Device tools

Timo Savola

tsavola@movial.fi

Device tools

by Timo Savola

Revision history

Version:	Author:	Description:
2005-03-11	Savola	Scratchbox 0.9.8.3
2004-06-07	Savola	Fixed typos
2004-05-17	Savola	sbrsh updates
2004-05-15	Savola	Initial version

Table of Contents

Chapter 1. Introduction

Scratchbox [1] contains two tools that help in running programs built for non-native architectures: **sbrsh** and **fakeroot**. They consist of parts that are compiled for the host system and the target system.

The scratchbox-core package provides the host-side tools and the scratchbox-toolchain-*name* package provides target-side binaries compiled with the toolchain *name*. See *Installing Scratchbox* [2] for instructions for obtaining these packages.

sbrsh and fakeroot are licensed under the GNU General Public License [3]. sbrsh's source code repository can be found at the ViewCVS section on the Scratchbox website [1]. fakeroot is maintained by the Debian project [4].

Chapter 2. Scratchbox Remote Shell

sbrsh is a remote command execution system similar to rsh and ssh. It is designed with slow devices and Scratchbox's special requirements in mind. It supports common types of program execution (including terminal emulation), but it is optimized for non-interactive usage. The communication happens on a TCP/IP connection and is not encrypted—sbrsh is meant to be used only on trusted networks (such as a company's LAN or an USB network between a PC and a handheld device).

The server daemon (sbrshd) is run on a device having the same CPU architecture as the compilation target that is being used in Scratchbox. It executes the commands issued by the client (sbrsh) inside a "sandbox" that is created by mounting network filesystems (typically exported by the host that runs the client) and binding local directories (such as `/dev`).

sbrshd contains support functionality that makes remote fakeroot sessions possible (this is described in Appendix A).

2.1. Installing the server

The target device should be running some kind of more-or-less standard Linux installation with kernel version 2.4 or newer. sbrshd has been tested with Familiar [5] and EE [6] distributions.

Only the **sbrshd** binary needs to be installed, but its init script is recommended. If you have installed the scratchbox-toolchain-*target* package, you can copy a binary suitable for the *target* system and the init script from the `/scratchbox/device_tools/sbrsh-`*version*`/`*target*`/` directory. If there isn't a binary available that matches your system (for example C library version is different), you can try a statically linked one. There is a statically linked version in the `/scratchbox/device_tools/sbrsh-`*version*`/`*arch*`-linux-static/` directory for each CPU architecture that has a uClibc [7] toolchain available.

There are a few things to know before starting the daemon. sbrshd uses the **mount** command to bind parts of the directory tree to other locations. The mount command provided by the standard util-linux software package uses the "`--bind`" option for this, but this may not be the case with all versions of mount. For example, Busybox [8] uses the "`-obind`" option instead. sbrshd tries to auto-detect if the mount binary is a symlink to Busybox, but sometimes it may be necessary to pass the correct bind option to sbrshd as the parameter of the "`-b`" option.

sbrshd also needs a list of valid login shells in order to verify that users are authorized to login to the target device. The preferred way is to provide the `/etc/shells` file that lists the paths to the shells. The alternative is to pass a list of paths to the "`-S`" option. The init script provides a list of a few common shells if `/etc/shells` is not available.

Finally: sbrshd needs to be started as root. It needs super-user privileges for mounting and for chrooting to the “sandbox” directory. The command execution is done under the user and group IDs of the user account that is used to execute the command.

If you installed the init script to `/etc/init.d/`, you can probably make the system run it automatically during boot (refer to your system’s documentation). You can use the script manually to start and stop sbrshd:

```
# /etc/init.d/sbrshd start
Starting Scratchbox Remote Shell daemon: done.
# /etc/init.d/sbrshd stop
Stopping Scratchbox Remote Shell daemon: done.
```

Note that by default the init script looks for **sbrshd** from `/sbin/`. You should edit the init script if you want to pass sbrshd some additional options. If you want to run sbrshd manually, you can simply do:

```
# sbrshd
3965
```

It goes automatically to daemon mode (starts a background process and returns to shell immediately). It prints out the process ID of the daemon, which can be used to kill it later on.

See Section B.2 for a full list of command-line options.

2.2. Configuring user accounts

The target device must have normal user accounts (i.e. not root) for executing commands via sbrsh. Each user must have a `.sbrshd` (server-side) configuration file in her home directory that is used by the sbrsh daemon for authentication. It lists all IP addresses (not hostnames) of the hosts that the user uses to connect to the device and corresponding passwords. The passwords are not encrypted at any point, so do not use any important ones. The `.sbrshd` file format is described in Section B.4.

Each user account of the Scratchbox installation has separate `.sbrsh` (client-side) configuration file that corresponds to the server-side one. It is located in the user’s home directory *inside the Scratchbox sandbox*. It lists some or all of the user’s compilation targets. The target settings include the IP address and port of the server, the password that matches the one in `.sbrshd` and a list of network and local filesystems that make up the sandbox. The `.sbrsh` file format is described in Section B.3.

The filesystem configuration needs to recreate the environment that exists inside the Scratchbox sandbox, with a few exceptions. The sandbox on the target device should use its “native” `/proc`, `/dev` and `/dev/pts` directories for things to work. It is also customary to use the device’s `/tmp` directory. The order of mounting matters; the filesystem that becomes the sandbox’s root must be listed first in the

configuration, and so on. The configuration that should be used with a typical Scratchbox installation is described in *Installing Scratchbox* [2].

2.3. Usage

sbrsh is normally invoked implicitly by Scratchbox's *CPU-transparency* feature, but it can also be used manually. It needs to know which target should be used and which program should be executed, but not much more. However, it may be a good idea to pass the current working directory for it with the "`-d`" option:

```
> sbrsh MY-TARGET -d $PWD ./hello
hello world
```

One notable difference to ssh is that sbrsh doesn't use the shell to execute commands, and thus does not read any profile or resource files. It creates the target environment by copying variables from the source environment. It also transfers the umask setting.

Sometimes you may not want to pass some architecture-specific variables for the target binaries or want to change them. sbrsh can override or unset variables in the remote end by using variables prefixed with "`SBOX_ENV_`". The following example should illustrate the behaviour:

```
> export SBOX_ENV_FOO=bar
> sbrsh MY-TARGET env | grep ^FOO
FOO=bar
> export SBOX_ENV_PATH=(UNSET)
> sbrsh MY-TARGET env
sbrsh server: Can't execute command: env (No such file or directory)
```

sbrsh doesn't copy the resource limit settings to the remote environment, but they can be tuned with environment variables of the form "`SBRSH_RLIMIT_`*resource*":

- Limit total CPU time of the process to 10 seconds:

  ```
  > export SBRSH_RLIMIT_CPU=10
  ```

- Always create core dumps:

  ```
  > export SBRSH_RLIMIT_CORE=unlimited
  ```

- Run a program under these conditions:

  ```
  > sbrsh MY-TARGET gnomovision
  ```

See Appendix B for a full list of command-line options and environment variables.

2.4. Debugging

The daemon reports all error conditions to syslog. It also supports a debug log where it writes lot of information about its state and what happens during command execution. It can be enabled with the "`-d`" command-line option, which takes the log filename as its parameter.

sbrshd does not have to be restarted in order to enable logging: it will open the log when it receives the `USR1` signal. Logging can also be turned off with the `USR2` signal. If a log filename was not specified with the "`-d`" option (i.e. logging was not initially enabled), the log will be written to `/tmp/sbrshd-`*port*`.log` ("*port*" being the listening port of the daemon—1202 by default).

Here is an example session where the debug log is written to the terminal device:

```
# sbrshd -d ‘tty‘
3992
01-06-1970 03:41:45.504  3992  DAEMON Debugging enabled
01-06-1970 03:41:45.506  3992  DAEMON sbrshd version 1.4.4 (protocol version 4)
01-06-1970 03:41:45.506  3992  DAEMON Listening at port 1202
01-06-1970 03:41:45.506  3992  DAEMON Valid login shells: /bin/sh /bin/bash
/bin/zsh /bin/ash /bin/tcsh
01-06-1970 03:41:45.507  3992  DAEMON Mounts expire after 900 seconds
01-06-1970 03:41:45.507  3992  DAEMON Waiting for connection
# kill -USR2 3992
01-06-1970 03:42:11.513  3992  DAEMON User defined signal 2
01-06-1970 03:42:11.513  3992  DAEMON Debugging disabled
# kill -USR1 3992
01-06-1970 03:42:14.370  3992  DAEMON Debugging enabled
01-06-1970 03:42:14.371  3992  DAEMON Checking for expired mounts
01-06-1970 03:42:14.371  3992  DAEMON Waiting for connection
```

The type of the process is displayed next to the process ID in the log. sbrshd has four types of processes that write log entries: `DAEMON` accepts connections and handles mounts, `HANDLER` handles I/O between the client and the command process, `RELAY` relays fakeroot messages and `COMMAND` "bootstraps" and executes the command.

Chapter 3. Fakeroot

Fakeroot [9] is a utility that runs programs in an environment that looks as if they were run with super-user privileges. It is used primarily for setting file ownerships and modes before packaging them. You can for example create device nodes and store them in a tarball while logged in as a normal user. Of course, the programs run from a fakeroot session cannot really do privileged system calls; fakeroot keeps an in-memory database of file ownerships and such things.

Fakeroot was developed by the Debian Project [4] to help in building Debian packages. The Debian packaging system needs a root environment so that it would be as easy as possible to set up ownerships and permissions.

3.1. Fakeroot in Scratchbox

Scratchbox introduces new requirements for fakeroot. During the development of Scratchbox an enhanced version of fakeroot was developed with the name *fakeroot-net*. It was later merged with the upstream project and nowadays Scratchbox uses the upstream codebase.

When using sbrsh to implement CPU-transparency in Scratchbox, the command execution can jump from host to target within a fakeroot session. Since both ends use the same filesystems (via NFS), they must also use the same fakeroot session. This is not possible with the original design that uses SYSV IPC. The Scratchbox version uses TCP/IP sockets for its internal communication. (The TCP version of the fakeroot command is also available in the Debian package with the name **fakeroot-tcp**.)

> **Note:** Using TCP sockets in fakeroot is not enough to implement network-transparent fakeroot sessions. The sbrsh server (sbrshd) is used to filter the information passed between the remote fakeroot environment and the fakeroot daemon (faked) that keeps the database. The reason for this is explained in Appendix A.

Scratchbox provides the fakeroot command, the fakeroot daemon (faked) and a host version of the fakeroot library (libfakeroot). They are sufficient for running host tools in fakeroot, but a target version of libfakeroot needs to be installed for each Scratchbox target in order to run target binaries in fakeroot. *Installing Scratchbox* [2] describes how to do that. Scratchbox's fakeroot is compatible with the libfakeroot provided by the Debian package, so you can use that aswell.

Refer to fakeroot's manual page (Appendix C) for usage instructions.

3.2. Known issues

The fakeroot environment is imposed upon a process by using the C-library's LD_PRELOAD environment variable. libfakeroot is preloaded by the dynamic linker whenever it loads a binary. This means that fakeroot does not work with statically linked binaries.

There is also another side-effect. Since libfakeroot is loaded into the same process image with the "victim" program, they share the same file descriptor table. Some programs (such as the configure scripts) use hard-coded descriptor numbers. libfakeroot needs one file descriptor for its communication socket, and if the program starts to use the same file descriptor, there will be trouble. fakeroot tries to monitor the status of its descriptor so that it can open a new socket if the descriptor has been changed. If you start seeing messages about hijacked file descriptors, you can try to make fakeroot use some other file descriptor with the **--fd-base** option. Its default value is (*descriptor_table_size* - 100).

3.3. Debugging

The fakeroot daemon can be launched with debug enabled and left running on the foreground:

```
$ faked --debug --foreground
33366:5027
```

The first number is the TCP/IP port it listens to, and the second number is its process ID. Now, in another terminal, setup a fakeroot session manually that uses the daemon we started:

```
$ export FAKEROOTKEY=33366
$ export LD_PRELOAD=/scratchbox/tools/lib/libfakeroot-tcp.so.0
```

Now you can run programs in the hand-made fakeroot session and see the daemon's cryptic debug output in the other terminal. This way you can also use a debugger to debug a program within a fakeroot environment.

> **Note:** `/scratchbox/tools/lib/libfakeroot-tcp.so.0` is the host version. If you are running target binaries, you should set LD_PRELOAD to `/usr/lib/libfakeroot/libfakeroot-tcp.so.0`.

When using a remote fakeroot session, the communication can be traced using the sbrsh daemon's debug log. See Section 2.4 for instructions.

Bibliography

[1] *Scratchbox website (http://scratchbox.org/)*.

[2] *Installing Scratchbox (http://scratchbox.org/docs/installdoc/index.html)*, Valtteri Rahkonen.

[3] *GNU General Public License (http://www.gnu.org/licenses/gpl.html)*, Free Software Foundation.

[4] *Debian GNU/Linux (http://www.debian.org/)*.

[5] *The Familiar Project website (http://familiar.handhelds.org/)*.

[6] *EE-distro - Scratchbox Support Distribution for ARM (http://scratchbox.org/index.html?id=59)*, Karri Niskala.

[7] *uClibc website (http://uclibc.org/)*.

[8] *Busybox website (http://busybox.net/)*.

[9] *fakeroot (http://fakeroot.alioth.debian.org/)*.

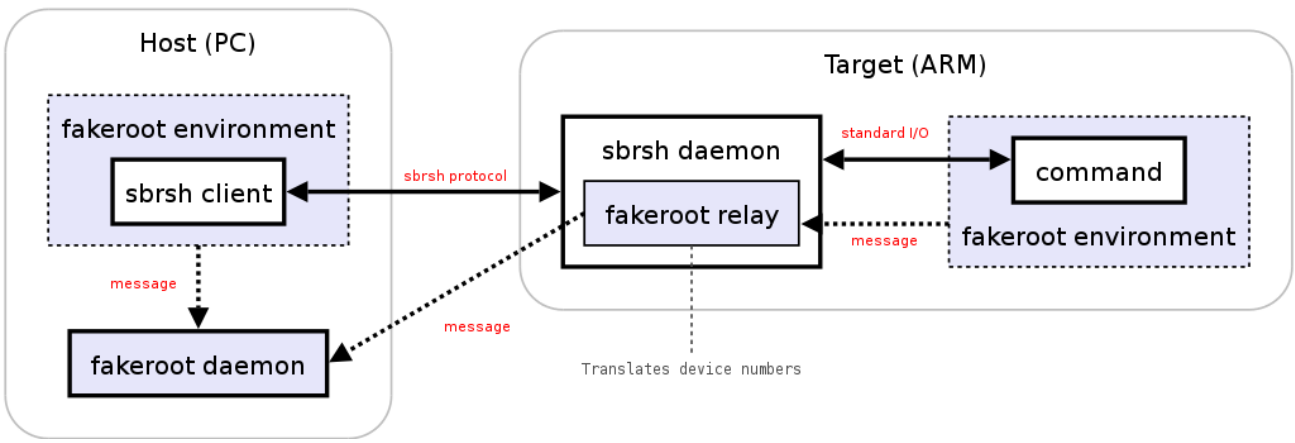
Appendix A. Implementation of network fakeroot

faked maintains a list of entries based on their device and inode numbers of the files that have been modified during a fakeroot session. The entries contain a data structure that is essentially the same as the one used by the `stat` system call. The TCP version introduces an additional `remote` field in the entry, which works like a “namespace” for the devices and inodes. All files on the local filesystems belog to the default namespace (`remote` is not set).

When a remote command is run within a fakeroot session, sbrsh resolves the device numbers of the NFS filesystems that are listed in its config file for the used target. If they are not exported by the local host but some third host, it tries to find out if the NFS filesystems are mounted on the local host and use the device numbers of the mount points.

sbrshd receives the list of mount entries and finds out what their device numbers are on the target device. Then it creates a *relay* process that listens for connections from local fakeroot sessions. When it receives one, it makes a corresponding connection to the faked running on the Scratchbox host. It maintains as many connection pairs as there are processes running within the local fakeroot session. The relay copies messages from the local session to the remote daemon and responses from the daemon to the session, and translates the device numbers in the messages between the local and remote device number “spaces”.

If the relay finds an unlisted device number in one of the incoming messages, it does not translate it but sets the value of the `remote` field to the IP address of the host it is running at. This way faked can serve unknown filesystems without the danger of device number/inode collisions.



Appendix B. sbrsh reference

B.1. Client usage

Executing remote command:

```
sbrsh target [-c|--config path] [-d|--directory dir] [command [args]]
```

Unmounting all filesystems of a *target* (see Section B.3):

```
sbrsh target [-c|--config path] --umount-all
```

target	symbolic name of the target configuration; must be specified before any options (see Section B.3)
path	the absolute path to the user's configuration file (default is `.sbrsh` in the home directory)
dir	the user's current directory at the target device (default is the filesystem root)
command	name of the command to be executed (may be looked up from `PATH`); if command is not specified, the user's login shell at the remote host is executed
args	zero or more arguments for *command*

B.2. Daemon usage

```
sbrshd [-p|--port port]                  [-d|--debug log]      [-m|--mount-bin mount]
       [-u|--umount-bin umount]          [-t|--mount-tab mtab] [-b|--bind-opt opt]
       [-e|--mount-expiration mins|none] [-s|--shells file]    [-S|--shell-list list]
```

port	sets a custom port number (default is 1202)
log	enables debugging to a log file
mount	specifies the mount binary path (default is `/bin/mount`)
umount	specifies the umount binary path (default is `/bin/umount`)
mtab	specifies the mount table path (default is `/proc/mounts`)
opt	specifies the option used when binding a path to a mount point (default is "`--bind`", or "`-obind`" if mount binary is *Busybox*)

mins	specifies the number of minutes to wait before expiring unused mount points (default is 15); 0 means that filesystems are unmounted immediately after commands exit; "`none`" means that filesystems are unmounted only when sbrshd exits
file	specifies the path to a file that lists all valid login shells (default is `/etc/shells`)
list	specifies a colon-separated list of valid login shells; `/etc/shells` is not read if this is specified

B.3. Client configuration file

sbrsh configuration file lists all known *targets* (see Section B.1). The first line of a *target* block must not contain whitespaces before the name of the *target*. The subsequent lines must be indented. "#" is a line end comment character. The root of the command's sandbox will be the '*ipaddress*`-`*target*' directory under the user's home directory.

The layout of the first line:

`target` [*`username@`*]*`ipaddress`*[`:`*`port`*] *`password`*

The subsequent lines define the mounts needed by the *target* (*type* is either "`nfs`" or "`bind`"):

`type filesystem point` [*`options`*]

Here's an example configuration:

```
ARM  john@10.0.0.3  asdf
  nfs   10.0.0.2:/scratchbox/users/john/targets/ARM  /      rw,nolock,noac
  nfs   10.0.0.2:/scratchbox/users/john/home         /home  rw,nolock,noac
  bind  /dev      /dev
  bind  /dev/pts  /dev/opts
  bind  /proc     /proc
  bind  /tmp      /tmp
```

B.4. Daemon configuration file

sbrshd configuration file lists all known client IP addresses and passwords. Each user has her own `.sbrshd` file in his home directory. "#" is a line end comment character.

The layout is:

```
ipaddress  password
```

B.5. Environment variables

The command execution environment at the target device can be controlled via a few environment variables.

"`SBOX_ENV_`" prefix will be stripped from all variables having one. If a corresponding variable without the prefix exists, it will be overridden. If the variable's value is "`(UNSET)`", the corresponding variable will be removed from environment. For example the dynamic linker can be controlled this way (via the `LD_*` variables) without affecting the sbrsh client itself.

The resource limits can be set using variables with the "`SBRSH_RLIMIT_`" prefix. The value can be either an integer or "`unlimited`". The supported setting are:

`SBRSH_RLIMIT_CPU`	CPU time in seconds
`SBRSH_RLIMIT_FSIZE`	max filesize
`SBRSH_RLIMIT_DATA`	max data size
`SBRSH_RLIMIT_STACK`	max stack size
`SBRSH_RLIMIT_CORE`	max core file size
`SBRSH_RLIMIT_RSS`	max resident set size
`SBRSH_RLIMIT_NPROC`	max number of processes
`SBRSH_RLIMIT_NOFILE`	max number of open files
`SBRSH_RLIMIT_MEMLOCK`	max locked-in-memory address space
`SBRSH_RLIMIT_AS`	address space (virtual memory) limit

Appendix C. fakeroot 1.2.4 manual page

NAME

fakeroot - run a command in an environment faking root privileges for file manipulation

SYNOPSIS

fakeroot [-l|--lib library] [--faked faked-binary] [-i load-file] [-s save-file] [-u|--unknown-is-real] [-b|--fd-base] [-h|--help] [-v|--version] [--] [command]

DESCRIPTION

fakeroot runs a command in an environment wherein it appears to have root privileges for file manipulation. This is useful for allowing users to create archives (tar, ar, .deb etc.) with files in them with root permissions/ownership. Without fakeroot one would need to have root privileges to create the constituent files of the archives with the correct permissions and ownership, and then pack them up, or one would have to construct the archives directly, without using the archiver.

fakeroot works by replacing the file manipulation library functions (chmod(2), stat(2) etc.) by ones that simulate the effect the real library functions would have had, had the user really been root. These wrapper functions are in a shared library /usr/lib/libfakeroot.so* which is loaded through the LD_PRELOAD mechanism of the dynamic loader. (See ld.so(8))

If you intend to build packages with fakeroot, please try building the fakeroot package first: the "debian/rules build" stage has a few tests (testing mostly for bugs in old fakeroot versions). If those tests fail (for example because you have certain libc5 programs on your system), other packages you build with fakeroot will quite likely fail too, but possibly in much more subtle ways.

Also, note that it's best not to do the building of the binaries themselves under fakeroot. Especially configure and friends don't like it when the system suddenly behaves differently from what they expect. (or, they randomly unset some environment variables, some of which fakeroot needs).

OPTIONS

-l library, --lib library

Specify an alternative wrapper library.

--faked binary

Specify an alternative binary to use as faked.

[--] command
Any command you want to be ran as fakeroot. Use '--' if in the command you have other options that may confuse fakeroot's option parsing.

-s save-file
Save the fakeroot environment to save-file on exit. This file can be used to restore the environment later using -i. However, this file will leak and fakeroot will behave in odd ways unless you leave the files touched inside the fakeroot alone when outside the environment. Still, this can be useful. For example, it can be used with rsync(1) to back up and restore whole directory trees complete with user, group and device information without needing to be root. See /usr/share/doc/fakeroot/README.saving for more details.

-i load-file
Load a fakeroot environment previously saved using -s from load-file. Note that this does not implicitly save the file, use -s as well for that behaviour. Using the same file for both -i and -s in a single fakeroot invocation is safe.

-u, --unknown-is-real
Use the real ownership of files previously unknown to fakeroot instead of pretending they are owned by root:root.

-b fd Specify fd base (TCP mode only). fd is the minimum file descriptor number to use for TCP connections; this may be important to avoid conflicts with the file descriptors used by the programs being run under fakeroot.

-h Display help.

-v Display version.

EXAMPLES
Here is an example session with fakeroot. Notice that inside the fake root environment file manipulation that requires root privileges succeeds, but is not really happening.

```
$  whoami
joost
$ fakeroot /bin/bash
#  whoami
root
# mknod hda3 b 3 1
# ls -ld hda3
brw-r--r--   1 root     root       3,   1 Jul  2 22:58 hda3
# chown joost:root hda3
# ls -ld hda3
```

```
brw-r--r--   1 joost    root       3,   1 Jul  2 22:58 hda3
# ls -ld /
drwxr-xr-x  20 root     root         1024 Jun 17 21:50 /
# chown joost:users /
# chmod a+w /
# ls -ld /
drwxrwxrwx  20 joost    users        1024 Jun 17 21:50 /
# exit
$ ls -ld /
drwxr-xr-x  20 root     root         1024 Jun 17 21:50 //
$ ls -ld hda3
-rw-r--r--   1 joost    users           0 Jul  2 22:58 hda3
```

Only the effects that user joost could do anyway happen for real.

fakeroot was specifically written to enable users to create Debian GNU/Linux packages (in the deb(5) format) without giving them root privileges. This can be done by commands like dpkg-buildpackage -rfakeroot or debuild -rfakeroot (actually, -rfakeroot is default in debuild nowadays, so you don't need that argument).

SECURITY ASPECTS

fakeroot is a regular, non-setuid program. It does not enhance a user's privileges, or decrease the system's security.

FILES

/usr/lib/libfakeroot/libfakeroot.so* The shared library containing the wrapper functions.

ENVIRONMENT

FAKEROOTKEY

The key used to communicate with the fakeroot daemon. Any program started with the right LD_PRELOAD and a FAKEROOTKEY of a running daemon will automatically connect to that daemon, and have the same "fake" view of the file system's permissions/ownerships. (assuming the daemon and connecting program were started by the same user).

LIMITATIONS

Library versions

Every command executed within fakeroot needs to be linked to the same version of the C library as fakeroot itself.

open()/create()

fakeroot doesn't wrap open(), create(), etc. So, if user joost does either

```
touch foo
fakeroot
ls -al foo
```

or the other way around,

```
            fakeroot
            touch foo
            ls -al foo

            fakeroot has no way of knowing that in the first case, the owner
            of foo really should be joost while the second case it should
            have been root. For the Debian packaging, defaulting to giving
            all "unknown" files uid=gid=0, is always OK. The real way around
            this is to wrap open() and create(), but that creates other
            problems, as demonstrated by the libtricks package. This package
            wrapped many more functions, and tried to do a lot more than
            fakeroot . It turned out that a minor upgrade of libc (from one
            where the stat() function didn't use open() to one with a stat()
            function that did (in some cases) use open()), would cause unex-
            plainable segfaults (that is, the libc6 stat() called the
            wrapped open(), which would then call the libc6 stat(), etc).
            Fixing them wasn't all that easy, but once fixed, it was just a
            matter of time before another function started to use open(),
            never mind trying to port it to a different operating system.
            Thus I decided to keep the number of functions wrapped by fake-
            root as small as possible, to limit the likelihood of 'colli-
            sions'.

      GNU configure (and other such programs)
            fakeroot, in effect, is changing the way the system behaves.
            Programs that probe the system like GNU configure may get con-
            fused by this (or if they don't, they may stress fakeroot so
            much that fakeroot itself becomes confused). So, it's advisable
            not to run "configure" from within fakeroot. As configure should
            be called in the "debian/rules build" target, running
            "dpkg-buildpackage -rfakeroot" correctly takes care of this.

BUGS
      It doesn't wrap open(). This isn't bad by itself, but if a program does
      open("file", O_WRONLY, 000), writes to file "file", closes it, and then
      again tries to open to read the file, then that open fails, as the mode
      of the file will be 000. The bug is that if root does the same, open()
      will succeed, as the file permissions aren't checked at all for root. I
      choose not to wrap open(), as open() is used by many other functions in
      libc (also those that are already wrapped), thus creating loops (or
      possible future loops, when the implementation of various libc func-
      tions slightly change).

COPYING
      fakeroot is distributed under the GNU General Public License. (GPL 2.0
      or greater).

AUTHORS
      joost witteveen
            <joostje@debian.org>

      Clint Adams
            <schizo@debian.org>
```

```
       Timo Savola

MANUAL PAGE
       mostly  by  J.H.M.  Dassen <jdassen@debian.org> Rather a lot mods/addi-
       tions by joost and Clint.

SEE ALSO
       faked(1) dpkg-buildpackage(1), debuild(1) /usr/share/doc/fakeroot/DEBUG



Debian Project                   6 August 2004                      fakeroot(1)
```