69bf02dc9b3e464c
enter password for repository:
{
  "time": "2015-08-12T12:52:44.091448856+02:00",
  "tree": "05cec17e8d3349f402576d02576a2971fc0d9f9776ce2f441c7010849c4ff5af",
  "paths": [
    "/home/user/work"
  ],
  "hostname": "kasimir",
  "username": "username",
  "uid": 501,
  "gid": 20
}
```

Scripting

Restic supports the output of some commands in JSON format, the JSON data can then be processed by other programs (e.g. jq). The following example lists all snapshots as JSON and uses `jq` to pretty-print the result:

```
$ restic -r /tmp/backup snapshots --json | jq .
[
  {
    "time": "2017-03-11T09:57:43.26630619+01:00",
    "tree": "bf25241679533df554fc0fd0ae6dbb9dcf1859a13f2bc9dd4543c354eff6c464",
    "paths": [
      "/home/work/doc"
    ],
    "hostname": "kasimir",
    "username": "fd0",
    "uid": 1000,
    "gid": 100,
    "id": "bbeed6d28159aa384d1ccc6fa0b540644b1b9599b162d2972acda86b1b80f89e"
  },
  {
    "time": "2017-03-11T09:58:57.541446938+01:00",
    "tree": "7f8c95d3420baaac28dc51609796ae0e0ecfb4862b609a9f38ffaf7ae2d758da",
    "paths": [
      "/home/user/shared"
    ],
    "hostname": "kasimir",
    "username": "fd0",
    "uid": 1000,
    "gid": 100,
    "id": "b157d91c16f0ba56801ece3a708dfc53791fe2a97e827090d6ed9a69a6ebdca0"
  }
]
```

Temporary files

During some operations (e.g. `backup` and `prune`) restic uses temporary files to store data. These files will, by default, be saved to the system's temporary directory, on Linux this is usually located in `/tmp/`. The environment variable `TMPDIR` can be used to specify a different directory, e.g. to use the directory `/var/tmp/restic-tmp` instead of the default, set the environment variable like this:

```
$ export TMPDIR=/var/tmp/restic-tmp
$ restic -r /tmp/backup backup ~/work
```

CHAPTER 3

FAQ

This is the list of Frequently Asked Questions for restic.

`restic check` reports packs that aren't referenced in any index, is my repository broken?

When `restic check` reports that there are pack files in the repository that are not referenced in any index, that's (in contrast to what restic reports at the moment) not a source for concern. The output looks like this:

```
$ restic check
Create exclusive lock for repository
Load indexes
Check all packs
pack 819a9a52e4f51230afa89aefbf90df37fb70996337ae57e6f7a822959206a85e: not referenced␣
↪in any index
pack de299e69fb075354a3775b6b045d152387201f1cdc229c31d1caa34c3b340141: not referenced␣
↪in any index
Check snapshots, trees and blobs
Fatal: repository contains errors
```

The message means that there is more data stored in the repo than strictly necessary. With high probability this is duplicate data. In order to clean it up, the command `restic prune` can be used. The cause of this bug is not yet known.

CHAPTER 4

Development

Contribute

Contributions are welcome! Please **open an issue first** (or add a comment to an existing issue) if you plan to work on any code or add a new feature. This way, duplicate work is prevented and we can discuss your ideas and design first.

More information and a description of the development environment can be found in CONTRIBUTING.md. A document describing the design of restic and the data structures stored on the back end is contained in Design.

If you'd like to start contributing to restic, but don't know exactly what do to, have a look at this great article by Dave Cheney: Suggestions for contributing to an Open Source project A few issues have been tagged with the label `help wanted`, you can start looking at those: https://github.com/restic/restic/labels/help%20wanted

Security

Important: If you discover something that you believe to be a possible critical security problem, please do *not* open a GitHub issue but send an email directly to alexander@bumpern.de. If possible, please encrypt your email using the following PGP key (0x91A6868BD3F7A907):

```
pub   4096R/91A6868BD3F7A907 2014-11-01
      Key fingerprint = CF8F 18F2 8445 7597 3F79  D4E1 91A6 868B D3F7 A907
      uid                          Alexander Neumann <alexander@bumpern.de>
      sub   4096R/D5FC2ACF4043FDF1 2014-11-01
```

Compatibility

Backward compatibility for backups is important so that our users are always able to restore saved data. Therefore restic follows Semantic Versioning to clearly define which versions are compatible. The repository and data structures contained therein are considered the "Public API" in the sense of Semantic Versioning. This goes for all released versions of restic, this may not be the case for the master branch.

We guarantee backward compatibility of all repositories within one major version; as long as we do not increment the major version, data can be read and restored. We strive to be fully backward compatible to all prior versions.

Building documentation

The restic documentation is built with Sphinx, therefore building it locally requires a recent Python version and requirements listed in `doc/requirements.txt`. This example will guide you through the process using virtualenv:

```
$ virtualenv venv # create virtual python environment
$ source venv/bin/activate # activate the virtual environment
$ cd doc
$ pip install -r requirements.txt # install dependencies
$ make html # build html documentation
$ # open _build/html/index.html with your favorite browser
```

CHAPTER 5

References

Design

Terminology

This section introduces terminology used in this document.

Repository: All data produced during a backup is sent to and stored in a repository in a structured form, for example in a file system hierarchy with several subdirectories. A repository implementation must be able to fulfill a number of operations, e.g. list the contents.

Blob: A Blob combines a number of data bytes with identifying information like the SHA-256 hash of the data and its length.

Pack: A Pack combines one or more Blobs, e.g. in a single file.

Snapshot: A Snapshot stands for the state of a file or directory that has been backed up at some point in time. The state here means the content and meta data like the name and modification time for the file or the directory and its contents.

Storage ID: A storage ID is the SHA-256 hash of the content stored in the repository. This ID is required in order to load the file from the repository.

Repository Format

All data is stored in a restic repository. A repository is able to store data of several different types, which can later be requested based on an ID. This so-called "storage ID" is the SHA-256 hash of the content of a file. All files in a repository are only written once and never modified afterwards. This allows accessing and even writing to the repository with multiple clients in parallel. Only the delete operation removes data from the repository.

Repositories consist of several directories and a top-level file called `config`. For all other files stored in the repository, the name for the file is the lower case hexadecimal representation of the storage ID, which is the SHA-256 hash of the file's contents. This allows for easy verification of files for accidental modifications, like disk read errors, by simply running the program `sha256sum` on the file and comparing its output to the file name. If the prefix of a filename is unique amongst all the other files in the same directory, the prefix may be used instead of the complete filename.

Apart from the files stored within the `keys` directory, all files are encrypted with AES-256 in counter mode (CTR). The integrity of the encrypted data is secured by a Poly1305-AES message authentication code (sometimes also referred to as a "signature").

In the first 16 bytes of each encrypted file the initialisation vector (IV) is stored. It is followed by the encrypted data and completed by the 16 byte MAC. The format is: `IV || CIPHERTEXT || MAC`. The complete encryption overhead is 32 bytes. For each file, a new random IV is selected.

The file `config` is encrypted this way and contains a JSON document like the following:

```
{
  "version": 1,
  "id": "5956a3f67a6230d4a92cefb29529f10196c7d92582ec305fd71ff6d331d6271b",
  "chunker_polynomial": "25b468838dcb75"
}
```

After decryption, restic first checks that the version field contains a version number that it understands, otherwise it aborts. At the moment, the version is expected to be 1. The field `id` holds a unique ID which consists of 32 random bytes, encoded in hexadecimal. This uniquely identifies the repository, regardless if it is accessed via SFTP or locally. The field `chunker_polynomial` contains a parameter that is used for splitting large files into smaller chunks (see below).

Filesystem-Based Repositories

The `local` and `sftp` backends are implemented using files and directories stored in a file system. The directory layout is the same for both backend types.

The basic layout of a repository stored in a `local` or `sftp` backend is shown here:

```
/tmp/restic-repo
- config
- data
|   - 21
|   |   - 2159dd48f8a24f33c307b750592773f8b71ff8d11452132a7b2e2a6a01611be1
|   - 32
|   |   - 32ea976bc30771cebad8285cd99120ac8786f9ffd42141d452458089985043a5
|   - 59
|   |   - 59fe4bcde59bd6222eba87795e35a90d82cd2f138a27b6835032b7b58173a426
|   - 73
|   |   - 73d04e6125cf3c28a299cc2f3cca3b78ceac396e4fcf9575e34536b26782413c
|   [...]
- index
|   - c38f5fb68307c6a3e3aa945d556e325dc38f5fb68307c6a3e3aa945d556e325d
|   - ca171b1b7394d90d330b265d90f506f9984043b342525f019788f97e745c71fd
- keys
|   - b02de829beeb3c01a63e6b25cbd421a98fef144f03b9a02e46eff9e2ca3f0bd7
- locks
- snapshots
|   - 22a5af1bdc6e616f8a29579458c49627e01b32210d09adb288d1ecda7c5711ec
- tmp
```

A local repository can be initialized with the `restic init` command, e.g.:

```
$ restic -r /tmp/restic-repo init
```

The local and sftp backends will also accept the repository layout described in the following section, so that remote repositories mounted locally e.g. via fuse can be accessed. The layout auto-detection can be overridden by specifying

the option `-o local.layout=default`, valid values are `default`, `cloud` and `s3`. The option for the sftp backend is named `sftp.layout`.

Object-Storage-Based Repositories

Repositories in a backend based on an object store (e.g. Amazon s3) have the same basic layout, with the exception that all data pack files are directly saved in the `data` path, without the sub-directories listed for the filesystem-based backends as listed in the previous section. The layout looks like this:

```
/config
/data
 - 2159dd48f8a24f33c307b750592773f8b71ff8d11452132a7b2e2a6a01611be1
 - 32ea976bc30771cebad8285cd99120ac8786f9ffd42141d452458089985043a5
 - 59fe4bcde59bd6222eba87795e35a90d82cd2f138a27b6835032b7b58173a426
 - 73d04e6125cf3c28a299cc2f3cca3b78ceac396e4fcf9575e34536b26782413c
[...]
/index
 - c38f5fb68307c6a3e3aa945d556e325dc38f5fb68307c6a3e3aa945d556e325d
 - ca171b1b7394d90d330b265d90f506f9984043b342525f019788f97e745c71fd
/keys
 - b02de829beeb3c01a63e6b25cbd421a98fef144f03b9a02e46eff9e2ca3f0bd7
/locks
/snapshots
 - 22a5af1bdc6e616f8a29579458c49627e01b32210d09adb288d1ecda7c5711ec
```

Unfortunately during development the s3 backend uses slightly different paths (directory names use singular instead of plural for `key`, `lock`, and `snapshot` files), for s3 the repository layout looks like this:

```
/config
/data
 - 2159dd48f8a24f33c307b750592773f8b71ff8d11452132a7b2e2a6a01611be1
 - 32ea976bc30771cebad8285cd99120ac8786f9ffd42141d452458089985043a5
 - 59fe4bcde59bd6222eba87795e35a90d82cd2f138a27b6835032b7b58173a426
 - 73d04e6125cf3c28a299cc2f3cca3b78ceac396e4fcf9575e34536b26782413c
[...]
/index
 - c38f5fb68307c6a3e3aa945d556e325dc38f5fb68307c6a3e3aa945d556e325d
 - ca171b1b7394d90d330b265d90f506f9984043b342525f019788f97e745c71fd
/key
 - b02de829beeb3c01a63e6b25cbd421a98fef144f03b9a02e46eff9e2ca3f0bd7
/lock
/snapshot
 - 22a5af1bdc6e616f8a29579458c49627e01b32210d09adb288d1ecda7c5711ec
```

The s3 backend understands and accepts both forms, new backends are always created with the former layout for compatibility reasons.

Pack Format

All files in the repository except Key and Pack files just contain raw data, stored as `IV || Ciphertext || MAC`. Pack files may contain one or more Blobs of data.

A Pack's structure is as follows:

```
EncryptedBlob1 || ... || EncryptedBlobN || EncryptedHeader || Header_Length
```

At the end of the Pack file is a header, which describes the content. The header is encrypted and authenticated. `Header_Length` is the length of the encrypted header encoded as a four byte integer in little-endian encoding. Placing the header at the end of a file allows writing the blobs in a continuous stream as soon as they are read during the backup phase. This reduces code complexity and avoids having to re-write a file once the pack is complete and the content and length of the header is known.

All the blobs (`EncryptedBlob1`, `EncryptedBlobN` etc.) are authenticated and encrypted independently. This enables repository reorganisation without having to touch the encrypted Blobs. In addition it also allows efficient indexing, for only the header needs to be read in order to find out which Blobs are contained in the Pack. Since the header is authenticated, authenticity of the header can be checked without having to read the complete Pack.

After decryption, a Pack's header consists of the following elements:

```
Type_Blob1 || Length(EncryptedBlob1) || Hash(Plaintext_Blob1) ||
[...]
Type_BlobN || Length(EncryptedBlobN) || Hash(Plaintext_Blobn) ||
```

This is enough to calculate the offsets for all the Blobs in the Pack. Length is the length of a Blob as a four byte integer in little-endian format. The type field is a one byte field and labels the content of a blob according to the following table:

Type	Meaning
0	data
1	tree

All other types are invalid, more types may be added in the future.

For reconstructing the index or parsing a pack without an index, first the last four bytes must be read in order to find the length of the header. Afterwards, the header can be read and parsed, which yields all plaintext hashes, types, offsets and lengths of all included blobs.

Indexing

Index files contain information about Data and Tree Blobs and the Packs they are contained in and store this information in the repository. When the local cached index is not accessible any more, the index files can be downloaded and used to reconstruct the index. The files are encrypted and authenticated like Data and Tree Blobs, so the outer structure is `IV || Ciphertext || MAC` again. The plaintext consists of a JSON document like the following:

```
{
  "supersedes": [
    "ed54ae36197f4745ebc4b54d10e0f623eaaaedd03013eb7ae90df881b7781452"
  ],
  "packs": [
    {
      "id": "73d04e6125cf3c28a299cc2f3cca3b78ceac396e4fcf9575e34536b26782413c",
      "blobs": [
        {
          "id": "3ec79977ef0cf5de7b08cd12b874cd0f62bbaf7f07f3497a5b1bbcc8cb39b1ce",
          "type": "data",
          "offset": 0,
          "length": 25
        },{
          "id": "9ccb846e60d90d4eb915848add7aa7ea1e4bbabfc60e573db9f7bfb2789afbae",
          "type": "tree",
          "offset": 38,
          "length": 100
        },
        {
```

```
          "id": "d3dc577b4ffd38cc4b32122cabf8655a0223ed22edfd93b353dc0c3f2b0fdf66",
          "type": "data",
          "offset": 150,
          "length": 123
        }
      ]
    }, [...]
  ]
}
```

This JSON document lists Packs and the blobs contained therein. In this example, the Pack `73d04e61` contains two data Blobs and one Tree blob, the plaintext hashes are listed afterwards.

The field `supersedes` lists the storage IDs of index files that have been replaced with the current index file. This happens when index files are repacked, for example when old snapshots are removed and Packs are recombined.

There may be an arbitrary number of index files, containing information on non-disjoint sets of Packs. The number of packs described in a single file is chosen so that the file size is kept below 8 MiB.

Keys, Encryption and MAC

All data stored by restic in the repository is encrypted with AES-256 in counter mode and authenticated using Poly1305-AES. For encrypting new data first 16 bytes are read from a cryptographically secure pseudorandom number generator as a random nonce. This is used both as the IV for counter mode and the nonce for Poly1305. This operation needs three keys: A 32 byte for AES-256 for encryption, a 16 byte AES key and a 16 byte key for Poly1305. For details see the original paper The Poly1305-AES message-authentication code by Dan Bernstein. The data is then encrypted with AES-256 and afterwards a message authentication code (MAC) is computed over the ciphertext, everything is then stored as IV ‖ CIPHERTEXT ‖ MAC.

The directory `keys` contains key files. These are simple JSON documents which contain all data that is needed to derive the repository's master encryption and message authentication keys from a user's password. The JSON document from the repository can be pretty-printed for example by using the Python module `json` (shortened to increase readability):

```
$ python -mjson.tool /tmp/restic-repo/keys/b02de82*
{
    "hostname": "kasimir",
    "username": "fd0"
    "kdf": "scrypt",
    "N": 65536,
    "r": 8,
    "p": 1,
    "created": "2015-01-02T18:10:13.48307196+01:00",
    "data": "tGwYeKoM0C4j4/9DFrVEmMGAldvEn/+iKC3te/QE/6ox/V4qz58FUOgMa0Bb1cIJ6asrypCx/
↪Ti/
↪pRXCPHLDkIJbNYd2ybC+fLhFIJVLCvkMS+trdywsUkglUbTbi+7+Ldsul5jpAj9vTZ25ajDc+4FKtWEcCWL5ICAOoTAxnPgT+Lk
↪",
    "salt": "uW4fEI1+IOzj7ED9mVor+yTSJFd68DGlGOeLgJELYsTU5ikhG/83/
↪+jGd4KKAaQdSrsfzrdOhAMftTSih5Ux6w==",
}
```

When the repository is opened by restic, the user is prompted for the repository password. This is then used with `scrypt`, a key derivation function (KDF), and the supplied parameters (`N`, `r`, `p` and `salt`) to derive 64 key bytes. The first 32 bytes are used as the encryption key (for AES-256) and the last 32 bytes are used as the message authentication key (for Poly1305-AES). These last 32 bytes are divided into a 16 byte AES key `k` followed by 16 bytes of secret key `r`. The key `r` is then masked for use with Poly1305 (see the paper for details).

Those message authentication keys (`k` and `r`) are used to compute a MAC over the bytes contained in the JSON field `data` (after removing the Base64 encoding and not including the last 32 byte). If the password is incorrect or the key file has been tampered with, the computed MAC will not match the last 16 bytes of the data, and restic exits with an error. Otherwise, the data is decrypted with the encryption key derived from `scrypt`. This yields a JSON document which contains the master encryption and message authentication keys for this repository (encoded in Base64). The command `restic cat masterkey` can be used as follows to decrypt and pretty-print the master key:

```
$ restic -r /tmp/restic-repo cat masterkey
{
    "mac": {
      "k": "evFWd9wWlndL9jc501268g==",
      "r": "E9eEDnSJZgqwTOkDtOp+Dw=="
    },
    "encrypt": "UQCqa0lKZ94PygPxMRqkePTZnHRYh1k1pX2k2lM2v3Q=",
}
```

All data in the repository is encrypted and authenticated with these master keys. For encryption, the AES-256 algorithm in Counter mode is used. For message authentication, Poly1305-AES is used as described above.

A repository can have several different passwords, with a key file for each. This way, the password can be changed without having to re-encrypt all data.

Snapshots

A snapshot represents a directory with all files and sub-directories at a given point in time. For each backup that is made, a new snapshot is created. A snapshot is a JSON document that is stored in an encrypted file below the directory `snapshots` in the repository. The filename is the storage ID. This string is unique and used within restic to uniquely identify a snapshot.

The command `restic cat snapshot` can be used as follows to decrypt and pretty-print the contents of a snapshot file:

```
$ restic -r /tmp/restic-repo cat snapshot 251c2e58
enter password for repository:
{
  "time": "2015-01-02T18:10:50.895208559+01:00",
  "tree": "2da81727b6585232894cfbb8f8bdab8d1eccd3d8f7c92bc934d62e62e618ffdf",
  "dir": "/tmp/testdata",
  "hostname": "kasimir",
  "username": "fd0",
  "uid": 1000,
  "gid": 100,
  "tags": [
    "NL"
  ]
}
```

Here it can be seen that this snapshot represents the contents of the directory `/tmp/testdata`. The most important field is `tree`. When the meta data (e.g. the tags) of a snapshot change, the snapshot needs to be re-encrypted and saved. This will change the storage ID, so in order to relate these seemingly different snapshots, a field `original` is introduced which contains the ID of the original snapshot, e.g. after adding the tag `DE` to the snapshot above it becomes:

```
$ restic -r /tmp/restic-repo cat snapshot 22a5af1b
enter password for repository:
{
  "time": "2015-01-02T18:10:50.895208559+01:00",
```

```
  "tree": "2da81727b6585232894cfbb8f8bdab8d1eccd3d8f7c92bc934d62e62e618ffdf",
  "dir": "/tmp/testdata",
  "hostname": "kasimir",
  "username": "fd0",
  "uid": 1000,
  "gid": 100,
  "tags": [
    "NL",
    "DE"
  ],
  "original": "251c2e5841355f743f9d4ffd3260bee765acee40a6229857e32b60446991b837"
}
```

Once introduced, the `original` field is not modified when the snapshot's meta data is changed again.

All content within a restic repository is referenced according to its SHA-256 hash. Before saving, each file is split into variable sized Blobs of data. The SHA-256 hashes of all Blobs are saved in an ordered list which then represents the content of the file.

In order to relate these plaintext hashes to the actual location within a Pack file , an index is used. If the index is not available, the header of all data Blobs can be read.

Trees and Data

A snapshot references a tree by the SHA-256 hash of the JSON string representation of its contents. Trees and data are saved in pack files in a subdirectory of the directory `data`.

The command `restic cat blob` can be used to inspect the tree referenced above (piping the output of the command to `jq .` so that the JSON is indented):

```
$ restic -r /tmp/restic-repo cat blob␣
↪b8138ab08a4722596ac89c917827358da4672eac68e3c03a8115b88dbf4bfb59 | jq .
enter password for repository:
{
  "nodes": [
    {
      "name": "testdata",
      "type": "dir",
      "mode": 493,
      "mtime": "2014-12-22T14:47:59.912418701+01:00",
      "atime": "2014-12-06T17:49:21.748468803+01:00",
      "ctime": "2014-12-22T14:47:59.912418701+01:00",
      "uid": 1000,
      "gid": 100,
      "user": "fd0",
      "inode": 409704562,
      "content": null,
      "subtree": "b26e315b0988ddcd1cee64c351d13a100fedbc9fdbb144a67d1b765ab280b4dc"
    }
  ]
}
```

A tree contains a list of entries (in the field `nodes`) which contain meta data like a name and timestamps. When the entry references a directory, the field `subtree` contains the plain text ID of another tree object.

When the command `restic cat blob` is used, the plaintext ID is needed to print a tree. The tree referenced above can be dumped as follows:

```
$ restic -r /tmp/restic-repo cat blob␣
↪8b238c8811cc362693e91a857460c78d3acf7d9edb2f111048691976803cf16e
enter password for repository:
{
  "nodes": [
    {
      "name": "testfile",
      "type": "file",
      "mode": 420,
      "mtime": "2014-12-06T17:50:23.34513538+01:00",
      "atime": "2014-12-06T17:50:23.338468713+01:00",
      "ctime": "2014-12-06T17:50:23.34513538+01:00",
      "uid": 1000,
      "gid": 100,
      "user": "fd0",
      "inode": 416863351,
      "size": 1234,
      "links": 1,
      "content": [
        "50f77b3b4291e8411a027b9f9b9e64658181cc676ce6ba9958b95f268cb1109d"
      ]
    },
    [...]
  ]
}
```

This tree contains a file entry. This time, the `subtree` field is not present and the `content` field contains a list with one plain text SHA-256 hash.

The command `restic cat blob` can also be used to extract and decrypt data given a plaintext ID, e.g. for the data mentioned above:

```
$ restic -r /tmp/restic-repo cat blob␣
↪50f77b3b4291e8411a027b9f9b9e64658181cc676ce6ba9958b95f268cb1109d | sha256sum
enter password for repository:
50f77b3b4291e8411a027b9f9b9e64658181cc676ce6ba9958b95f268cb1109d  -
```

As can be seen from the output of the program `sha256sum`, the hash matches the plaintext hash from the map included in the tree above, so the correct data has been returned.

Locks

The restic repository structure is designed in a way that allows parallel access of multiple instance of restic and even parallel writes. However, there are some functions that work more efficient or even require exclusive access of the repository. In order to implement these functions, restic processes are required to create a lock on the repository before doing anything.

Locks come in two types: Exclusive and non-exclusive locks. At most one process can have an exclusive lock on the repository, and during that time there must not be any other locks (exclusive and non-exclusive). There may be multiple non-exclusive locks in parallel.

A lock is a file in the subdir `locks` whose filename is the storage ID of the contents. It is encrypted and authenticated the same way as other files in the repository and contains the following JSON structure:

```
{
  "time": "2015-06-27T12:18:51.759239612+02:00",
  "exclusive": false,
```

```
  "hostname": "kasimir",
  "username": "fd0",
  "pid": 13607,
  "uid": 1000,
  "gid": 100
}
```

The field `exclusive` defines the type of lock. When a new lock is to be created, restic checks all locks in the repository. When a lock is found, it is tested if the lock is stale, which is the case for locks with timestamps older than 30 minutes. If the lock was created on the same machine, even for younger locks it is tested whether the process is still alive by sending a signal to it. If that fails, restic assumes that the process is dead and considers the lock to be stale.

When a new lock is to be created and no other conflicting locks are detected, restic creates a new lock, waits, and checks if other locks appeared in the repository. Depending on the type of the other locks and the lock to be created, restic either continues or fails.

Backups and Deduplication

For creating a backup, restic scans the source directory for all files, sub-directories and other entries. The data from each file is split into variable length Blobs cut at offsets defined by a sliding window of 64 byte. The implementation uses Rabin Fingerprints for implementing this Content Defined Chunking (CDC). An irreducible polynomial is selected at random and saved in the file `config` when a repository is initialized, so that watermark attacks are much harder.

Files smaller than 512 KiB are not split, Blobs are of 512 KiB to 8 MiB in size. The implementation aims for 1 MiB Blob size on average.

For modified files, only modified Blobs have to be saved in a subsequent backup. This even works if bytes are inserted or removed at arbitrary positions within the file.

Threat Model

The design goals for restic include being able to securely store backups in a location that is not completely trusted, e.g. a shared system where others can potentially access the files or (in the case of the system administrator) even modify or delete them.

General assumptions:

- The host system a backup is created on is trusted. This is the most basic requirement, and essential for creating trustworthy backups.

The restic backup program guarantees the following:

- Accessing the unencrypted content of stored files and metadata should not be possible without a password for the repository. Everything except the metadata included for informational purposes in the key files is encrypted and authenticated.
- Modifications (intentional or unintentional) can be detected automatically on several layers:
  1. For all accesses of data stored in the repository it is checked whether the cryptographic hash of the contents matches the storage ID (the file's name). This way, modifications (bad RAM, broken harddisk) can be detected easily.
  2. Before decrypting any data, the MAC on the encrypted data is checked. If there has been a modification, the MAC check will fail. This step happens even before the data is decrypted, so data that has been tampered with is not decrypted at all.

However, the restic backup program is not designed to protect against attackers deleting files at the storage location. There is nothing that can be done about this. If this needs to be guaranteed, get a secure location without any access from third parties. If you assume that attackers have write access to your files at the storage location, attackers are able to figure out (e.g. based on the timestamps of the stored files) which files belong to what snapshot. When only these files are deleted, the particular snapshot vanished and all snapshots depending on data that has been added in the snapshot cannot be restored completely. Restic is not designed to detect this attack.

REST Backend

Restic can interact with HTTP Backend that respects the following REST API. The following values are valid for `{type}`: `data`, `keys`, `locks`, `snapshots`, `index`, `config`. `{path}` is a path to the repository, so that multiple different repositories can be accessed. The default path is `/`.

POST {path}?create=true

This request is used to initially create a new repository. The server responds with "200 OK" if the repository structure was created successfully or already exists, otherwise an error is returned.

DELETE {path}

Deletes the repository on the server side. The server responds with "200 OK" if the repository was successfully removed. If this function is not implemented the server returns "501 Not Implemented", if this it is denied by the server it returns "403 Forbidden".

HEAD {path}/config

Returns "200 OK" if the repository has a configuration, an HTTP error otherwise.

GET {path}/config

Returns the content of the configuration file if the repository has a configuration, an HTTP error otherwise.

Response format: binary/octet-stream

POST {path}/config

Returns "200 OK" if the configuration of the request body has been saved, an HTTP error otherwise.

GET {path}/{type}/

Returns a JSON array containing the names of all the blobs stored for a given type.

Response format: JSON

HEAD {path}/{type}/{name}

Returns "200 OK" if the blob with the given name and type is stored in the repository, "404 not found" otherwise. If the blob exists, the HTTP header `Content-Length` is set to the file size.

GET {path}/{type}/{name}

Returns the content of the blob with the given name and type if it is stored in the repository, "404 not found" otherwise.

If the request specifies a partial read with a Range header field, then the status code of the response is 206 instead of 200 and the response only contains the specified range.

Response format: binary/octet-stream

POST {path}/{type}/{name}

Saves the content of the request body as a blob with the given name and type, an HTTP error otherwise.

Request format: binary/octet-stream

DELETE {path}/{type}/{name}

Returns "200 OK" if the blob with the given name and type has been deleted from the repository, an HTTP error otherwise.

CHAPTER 6

Talks

The following talks will be or have been given about restic:

- 2016-01-31: Lightning Talk at the Go Devroom at FOSDEM 2016, Brussels, Belgium
- 2016-01-29: restic - Backups mal richtig: Public lecture in German at CCC Cologne e.V. in Cologne, Germany
- 2015-08-23: A Solution to the Backup Inconvenience: Lecture at FROSCON 2015 in Bonn, Germany
- 2015-02-01: Lightning Talk at FOSDEM 2015: A short introduction (with slightly outdated command line)
- 2015-01-27: Talk about restic at CCC Aachen (in German)

CHAPTER 7

Introduction

restic is a backup program that is fast, efficient and secure.

For detailed usage and installation instructions check out the documentation.

CHAPTER 8

Quick start

Once you've installed restic, start off with creating a repository for your backups:

```
$ restic init --repo /tmp/backup
enter password for new backend:
enter password again:
created restic backend 085b3c76b9 at /tmp/backup
Please note that knowledge of your password is required to access the repository.
Losing your password means that your data is irrecoverably lost.
```

and add some data:

```
$ restic -r /tmp/backup backup ~/work
enter password for repository:
scan [/home/user/work]
scanned 764 directories, 1816 files in 0:00
[0:29] 100.00%  54.732 MiB/s  1.582 GiB / 1.582 GiB  2580 / 2580 items  0 errors  ETA␣
↪0:00
duration: 0:29, 54.47MiB/s
snapshot 40dc1520 saved
```

For more options check out the usage guide.

CHAPTER 9

Design Principles

Restic is a program that does backups right and was designed with the following principles in mind:

- **Easy:** Doing backups should be a frictionless process, otherwise you might be tempted to skip it. Restic should be easy to configure and use, so that, in the event of a data loss, you can just restore it. Likewise, restoring data should not be complicated.
- **Fast**: Backing up your data with restic should only be limited by your network or hard disk bandwidth so that you can backup your files every day. Nobody does backups if it takes too much time. Restoring backups should only transfer data that is needed for the files that are to be restored, so that this process is also fast.
- **Verifiable**: Much more important than backup is restore, so restic enables you to easily verify that all data can be restored.
- **Secure**: Restic uses cryptography to guarantee confidentiality and integrity of your data. The location the backup data is stored is assumed not to be a trusted environment (e.g. a shared space where others like system administrators are able to access your backups). Restic is built to secure your data against such attackers.
- **Efficient**: With the growth of data, additional snapshots should only take the storage of the actual increment. Even more, duplicate data should be de-duplicated before it is actually written to the storage back end to save precious backup space.

CHAPTER 10

News

You can follow the restic project on Twitter @resticbackup or by subscribing to the development blog.

CHAPTER 11

License

Restic is licensed under "BSD 2-Clause License". You can find the complete text in `LICENSE`.