rat: A Secure Archiving Program With Fast Retrieval

Willem A. (Vlakkies) Schreüder – University of Colorado at Boulder
Maria Murillo – University of Colorado at Boulder

ABSTRACT

A new archive format called **rat** was developed. This format was designed to allow very fast retrieval of individual files. This is achieved using a table of contents to quickly find the file.

Each file in the archive is individually compressed with a compression method specific to the file. A user created configuration file is used to specify what type of compression to use on each file based on parameters such as the file extension and file size. Multiple sets of rules can be defined and activated from the command line to achieve different aims such as speed or size or to deal with different types of file sets. Parameters passed to the compression algorithms may also be specified.

The format also provides for signatures to be stored with the files. The program will generate and save the signature when the archive is created and verify the file when the archive is restored. Encryption is possible but not implemented.

The format is quite robust. If the archive is truncated or the table of contents is lost, the files in the portion that survived can still be recovered. Every file and table is preceded by a magic number so even recovery from bit rot may be possible.

The current implementation incorporates gzip (as zlib), bzip2, and LZO compression. Library versions of these compression algorithms are linked in for performance reasons. Only PGP signatures are currently implemented. Due to export restrictions on encryption software, a child process is spawned to execute a separate binary to do the signature creation and verification.

A library called **librat** implements all the functionality required to create the archive and restore files from the archive. Alternate user interfaces or embedded applications are therefore quite readily created. Three front ends to **librat** have been implemented. The first front end is a simple command line interface similar to **tar**. The second front end is character based interface that allows the user to browse the archive and selectively restore files similar to the **restore** program used with **dump**. The third front end is a GUI implemented using Qt.

Introduction

Archiving tools are widely used for many purposes. Of these **tar**, **zip**, and **dump**/**restore** are probably the most widely used. **dump**/**restore** is mostly for tape backups, although **restore** does provide a user interface to browse the archive and select which files to restore. **zip** is very popular on DOS based systems and is geared towards making archives on disk rather than to tape. **tar** is the most widely used archive program on UNIX systems. It is used for backing up to tape as well as making archives on disk.

As a class project for the System Administration class of Evi Nemeth at the University of Colorado at Boulder, a requirement for a public domain archive program that combine the best features of **tar**, **zip** and **dump**/**restore** was posed as a term project. The specific features that this program should have were defined as follows:

- The archive should use space as efficiently as possible.
- The archive should be geared towards fast retrieval of individual files.
- The format should be robust and allow access to files in case of errors such as a truncated archive.
- The format should support security.
- The format should be extensible.
- Special files such as hard links, soft links, files with holes and device files must be supported.
- The archive is intended to be written to a random access device such as a disk drive rather than a tape drive.

For the implementation, it was decided that the following features are important:

- The design should be modular to allow different front ends and embedded applications.
- Support for multiple, fixed-size volumes should be incorporated.
- Support for commonly used compression methods should be linked into the program for performance reasons.

- Due to export restrictions, any encryption software should be invoked as an independent binary.
- The implementation should support remote backups.
- A front end with a **restore**-like browsing facility should be provided.

Fulfilling the above requirements was determined to be more important than being compatible with existing software like **tar**.

Much consideration was given to adapting existing software like **tar** to the task. **tar** is a de facto industry standard, but has been superseded by pax in the POSIX.2 specification. The new, extended file format for tar is specified in POSIX.1. **tar** is freely available, and the Gnu tar-1.12 implementation already supports device files, files with holes, remote backups and such desirable features.

One approach would be to implement per-file compression in tar, and then store the table of contents as just another file. When restoring the archive with an older version of tar, the files would be restored as the compressed version of the file which can then be manually uncompressed. The table of contents would be restored as an extra file. While this is a valid alternative, this approach was abandoned because the tar format does not use space efficiently. **tar** archives are written in 512 byte blocks. The header for each file uses 512 bytes, and the data are stored in 512 byte blocks. A one byte file would therefore consume 1024 bytes. The blocks are padded with null bytes. When the entire archive is compressed using gzip, these inefficiencies are less important because long runs of null bytes compress well. However, the drawback of this approach is that the entire archive must be filtered through the decompression algorithm to retrieve any individual file.

Header
File*0*
. . .
File*n*
Table of Contents
UID Table
GID Table

Figure 1: Rat archive format.

In order to fulfill all the requirements set out above, a completely new archive format named **rat** was designed and implemented. The name **rat** was chosen to be **tar** backwards, and to fit in with the animal theme of some of the names in the GNU project and book covers from O'Reilly and Associates.

rat Archive Format

Figure 1 shows the layout of the **rat** archive. All strings in the archive are saved as variable length sequence of bytes terminated by a null. All integers are saved as unsigned binary integers with the most significant byte first.

The header section contains the archive label and similar global information, as well as 64 bit pointers to the start of the files section, table of contents (TOC) and UID and GID tables. The order in which the remaining sections of the archive appears is determined by these pointers, but this is usually in the order shown in Figure 1.

Magic	(uint32)
Header Size	(uint16)
Signature Type	(uint8)
Compression Type	(uint8)
Mode	(uint16)
UID	(uint32)
GID	(uint32)
Mtime	(uint64)
Ctime	(uint64)
Uncompressed Size	(uint64)
Flags	(uint16)
Flag Dependent Data (ACLs,...)	
Data Size	(uint64)
Data	
Signature Size	(uint16)
Signature	

Figure 2: File Layout.

Files

The files section is the bulk of the archive. It consists of files and file pointers saved contiguously. A file pointer contains the file name, file type and a pointer to the file. A file contains the data for the file as shown in Figure 2. Each file contains all the relevant information from the inode of the file, the contents of the file (data blocks) and possibly a signature.

This layout closely mirrors the layout in the disk file system. The file pointers correspond to directory entries. Hard links are supported by having multiple file pointers point to the same file. The file pointer names the file, but does not contain any data other than the file type and a pointer to the file. The special value zero in the pointer field is used to indicate that the file immediately follows the file pointer.

The file entry contains the file data but not the name of the file. The file data consists of three sections, a header section, data section and signature section. Each section is preceded by a size entry, indicating the size of the rest of the section. This makes it easy to process the contents of the file by reading only the size entries and treating the remained of the section as a block of data. The data size is stored as a 64 bit integer so extremely large files are supported.

The header section contains file attribute information such as the owner, permissions, size and times, but not the name of the file. Times are stored as 64 bit integers in the UNIX style. Special information such as the device major and minor numbers and soft link text are stored in the data section. The data size will always reflect the true number of bytes used by the data section.

Only the UID and GID are stored for each file. The corresponding user and group names can be obtained from the UID and GID tables stored in the archive. The Flags field is used to signal various special cases. Bit 0 in the Flags field is used to indicate that the file contains holes. Such files are stored like any other file since the compression algorithms compresses long sequences of zeroes quite efficiently. When this flag is set, the restore function finds long sequences of zeroes and recreates the holes. Bit 1 in the Flags field is used to indicate that the file has an associated Access Control List (ACL) which will appear at the end of the header. The remaining bits are reserved for future expansion.

Code	Algorithm
0	None
1	gzip (zlib)
2	bzip2
3	LZO

Table 1: Compression Codes.

The Compression Type field defines the compression method used in the data section. Table 1 shows the codes used for the different methods currently implemented. The Signature Type field defines the type of signature used. Currently only values of 0 (CRC-32), 1 (MD5) and 2 (MD5+PGP) are implemented. This field can also specify an encryption method.

The signature section contains the checksum of the header and data sections. The checksum always appears first in the signature section. When files are signed using PGP, the PGP signature immediately follows the checksum.

It should be noted that the file pointers in the file section replicates the information already contained in the table of contents (TOC) described below. The purpose of the file pointers is to allow recovery of the information in the archive if the TOC is missing, such as when the archive is accidentally truncated.

Table of Contents

The Table of Contents (TOC) typically follows the files section. The layout of the TOC is shown in Figure 3. For each file, the TOC contains only the file name, file type and two 64 bit pointers, one to the file pointer and one to the beginning of the actual file, both of which are in the files section. When the item in the TOC is a directory, the full path to the file is saved as the name. For all other items such as regular files, links and device files only the name of the file is stored. The full path is implied by the preceding directory in the TOC. The local directory is implied at the beginning of the TOC. In order for this to uniquely name each file, subdirectories must always appear after non-directory items. Reconstructing the full path name is trivial if the TOC is read sequentially. When the paths are long, this convention reduces the size of the TOC considerably.

Magic			(uint32)
Signature Type			(uint8)
Number of Entries			(uint64)
Name*0* (...0)	Type*0* (uint8)	Link*0* (uint64)	File*0* (uint64)
	. . .		
Name*n*	Type*n*	Link*n*	File*n*
Signature Size			(uint16)
Signature			

Figure 3: Table of contents.

The purpose for saving both the offset of the file pointer and the file itself in the TOC is to expedite updating of the archive. The pointers are used to track which file pointers and files are still referenced. Unreferenced items are removed when the archive is updated.

A checksum of the TOC is computed and stored in a signature section identical to that for a file in the files section. If so indicated by the signature type code, the checksum will also be signed.

The layout of the UID and GID tables are similar to the TOC. It lists the UID and user name or GID and group name in pairs, ordered by UID or GID. These tables are typically small, but can save a large amount of space since the user and group names does not have to be saved for each file. Both the UID and GID tables have corresponding signature sections.

Security

The **rat** archive format attempts to make the archive secure by securing each individual file, the TOC, UID and GID tables. A checksum is computed for every such entity. Currently CRC-32 (32 bit) and MD5 (128 bit) checksums are implemented, but more secure checksums can be added as they become available. The type of checksum can be set in the **.ratrc** configuration file, the default being MD5.

Of course, checksums only protect the archive from accidental corruption, not malicious alteration. To secure the archive, the checksum can be signed. In the current implementation, only the use of PGP to sign the archive is supported, and implies that an MD5 checksum will be used.

When a file or the TOC, UID or GID table is read from the archive, the checksum is recomputed and compared against the stored checksum. If signed, PGP is used to verify the checksum. If either the checksum does not match the stored checksum, or the signature does not verify the checksum, the read will return the result ErrCheck.

librat Implementation

The **rat** archive was implemented as a library named **librat**. The **librat** interface is quite compact. The calls to **librat** to open and close the archive are:

```
rat_t rat_init(char* device,
      int verbose,
      char* label,int level,
      char* user,char* pass);
int   rat_open_write(rat_t rat);
int   rat_open_read(rat_t rat);
int   rat_open_update(rat_t rat);
int   rat_close(rat_t rat);
```

A call to rat_init is used to initialize the library and initialize parameters such as the device to be used for the archive, verboseness level, set of rules (level) and user name and pass phrase for the signature.

rat_init reads the **.ratrc** configuration file and returns a private structure of type rat_t that is used by all other functions.

When creating an archive, rat_open_write is used to open the archive, while rat_open_read is used to open the archive for reading. To modify an existing archive, rat_open_update is used to open the archive. When done, rat_close is used to close the archive.

Reading an archive is achieved by calling the functions

```
int   rat_open_toc(rat_t rat);
int   rat_next_toc(rat_t rat,
        char path[],
        char* type,
        u_int64_t* offset);
int   rat_inq_file(rat_t rat,
        const u_int64_t offset,
        info_t* info);
int   rat_read_file(rat_t rat,
        u_int64_t offset);
int   rat_check_toc(rat_t rat);
```

Each call to rat_open_toc starts reading the TOC from the beginning. Each call to rat_next_toc returns the full path name of the next entry in the TOC as well as the type and offset. The returned type can be tested using an set of enumerated constants defined in the header file. The offset returned by rat_next_toc can be passed to rat_inq_file or rat_read_file. A call to rat_inq_file returns all the fields stored in the file header in an structure of type info_t. A call to rat_read_file restores the file, including any necessary directories. If the file has a signature, rat_read_file also checks the signature. Note that rat_read_file will create missing intervening directories, but will not recursively restore files and subdirectories.

The integrity of the archive can be checked by calling rat_check_toc. Not only does this call check that the TOC matches file pointer entries in the files section, but it also checks the checksums and signatures of the files.

After calling rat_open_write or rat_open_update, a file can be added to the archive by calling

```
int rat_add_toc(rat_t rat,
               const char* path);
```

A call to rat_add_toc simply adds the list of files specified to the TOC. Only when rat_close is called are the files in the TOC written to the archive. rat_add_toc recursively adds all files and subdirectories when the path is a directory. rat_add_toc returns -1 if the addition failed, typically because the file does not exist or is not readable. If the addition succeeded, rat_add_toc returns the number of entries in the TOC that was replaced. Adding all new files will result in a 0 being returned. The existing entries in the TOC are searched for every file being added to ensure that each file is unique. Adding the same file twice causes the previous occurrence to be replaced.

After calling rat_open_write or rat_open_update, a file can be removed from the archive by calling

```
int64_t rat_find_toc(rat_t rat,
                const char* path);
int rat_del_toc(rat_t rat,
                u_int64_t index);
```

A call to rat_find_toc is used to get the index of the named file in the TOC. If the file is not in the TOC, -1 is returned. Both files and directories are found by rat_find_toc. A call to rat_del_toc with the index returned by rat_find_toc deletes that file from the TOC. If the index points to a directory, all files and subdirectories of that directory are also deleted from the TOC. As with rat_add_toc, removing the files from the archive is delayed until rat_close is called.

A call to rat_close closes the archive. If the archive was opened with rat_open_read, the archive is simply closed. If the archive was opened with rat_open_write, all the items in the TOC are written to the archive before it is closed. If the archive was opened with rat_open_update, a call to rat_close will cause the archive to be updated to reflect the current contents of the TOC. When rat_open_update is first called, the TOC, UID and GID tables are read into memory. Subsequently, when rat_close is called, the first operation is to remove all items in the archive that were removed by calls to rat_del_toc or replaced by calls to rat_add_toc. This is done in place, without copying the archive, by moving the remaining entries in the archive closer to the head of the archive so that the parts of the archive that did not change are contiguously packed at the head of the archive. New files are then appended to the truncated archive. Finally, the TOC, UID and GID files are appended from memory. This procedure makes adding only new files efficient since it requires only the TOC, UID and GID tables to be read from the archive, the files added, and then the

TOC, UID and GID tables rewritten to the archive.

Calls to rat_open and rat_close can be freely intermixed. The library will flush changes to the archive as necessary. For example, after creating a file with calls to rat_open_write and rat_add_toc, a call to rat_open_read will write all files to the archive and reopen the archive for reading. Extra calls to rat_close will return safely.

Finally, rat_free should be used to free the rat structure allocated by rat_init.

The .ratrc Configuration File

When rat_init is called, **librat** attempts to read a file called **.ratrc**. It first tries the current directory for a **.ratrc** file, then the user's home directory for a **.ratrc** file, and finally /etc/ratrc. The first file found is used.

The purpose of the **.ratrc** file is to allow the user to configure **librat**. In particular, **.ratrc** is intended to control the compression algorithms to be used on particular files and how to interface with the security programs. An example **.ratrc** file is shown in Figure 4.

```
# How to run pgp
pgps  pgps -btu%s %s -o-
pgpv  pgpv +batchmode=1 %s.sig
# Global parameters for BZIP2
BZ2blockSize 9
BZ2workFact  0
# Set of Rules 1
level 1
any CompNone
# Set of Rules 2
level 2
any CompGZ
# Set of Rules 3
level 3
any CompBZ2
# Set of Rules 4
level 4
any CompLZO
# Set of Rules 9 (default)
level 9
ext     .rat   CompNone
ext     .gz    CompNone
ext     .bz2   CompNone
ext     .gif   CompNone
ext     .jpg   CompNone
hole           CompBZ2
smaller 8192   CompGZ
any            CompBZ2
```

Figure 4: Example **.ratrc** file.

When calling rat_init, the level parameter is interpreted as the compression level. In the **.ratrc** file, this is manifested as up to nine sets of rules named 1 through 9, each corresponding to a compression level. Only the rules outside any set of rules and the selected set of rules (level) are evaluated. Rules are evaluated in the order they appear in the **.ratrc** file. The first rule to match is used to select the algorithm to use for each file.

The user can configure these sets of rules to reflect local experience with the types of files being archived. Different sets of rules can be tuned for speed, archive size, or any such desired metric. The **.ratrc** file shown in Figure 4 defines five such sets of rules that were used to generate the results shown in Table 3 below. Set 1 causes all files to be stored without compression. Sets 2, 3 and 4 causes all files to be compressed using zlib (as in gzip), bzip2, and LZO, respectively.

Set 9, which is the default rule set, uses a nontrivial set of rules. Files for which the file name ends in .rat, .gz, .bz2, .gif and .jpg are assumed to be already compressed, so attempting to re-compress them is futile. Therefore these files are stored in the archive without further compression. The bzip2 algorithm in general does better with big files than the gzip algorithm, while the opposite is true for small files. Therefore Set 9 uses the gzip algorithm to compress files less than 8192 bytes in size, while the bzip2 algorithm is used for all other files. The bzip2 algorithm is also used for files with holes.

Some compression algorithms are themselves configurable. For example, Figure 4 shows the use of the BZ2blockSize and BZ2workFact parameters which are used to control the bzip2 algorithm. Since these parameters are specified outside any set of rules, they apply to all sets. However, different values could be specified inside the rule set which would then take precedence. Using such configurable parameters allows the user to tune the algorithms to best suit the local conditions.

Finally, the example in Figure 4 also controls how to invoke PGP on the local system. The particular version shown is for PGP 5.0i. The pgps line controls how the signature is generated, while the pgpv line controls how the signature is verified. The parameters filled in on the pgps line are the user name and the file name, while only the file name is filled in on the pgpv line.

Example Implementations

Three example implementations were written. The first implementation is a simple command line style interface similar to tar. This implementation required 244 lines of code (including comments) of which only about 50 lines actively manipulate the archive.

A more elaborate implementation that allows the user to browse the archive was also written. In this implementation familiar commands such as ls, pwd and cd are used to browse and traverse the archive. The header command is used to list all the information about a file. Commands such add and delete are used to add or remove files to or from a list of files to restore. The command lsmark is used to show the list of files marked for extraction, while the extract command is then used to restore all files marked for extraction to disk.

Finally an implementation with a graphical user interface (GUI) was written using the Qt library. This implementation shows the files in the archive as a tree structure with boxes next to each file name. Boxes can be checked to specify that files are to be extracted. Clicking on the box next to a directory marks or unmarks all files in the directory and subdirectories. This implementation required only 390 lines of C++ code. Figure 5 shows a screen shot of this implementation.

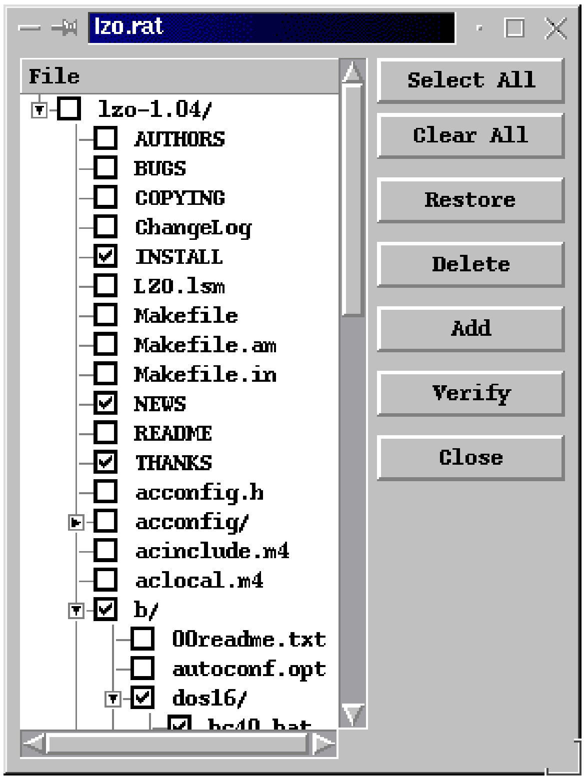

Figure 5: Qt rat.

All the implementations use calls to rat_open_toc and rat_next_toc to build a tree structure of the TOC. For detailed file listings calls to rat_inq_file are used. To translate the UID and GID to user and group names, calls are made to

```
const char* rat_inq_uid(rat_t rat,
                        u_int32_t uid)
const char* rat_inq_gid(rat_t rat,
                        u_int32_t gid)
```

Once the user has selected a list of the files to extract, repeated calls are made to rat_read_file to restore the files.

Results

The **rat** implementation was tested on a mixed data set. Statistics for the data are shown in Table 2. The total size of all the files is 236.5 MB. Note that the standard deviation is about ten times the average file size, and the largest file is 36 MB. The files are a mix of C source code files, object files, ELF binaries and postscript files. Tables 3a and 3b compares the results obtained with **rat**, **tar** and **zip**.

With no compression, the **rat** archive is only 0.3 MB larger than the total file size for 3186 files, while the **tar** archive is 2.2MB larger than the total file size. Creating the archive and restoring the entire archive takes a bit longer with rat, but **rat** retrieves a single file **much** faster than **tar**. It is clear that the **rat** file store and retrieve functions need to be optimized.

When using gzip, bzip2 or LZO to compress the archive the **rat** and **tar** results are very similar as far as the archive size, creation time and full restoration time are concerned. Support for gzip is incorporated into tar. bzip2 compression is achieved by piping the output through the bzip2 utility, while decompression is achieved by processing the output from the bzcat utility. LZO compression and decompression is similarly achieved by piping through the lzop utility. When using gzip, bzip2 or LZO to compress the entire **tar** archive, the compression is theoretically more efficient than if each file is compressed individually as in **rat**. This is borne out by Table 3, but the increase in size is only a few percentage points. In all cases, however, almost instantaneous access to an individual file is achieved with **rat** while **tar** can take several minutes.

The **zip** utility uses an algorithm very similar to gzip, but also does per file compression. Table 3 shows that the resulting file size is slightly larger than the **rat** file, but better performance than either **tar** or **rat** when creating the archive. When restoring a single file, the performance of **rat** and **zip** are similar, but **zip** is slightly faster than **rat** when restoring all the files.

Note that the mixed set of rules that uses gzip on files less than 8192 bytes and bzip2 on larger files yields a slight improvement in archive size. Defining optimal sets of rules for file size remains a task for the future.

Finally, the data was archived and signed using a 1024 bit PGP key. This adds 66 bytes per file so the resulting archive is only slightly larger than the archive without signatures. However, creating and checking the signatures takes a considerable amount of time. This extra time is attributable to both the actual work of creating or checking the signature as well as spawning a new process to run PGP for every file.

Adding a single new file to the archive with **rat** takes less than a second and does not involve copying the archive. Adding a single new file to a **zip** or **tar** archive takes quite a bit longer, probably because of the time it takes to copy the archive. A file cannot be added to a compressed tar archive. The whole archive must first be uncompressed, the file added, and then the whole archive recompressed.

Deleting a single file from the archive with **rat** or **zip** takes approximately the same time and is quite fast. **tar** does not support deleting a file a file from the archive.

Conclusions

The three sample implementations showed that **librat** is easy to use and provides sufficient functionality to construct a sophisticated user interface. The sets of rules in the **.ratrc** file proved to be very useful during the performance tests.

Number of Files	3186
Total Size	236.5 MB
Average Size	76.0 kB
Std.Dev. Size	749.6 kB
Maximum Size	36.1 MB

Table 2: Test Data.

The results shown in Table 3 and other tests performed with this implementation proved that archives could be produced that are comparable in size to those with **tar** or **tar** and **gzip**, **bzip2** or LZO for about the same amount of effort on the archive creation time, but with dramatically improved extraction speed for individual files. In terms of performance, rat compares well to **zip**.

The ability to sign and verify files proved to be very useful and adds little to the file size but is rather slow. The major advantage of signatures over encryption proved to be that usable files can be extracted even when signature verification software is not available.

Future Work

The **.ratrc** file allows very flexible specification of the compression method to be used on each file. A good heuristic to select a near optimal set of rules for general use remains to determined. It is likely that the type of rules may have to be expanded with such a heuristic. In particular, the rules should be expanded to check the file's magic as done by the file utility.

A major drawback of externally executing the signature creation and verification program is the poor performance. If the signature code (as opposed to the encryption code) could be build directly into **librat**, this should significantly alleviate the performance problems.

Acknowledgements

The authors wish to thank the instructor and teaching assistants for the System Administration course at the University of Colorado at Boulder during the Fall of 1999, Evi Nemeth, Vega Paithankar, Josh Prismon and Ali Rayl for their support, and Rob Braun for suggesting and guiding the project.

Availability

The initial distribution version of **rat** is rat-0.1. It consists of **librat**, the simple command line front end and the Qt front end. It is available for download from http://www.cs.colorado.edu/~vlakkies/ .

Program	Compression	Size (MB)	Build Time (sec)	Extract One File (sec)	Extract All Files (sec)
tar	none	238.7	39	23	65
rat	none	236.8	74	<1	80
tar	gzip	77.7	262	27	62
rat	gzip	78.7	294	<1	72
zip	deflate	78.8	222	<1	67
tar	bzip2	67.3	1011	186	206
rat	bzip2	72.3	1090	<1	210
tar	LZO	104.3	61	16	75
rat	LZO	105.5	89	<1	69
rat	mixed	72.0	1064	<1	209
rat	gzip+PGP	78.7	10861	1	1689

Table 3a: **rat**, **tar** and **zip** performance.

Program	Add (sec)	Delete (sec)
tar	38	N/A
zip	24	15
rat	<1	7

Table 3b: **rat**, **tar** and **zip** performance.

Author Information

Vlakkies Schreüder is a consulting engineer, software developer and system administrator with Principia Mathematica, and a full time student. He holds a Ph.D in Computational Fluid Dynamics from the University of Stellenbosch and is currently working on a second Ph.D in Parallel Systems at the University of Colorado at Boulder. Contact him at <vlakkies@colorado.edu>.

Maria Murillo holds an MS in Geophysics from the University of Tulsa. She is currently a full time student pursuing a Ph.D in Computer Science at the University of Colorado at Boulder. Contact her at <murillo@colorado.edu>.

References

J. Seward, bzip2, a block-sorting file compressor, http://www.bzip2.org .

J. Gailly and M. Adler, GNU Zip, http://www.gzip.org/ .

M. F. X. J., Oberhumer, LZO – a real-time data compression library, http://wildsau.idv.uni-linz.ac.at/mfx/lzo.html .

P. Zimmerman, Pretty Good Privacy, http://www.pgpi.com/ .

AT&T, tape file archiver, http://ftp.digital.com/pub/GNU/tar/ .

M. Adler, R. B. Wales, J. Gailly, G. Roelofs, O. van der Linden and K. U. Rommel, Info-ZIP, http://www.cdrom.com/pub/infozip/ .

J. Gailly and M. Adler, zlib, a general purpose data compression library, http://www.cdrom.com/pub/infozip/zlib/ .