Optimizing eCryptfs for better performance and security

Li Wang
School of Computer
National University of Defense Technology
liwang@nudt.edu.cn

Yunchuan Wen
Kylin, Inc.
wenyunchuan@kylinos.com.cn

Jinzhu Kong
School of Computer
National University of Defense Technology
kongjinzhu@163.com

Xiaodong Yi
School of Computer
National University of Defense Technology
xdong_yi@163.com

Abstract

This paper describes the improvements we have done to eCryptfs, a POSIX-compliant enterprise-class stacked cryptographic filesystem for Linux. The major improvements are as follows. First, for stacked filesystems, by default, the Linux VFS framework will maintain page caches for each level of filesystem in the stack, which means that the same part of file data will be cached multiple times. However, in some situations, multiple caching is not needed and wasteful, which motivates us to perform redundant cache elimination, to reduce ideally half of the memory consumption and to avoid unnecessary memory copies between page caches. The benefits are verified by experiments, and this approach is applicable to other stacked filesystems. Second, as a filesystem highlighting security, we equip eCryptfs with HMAC verification, which enables eCryptfs to detect unauthorized data modification and unexpected data corruption, and the experiments demonstrate that the decrease in throughput is modest. Furthermore, two minor optimizations are introduced. One is that we introduce a thread pool, working in a pipeline manner to perform encryption and write down, to fully exploit parallelism, with notable performance improvements. The other is a simple but useful and effective write optimization. In addition, we discuss the ongoing and future works on eCryptfs.

1 Introduction

eCryptfs is a POSIX-compliant enterprise cryptographic filesystem for Linux, included in the mainline Linux kernel since version 2.6.19. eCryptfs is derived from Cryptfs [5], which is part of the FiST framework [6]. eCryptfs provides transparent per file encryption. After mounting a local folder as an eCryptfs folder with identity authentication passed, the file copied into the eCryptfs folder will be automatically encrypted. eCryptfs is widely used, for example, as the basis for Ubuntu's encrypted home directory, natively within Google's ChromeOS, and transparently embedded in several network attached storage (NAS) devices.

eCryptfs is implemented as a stacked filesystem inside Linux kernel, it does not write directly into a block device. Instead, it mounts on top of a directory in a lower filesystem. Most POSIX compliant filesystem can act as a lower filesystem, for example, ext4 [2], XFS [4], even NFS [3]. eCryptfs stores cryptographic metadata in the header of each file written, so that encrypted files can be copied between hosts, and no additional information is needed to decrypt a file, except the ones in the encrypted file itself.

The rest of this paper is organized as follows. For background information, Section 1 introduces the eCryptfs cryptographic filesystem. Section 2 describes the optimizations for eCryptfs performance. Section 3 presents the data integrity enforcement for eCryptfs security. Section 4 discusses our ongoing and future works on eCryptfs. Section 5 concludes the paper.

2 Performance Improvements

2.1 Redundant Cache Elimination

The page cache is a transparent filesystem cache implemented in Linux VFS (Virtual File System) framework.

The fundamental functionality is to manage the memory pages that buffer file data, avoiding frequent slow disk accesses. For eCryptfs (and other stacked file systems in Linux), there exist (at least) two levels of page caches, as shown in Figure 1. The upper level is the eCryptfs page cache, which buffers the plain text data to interact with the user applications. The lower level page cache belongs to the file system on top of which eCryptfs is stacked, and it buffers the cipher text data as well as the eCryptfs file metadata. The eCryptfs file read/write works as follows,

- Read operations result in the call of VFS `vfs_read`, which searches the eCryptfs page cache. If no matching page is found, `vfs_read` calls the file system specific `readpage` call back routine to bring the data in. For eCryptfs this is `eCryptfs_readpage`, which calls `vfs_read` again to cause the lower file system to read the data from the disk into eCryptfs page cache. The data is then decrypted and copied back to the user application.
- Write operations result in the call of VFS `vfs_write`, which copies the data from user space buffer into the eCryptfs page cache, marks the corresponding page dirty, then returns without encryption (unless the system currently has large amount of dirty pages). Encryption is normally performed asynchronously by the dedicated OS kernel thread, during the job of flushing dirty pages into lower page cache, by invoking file system specific `writepage` call back routine, here is `ecryptfs_writepage`. This routine encrypts a whole page of data into a temporary page, then invokes `vfs_write` to copy the encrypted data from the temporary page into the lower page cache.

In real life, eCryptfs is often deployed in archive and backup situations. For the former case, people archive their private documents into an eCryptfs protected directory, consequently, the corresponding eCryptfs files are created and opened for writing, and those files are later opened generally for reading. The latter case is similar, user copies files out from eCryptfs folder, modifies, and copies the revised files back to replace the original ones. In this case, the eCryptfs files are opened for either reading or writing as well, in other words, in the above situations, the files are almost never online edited, i.e., opened for both reading and writing. This is also true for some other situations.

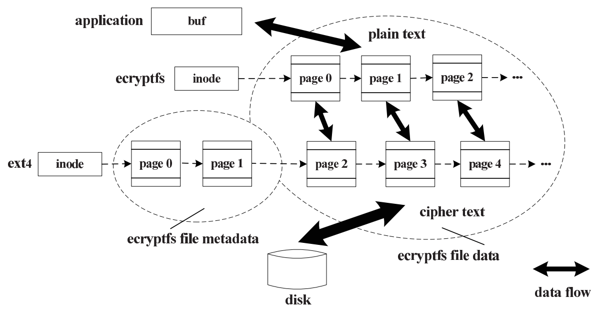

Figure 1: Page caches for eCryptfs on top of ext4 under original implementation.

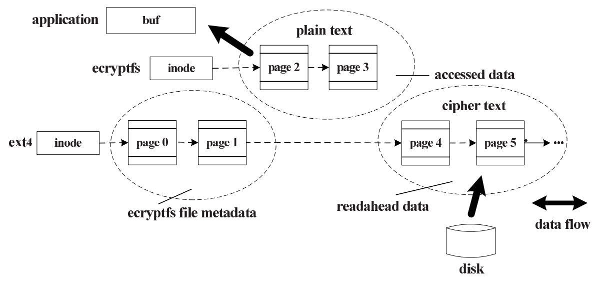

Figure 2: Page caches for eCryptfs on top of ext4 under read optimization for encrypted file.

If the eCryptfs file is opened only for reading, the lower page cache is not needed since the cipher data will not be used after decryption. This motivates us to perform redundant cache elimination optimization, thus reduce ideally half of the memory consumption. The page cache is maintained only at the eCryptfs level. For first read, once the data has been read out from disk, decrypted and copied up to eCryptfs page cache, we free the corresponding pages in the lower page cache immediately by invoking `invalidate_inode_pages2_range`, as shown in Figure 2. A better solution is to modify the VFS framework to avoid allocating the lower page entirely, but that would be much more complicated, and we want to limit our revisions in the scope of eCryptfs codes.

If the eCryptfs file is opened for writing, and the write position is monotonic increasing, which guarantees the same data area will not be repeatedly written, then the eCryptfs page cache is not needed since the plain data will not be used after encryption. It is beneficial to maintain the page cache only at the lower level. Once the data has been encrypted and copied down to lower page cache, the corresponding pages in eCryptfs page cache are freed immediately.

```
enum efsrwstate {
   ECRYPTFS_RW_INIT,
   ECRYPTFS_RW_RDOPT,
   ECRYPTFS_RW_WROPT,
   ECRYPTFS_RW_NOOPT,
};
struct ecryptfs_rw_state {
   struct mutex lock;
   efsrwstate state;
};
struct ecryptfs_inode_info {
   ...
   struct ecryptfs_rw_state rw_state;
 };
```

Figure 3: The data structures for redundant cache elimination for encrypted file.

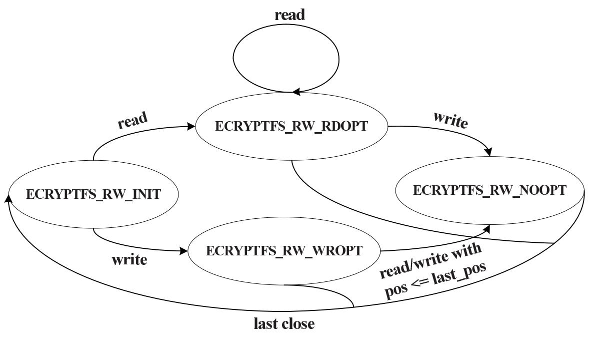

Figure 4: The state machine for redundant cache elimination for encrypted file.

To achieve this optimization, we maintain a simple state machine as an indicator. There are four states as shown in Figure 3, `ECRYPTFS_RW_INIT` indicates an initialized state, with neither readers nor writers. `ECRYPTFS_RW_RDOPT` indicates the file is currently opened only for reading, and the read optimization, i.e, lower page cache early release applies. `ECRYPTFS_RW_WROPT` indicates the file is currently opened only for writing and the write position is monotonic increasing, in which case, the write optimization, i.e, upper page cache early release applies. `ECRYPTFS_RW_NOOPT` applies for remaining cases. As shown in Figure 4, when a `file` data structure is initialized, the state is set to `ECRYPTFS_RW_INIT`. If it is then opened for reading, the state transitions to `ECRYPTFS_RW_RDOPT`, and remains unchanged until a

```
1  static int ecryptfs_readpage(struct file * file,
                                        struct page *page)
2  {
3     …
4     flags = inode_info->crypt_stat.flags.
5     /* skip non-encrypted file */
6     if (!(flags & ECRYPTFS_ENCRYPTED))
7        goto out;
8     mutex_lock(&inode_info->rw_state.lock);
9     state = inode_info->rw_state.state;
10    if (state == ECRYPTFS_RW_RDOPT) {
11       pgoff_t index;
12       /* calculate the lower page index */
13       index = (ecryptfs_lower_header_size(
            crypt_stat) >> PAGE_CACHE_SHIFT)
                        + page->index;
14       /* release the lower page */
15       invalidate_inode_pages2_range(
            lower_inode->i_mapping, index, index);
16    }
17    mutex_unlock(&inode_info->rw_state.lock);
18    out:
19    ...
20  }
```

Figure 5: The `readpage` routine for eCryptfs for encrypted file.

writer gets involved. In that case, the state changes from `ECRYPTFS_RW_RDOPT` to `ECRYPTFS_RW_NOOPT`. If the file is initially opened for writing, the state becomes `ECRYPTFS_RW_WROPT`, and remains unchanged until the write position decreases or any reader gets involved, in which case, the state becomes `ECRYPTFS_RW_NOOPT`. After the file is closed by the last user, the state returns to `ECRYPTFS_RW_INIT`.

Figure 5 shows the eCryptfs code implementing `readpage`. After the data have been copied into eCryptfs page cache, and redundant cache elimination is applicable, in line 13, the corresponding lower page index is calculated, then in line 15, the lower page is released.

We evaluate redundant cache elimination on a machine with a Pentium Dual-Core E5500 2.8GHz CPU, 4G of memory (DDR3), the kernel version 3.1.0-7 i686 with PAE enabled, the same test system is used for all the experiments of this paper described. We use the command '`iozone -t` *x* `-s` *y* `-r 4M -i 1`' to measure 'Re-read' throughput. Here *x* ranges from 1 to 256, increasing by orders of 2, representing the number of processes, and *y* ranges from 2G to 8M, representing the file size, such

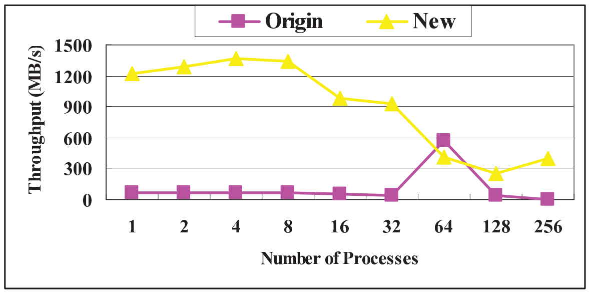

Figure 6: Re-read throughput for redundant cache elimination over original implementation.

that the total operated data set $x * y$ is always equal to 2G.

As shown in Figure 6, the optimized eCryptfs achieves a big speedup under this group of tests. This is because the data set is 2G, if two level of page caches are present, it will consume up to 4G memory, thus lead to a poor re-read throughput due to memory page thrashing. With the redundant cache elimination, the memory is enough to buffer the data set, therefore brings a good throughput.

There is furthermore a special case when the eCryptfs folder is mounted with `ecryptfs_passthrough` flag set, which allows for non-eCryptfs files to be read and written from within an eCryptfs mount. For those non-eCryptfs files, eCryptfs just relays the data between lower file system and user applications without data transformations. In this case, the data will be double cached in each level of page cache, and the eCryptfs page cache is not needed at all. For this case, we directly bypass the eCryptfs page cache by copying data between lower file system and user buffer directly.

For example, Figure 7 shows the codes implementing `read`. In line 14, `vfs_read` is invoked with the lower file and user buffer as input, this will read data from lower file system directly to user buffer. In line 19, the eCryptfs inode attributes are updated according to the ones of the lower inode.

Linux as well as most other modern operation systems provide another method to access file, namely the memory mapping by means of the `mmap` system call. This maps an area of the calling process' virtual memory to files, i.e, reading those areas of memory causes the file to be read. While reading the mapped memory areas for the first time, an exception is triggered and the architecture dependent page fault handler will call file system

```
1  static ssize_t ecryptfs_read(struct file *file,
                char *buf,  size_t count, loff_t *ppos)
2  {
3     int err;
4     struct file *lower_file;
5     loff_t pos_copy = *ppos;

6     /* skip encrypted file */
7     if (ecryptfs_file_to_private(file)->crypt_stat->
             flags & ECRYPTFS_ENCRYPTED)
8       goto out;
9     lower_file = ecryptfs_file_to_lower(file);
10    /*
11     * directly read data into the user buffer from
12     * the lower file, bypass the ecryptfs page cache
13     */
14    err =  vfs_read(lower_file, buf, count, &pos_copy);

15    /* update the ecryptfs inode attributes */
16    lower_file->f_pos = pos_copy;
17    *ppos = pos_copy;
18    if (err >= 0) {
19      fsstack_copy_attr_atime(file->f_dentry->d_inode,
                           lower_file->f_dentry->d_inode);
20    }

21    return err;
22  }
```

Figure 7: The `read` routine for eCryptfs for non-ecrypted file.

```
const struct file_operations ecryptfs_main_fops = {
   ...
   .mmap = ecryptfs_file_mmap,
 };

 static int ecryptfs_file_mmap(struct file *file,
                     struct vm_area_struct *vma)
 {
     ...
     vma->vm_file = lower_file;
     rc = lower_file->f_op->mmap(lower_file, vma);
     get_file(lower_file);
     new_ops = &inode_info->ecryptfs_vm_ops;
     mutex_lock(&inode_info->lower_file_mutex);
     if (!inode_info->lower_vm_ops) {
        inode_info->lower_vm_ops = vma->vm_ops;
        new_ops.fault = ecryptfs_vma_fault;
        ... // the assigns to other interfaces omitted
      }
     mutex_unlock(&inode_info->lower_file_mutex);
     vma->vm_ops = new_ops;
     vma->vm_private_data = file;

     return rc;
  }
```

Figure 8: The `mmap` routine for eCryptfs for non-encrypted file.

```
static int ecryptfs_vma_fault(struct vm_area_struct *
                              vma, struct vm_fault *vmf)
{
    file = vma->vm_private_data;
    inode = file->f_dentry->d_inode;
    inode_info = ecryptfs_inode_to_private(inode);

    return inode_info->lower_vm_ops->fault(vma, vmf);
}
```

Figure 9: The `fault` routine for eCryptfs for non-encrypted file.

specific `readpage` routine to read the data from disk into page cache.

With regard to `mmap`, to bypass the eCryptfs page cache, the approach is shown in Figure 8. In line 8, the owner of the `vma` address space is replaced with the lower file. In line 9, the lower filesystem `mmap` implementation is called, this will assign the lower filesystem implemented operation set to the field `vm_ops` of `vma`. In line 14, this set is saved and then `vm_ops` is replaced with the eCryptfs implemented operation set in line 19, which, in most cases, simply call the saved lower operation set, and update some attributes, if necessary. For example, the implementation of `fault` interface is shown in Figure 9. By this approach, the eCryptfs page cache will not be generated.

2.2 Kernel Thread Pool based Encryption

Linux-2.6.32 introduced the feature of per-backing-device based writeback. This feature uses a dedicated kernel thread to flush the dirty memory of each storage device. eCryptfs makes use of this feature by registering an 'eCryptfs' backing device while mounting a eCryptfs folder. Therefore, a dedicated kernel thread will be generated to flush the pages dirtied by eCryptfs from the ecryptfs page cache to the lower page cache. Basically, the kernel thread achieves this goal in two steps, first encrypting the data, then writing encrypted data to lower page cache. Since encryption is CPU intensive task, and the current write back framework supports only one thread per device, it could not exploit the power of multiple CPU cores to perform parallel encryption, and will slow down the speed of submitting pages to lower page cache.

A better way is to use a kernel thread pool to do this job in a pipeline manner. Two groups of kernel threads

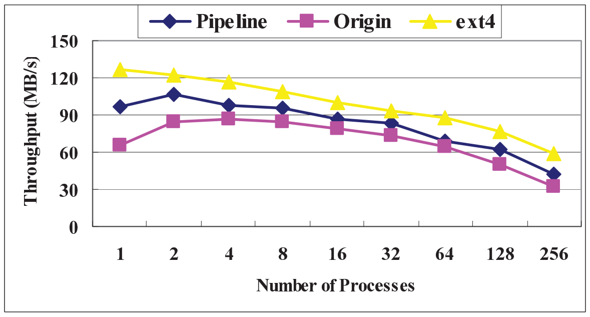

Figure 10: Write throughput for thread pool based encryption over original implementation and ext4.

are generated, one is responsible for encryption, called `e-thread`. the other is for writing data down, called `w-thread`. The number of e-thread is twice of the number of CPU cores. w-threads are spawned on-demand. `ecryptfs_writepage` submits the current dirty page to pipeline, wakes up the e-threads, then returns. The e-threads grab dirty pages from the pipeline, perform encryption, submit the encrypted pages to the next station of the pipeline, and dynamically adjust the number of w-threads according to the number of write pending pages, if necessary. w-threads write the encrypted pages to the lower page cache.

This approach is evaluated by measuring 'Write' throughput using `iozone`. For comparison, the throughput on ext4 is also tested to give an upper bound. The parameters are '`iozone -t` *x* `-s` *y* `-r 4M -i 0 -+n`', where *x* is from 1 to 256, corresponding, *y* from 8G to 32M. As shown in Figure 10, the optimized codes achieve an obviously higher throughput than the original implementation.

2.3 Write Optimization

The Linux VFS framework performs buffered writes at a page granularity, that is, it copies data from user space buffers into kernel page cache page by page. During the process, VFS will repeatedly invoke the file system specific `write_begin` call back routine, typically once per page, to expect the file system to get the appropriate page in page cache ready to be written into. For eCryptfs, the routine is `ecryptfs_write_begin`, which looks for the page cache, and grabs a page (allocates it if desired) for writing. If the page does not contain valid data, or the data are older than the counterpart on the disk, eCryptfs will read out the corresponding data from the disk into the eCryptfs page cache, decrypt

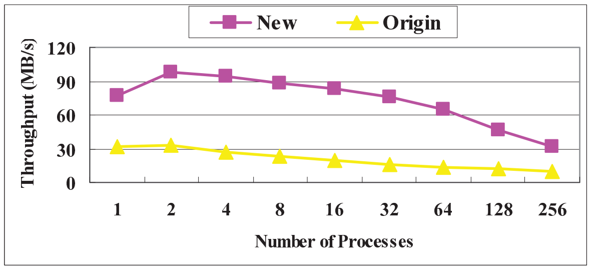

Figure 11: Write throughput for write optimization over original implementation.

them, then perform writing. However, for current page, if the length of the data to be written into is equal to page size, that means the whole page of data will be overwritten, in which case, it does not matter whatever the data were before, it is beneficial to perform writing directly.

This optimization is useful while using eCryptfs in backup situation, user copies file out from eCryptfs folder, modifies, and copies the revised file back to replace the original one. In such situation, the file in eCryptfs folder is overwritten directly, without reading.

Although the idea is simple, there is one more issue to consider related to the Linux VFS implementation. As described above, VFS calls `write_begin` call back routine to expect the file system to prepare a locked updated page for writing, then VFS calls `iov_iter_copy_from_user_atomic` to copy the data from user space buffers into the page, at last, VFS calls `write_end` call back routine, where, typically, the file system marks the page dirty and unlocks the page. `iov_iter_copy_from_user_atomic` may end up with a partial copy, since some of the user space buffers are not present in memory (maybe swapped out to disk). In this case, only part of data in the page are overwritten. Our idea is to let `ecryptfs_write_end` return zero at this case, to give `iov_iter_fault_in_readable` a chance to handle the page fault for the current iovec, then restart the copy operation.

This optimization is evaluated by measuring 'Write' throughput using `iozone`. The command parameters are '`iozone -t` *x* `-s` *y* `-r 4M -i 0 -+n`', where *x* is from 1 to 256, correspondingly, *y* is from 8G to 32M, and the files written into have valid data prior to the experiments. As shown in Figure 11, the optimized codes enjoy around 3x speedup over the original implementation under this group of tests.

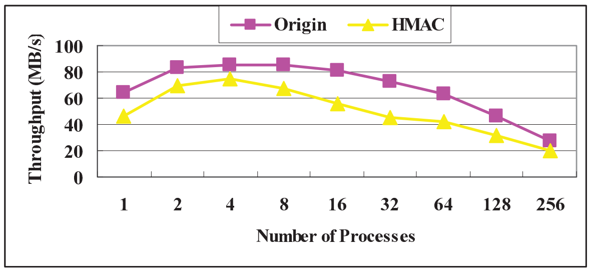

Figure 12: Write throughput for eCryptfs with HMAC-MD5 verification over the one without.

3 Data Integrity Enforcement

In cryptography, HMAC (Hash-based Message Authentication Code) [1] is used to calculate a message authentication code (MAC) on a message involving a cryptographic hash function in combination with a secret key. HMAC can be used to simultaneously verify both the data integrity and the authenticity of a message. Any cryptographic hash function, such as MD5 or SHA-1, may be used in the calculation of an HMAC; the resulting MAC algorithm is termed HMAC-MD5 or HMAC-SHA1 accordingly. The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, the size of its hash output length in bits, and on the size and quality of the cryptographic key.

We implemented HMAC verification for eCryptfs, enabling it to detect the following situations,

- Unauthorized modification of file data
- Data corruption introduced by power loss, disk error etc.

Each data extent is associated with a HMAC value. Before an extent is encrypted and written to lower file page cache, its HMAC is calculated, with the file encryption key and the data of the current extent as input, and the HMAC is saved in the lower file as well. When the data is later read out and decrypted, its HMAC is recalculated, and compared with the value saved. If they do not match, it indicates that the extent has been modified or corrupted, eCryptfs will return an error to user application. Since the attacker does not know the file encryption key, the HMAC value cannot be faked.

The HMAC values are grouped into dedicated extents, rather than appended at end of each extent due to a performance issue. According to our test, `vfs_read` and `vfs_write` with non extent-aligned length are much slower than the aligned counterpart. Accordingly, the data extents are split into groups.

The decrease in throughput due to HMAC verification is evaluated by measuring 'Write' throughput using `iozone`, the hash algorithm is MD5. The command parameters are '`iozone -t` *x* `-s` *y* `-r 4M -i 0 -+n`', where *x* is from 1 to 256, corresponding, *y* from 8G to 32M. As shown in Figure 12, the decrease is modest.

4 Future Work

We are implementing per-file policy support for eCryptfs. That is, allow to specify policies at a file granularity. For example, a file should be encrypted or not, what encryption algorithm should be used, what is the length of the key, etc. In addition, we are considering to implement transparent compression support for eCryptfs.

Related to VFS, we are taking into account to modify VFS to give filesystems more flexibilities, to maintain page cache at their decisions.

5 Conclusion

This paper presents the optimizations to eCryptfs for both performance and security. By default, Linux VFS framework maintains multiple page caches for stacked filesystems, which, in some situations, is needless and wasteful, motivating us to develop redundant cache elimination, the benefits of which have been verified experimentally. Furthermore, this approach is applicable to many other stacked filesystems. We enhance the eCryptfs security by introducing HMAC verification. To exploit parallelism, we introduce a thread pool, working in a pipeline manner to do the encryption and write down jobs. We have also presented a simple write optimization, which is very useful while using eCryptfs in backup situation.

References

[1] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 1–15, London, UK, 1996. Springer-Verlag.

[2] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Linux Symposium*, 2007.

[3] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *USENIX Conference*, 1985.

[4] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Conference*, 1996.

[5] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, 1998.

[6] E. Zadok and J. Nieh. Fist: A language for stackable file systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, 2000. USENIX Association.