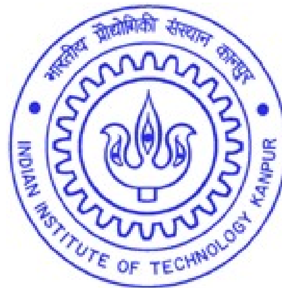
File System Independent Metadata Organization for TransCrypt

Arun Raghavan



Department of Computer Science & Engineering

Indian Institute of Technology Kanpur

June 2008

File System Independent Metadata Organization for TransCrypt

A Thesis Submitted

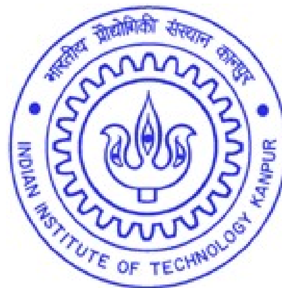
In Partial Fulfillment of the Requirements

For the Degree of

Master of Technology

by

Arun Raghavan



to the

Department of Computer Science & Engineering

Indian Institute of Technology Kanpur

June 2008

Certificate

This is to certify that the work contained in the thesis entitled *"File System Independent Metadata Organization for TransCrypt"*, by *Arun Raghavan*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Prof. Rajat Moona)
Department of Computer
Science & Engineering,
Indian Institute of Technology
Kanpur,
Kanpur, Uttar Pradesh 208016

(Prof. Dheeraj Sanghi)
Department of Computer
Science & Engineering,
Indian Institute of Technology
Kanpur,
Kanpur, Uttar Pradesh 208016

Abstract

With the wide-spread adoption of digital information storage, the problem of data theft has come to be of utmost importance to both individuals and organizations. While several encrypting file systems have been developed to address this problem, only a few are scalable enough to be deemed ready for enterprise use, and even these suffer from a variety of issues, ranging from poor performance to inadequate trust models. The TransCrypt encrypting file system was developed to address these problems, and provide a secure, scalable and efficient enterprise-class solution to the data security problem. It uses a flexible, yet secure, key management scheme that simplifies sharing of files without compromising on security. One of TransCrypt's distinguishing features is that the super user is not considered a trusted entity in its threat model.

A prototype of TransCrypt, based on the `ext3` file system, had been implemented on the Linux 2.6 kernel. In this prototype, the code was tightly coupled with the `ext3` file system and modified its on-disk structures to store some additional metadata. This implied that TransCrypt could not be used with other, more advanced, file systems without modifying their code as well. Moreover, modifications were made to existing code, both in the Linux kernel as well as in userspace utilities, introducing the additional overhead of code maintenance. In this work, we have addressed this limitation of the TransCrypt design. TransCrypt has now been redesigned to employ a layered architecture, with its layers being file system-independent. This was made possible by the use of the extended attribute mechanism for metadata storage, which is provided by several modern file systems. As a result of this work, TransCrypt can now be used as a cryptographic layer over any of these file systems.

Acknowledgments

I wish to express my gratitude to Prof. Rajat Moona and Prof. Dheeraj Sanghi, without whom this thesis would not have been possible. Their constant support, advise and guidance made it a pleasure to work on this project. I also thank the Prabhu Goel Research Center for Computer and Internet Security and Tata Consultancy Services, Lucknow, for partially supporting me and providing me with the freedom to devote myself completely to this project.

I also thank Sainath Vellal and Satyam Sharma for their help and suggestions throughout my work. Thanks are also due to Mayur Shardul for patiently listening to me and helping me flesh out my ideas. I also thank my entire class, the M.Tech. 2006 batch of CSE — it was a pleasure to be surrounded by a set of such helpful, fun-loving and talented individuals.

Finally, I thank my family and coterie of close friends whose love, support and encouragement made all of this possible.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Related Work 3

1.2.1 dm-crypt with LUKS 4

1.2.2 eCryptfs 5

1.3 TransCrypt 6

1.4 Contribution of this Thesis 8

1.5 Organization 9

2 Background **10**

2.1 The Linux VFS 10

2.2 Metadata Storage 12

2.2.1 On Disk Storage 12

2.2.2 In Memory Storage 14

2.3 Access Control 16

2.3.1 UNIX Permissions 17

2.3.2 POSIX Access Control Lists 17

2.4 Extended Attributes 20

2.4.1 Implementations 21

2.4.2 ACL Storage 22

3 TransCrypt Implementation **24**

3.1 In-Kernel Architecture 24

3.1.1 VFS Core 25

3.1.2 ext3 26

3.1.3 Page Cache 28

3.1.4 Keyring 28
3.1.5 Communication with Userspace 29
3.1.6 CryptoAPI 31
3.2 Userspace 31
3.2.1 libacl 32
3.2.2 e2fsprogs 33
3.2.3 Daemons 34
3.2.4 Utilities 35

4 The TransCrypt-Enhanced Virtual File System 36
4.1 Design 37
4.1.1 File System Metadata 38
4.1.2 Per-File Metadata 40
4.1.3 Extended Attribute Caveat 40
4.2 Implementation 41
4.2.1 Volume Parameters 41
4.2.2 Tokens 43
4.2.3 Userspace Utilities 44
4.2.4 Example usage 45

5 Testing 46
5.1 Test Strategy 47
5.2 Test Methodology 49
5.2.1 Python “unittest” framework 50
5.2.2 Common functionality 51
5.2.3 Implementing Test Cases 52

6 Conclusion 54
6.1 Performance Evaluation 54
6.2 Summary 58
6.3 Future Work 59

List of Figures

1.1 LUKS Partition Header 4
1.2 eCryptfs Underlying File Format 5
1.3 TransCrypt Architecture 7

2.1 The Linux VFS 11
2.2 `ext3` block group structure 13

3.1 TransCrypt and the Linux VFS 25

4.1 Modified TransCrypt and the Linux VFS 37
4.2 On-Disk EA format for TransCrypt Volume Parameters 41
4.3 On disk EA format for tokens 42

6.1 File create and open performance 57
6.2 ACL reading and modification performance 57

Chapter 1

Introduction

1.1 Motivation

Today, data security has come to be of utmost importance. Computer-based data storage is ubiquitous, even for confidential data. Large corporations and institutions usually have centralized storage devices and with an ever-increasing number of users with portable computers. A result of these two developments is that data theft has become alarmingly common [1, 2].

While several solutions that have been devised to address this problem [3, 4, 5, 6, 7, 8], these have been found wanting in certain aspects, such as strong trust models, and flexibility with regards to file sharing. A need was felt for a file system that addresses these issues. It is in this light that the TransCrypt encrypting file system [9, 10] was created.

TransCrypt is a transparent, enterprise-class, encrypting file system. Being

transparent, no changes need to be made to existing programs and tools in order to use TransCrypt. The trust model employed is stronger than most existing solutions — even the super user (`root`) is not trusted in the TransCrypt trust model. However, flexibility is not compromised — the standard UNIX semantics for sharing files between users are supported to a reasonable extent.

The TransCrypt implementation was initially carried out as a set of modifications to the `ext3` file system [11] in the Linux kernel. This included changes to the Linux kernel as well as the userspace `e2fsprogs` package [12] which contains libraries and tools to work with `ext3` volumes. As the code base grew more complex, maintaining TransCrypt code against the `ext3` code grew harder. Moreover, it was not possible to exploit more advanced file systems such as XFS [13], JFS [14], and ReiserFS [15].

Thus, there was a need to redesign certain parts of the TransCrypt implementation so as to make it independent of the underlying file system. This would greatly reduce the amount of external code that needed to be modified, significantly improving code maintainability. Moreover, the ability to use more advanced file systems with TransCrypt would allow users to benefit from these file systems' performance, reliability and other improvements without requiring substantial work on the TransCrypt code itself. This thesis describes the approach that was used to accomplish this goal.

1.2 Related Work

Several solutions have been developed to solve the data security problem using cryptographic methods. The two primary techniques that have evolved are *volume encryption* and *file system level encryption* [16]. Volume encryption involves encrypting *all* data that is to be written to the storage device before writing, and decrypting the data when it is read from the disk. There is generally a single key that is used to encrypt all data on the device. Example implementations include `dm-crypt` [4], and FileVault [6]. File system level encryption, on the other hand, works with file system objects (files and directories, for example), rather than the device as a whole. Usually, there are separate keys for data and metadata, as well as different keys for individual objects (one key per file, for example). Encryption and decryption are done separately for each object. eCryptfs [5], CFS [3], and EncFS [17] use this approach.

While volume based encryption is sufficient for standalone or small-scale use, enterprise users require greater flexibility with regards to such common features as file sharing and incremental backups. File system level encryption allows these to be incorporated without compromising on security. These solutions are usually significantly more complex than volume level encryption. For enterprise systems, the added complexity is usually deemed worth the benefit gained and this is why this model was chosen for TransCrypt. `dm-crypt` and eCryptfs were found to be the most mature solutions, though they suffer from a number of shortcomings. The following sections takes a closer look at these two file systems and their implementations, particularly with regards to how they address the problem of metadata storage.

Figure 1.1: LUKS Partition Header [20]

1.2.1 dm-crypt with LUKS

`dm-crypt` is a volume based encrypting system for Linux. It uses the Linux kernel's device mapper infrastructure to perform on-the-fly encryption/decryption of data. Users create a virtual device using the `dm-setup` tool [18], which creates a mapping in the kernel between a virtual block device that is instantiated from a specified "target" (in this case `dm-crypt`) and an underlying physical block device. The user performs subsequent disk operations (`mkfs`, `mount`, etc.) on this virtual block device. The `dm-setup` utility takes several parameters to define the cryptographic context for the encrypted file system.

Using `dm-setup` is cumbersome, because the user is forced to remember all the parameters passed to the utility every time the mapping needs to be created. The `cryptsetup` utility [19] was created as a wrapper around `dm-setup` to make these management tasks simpler, but the cryptographic context was still not persistent. As part of the LUKS [19] project, changes were made to the partition header to accommodate this metadata storage, and subsequently, changes were made to `cryptsetup` to provide an interface similar to the older mechanism.

Figure 1.1 shows the structure of metadata storage on disk in a LUKS-based `dm-crypt` partition. The LUKS `phdr` field contains such metadata as the LUKS version, the cipher used for encryption, key size, and other cryptographic parameters. The fields `KM1` through `KM8` represent key material slots. The key used to encrypt the volume is encrypted using a user-supplied passphrase (which is converted to a key

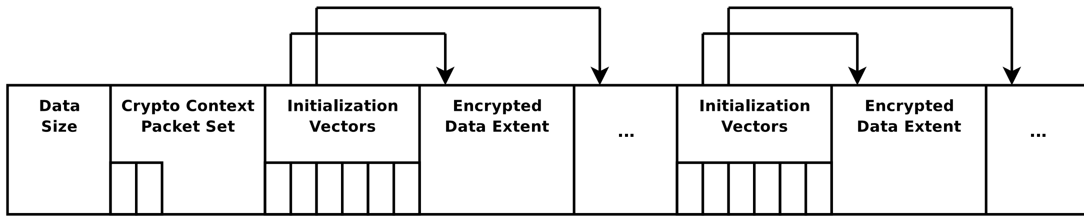


Figure 1.2: eCryptfs Underlying File Format [5]

using standard cryptographic techniques [21]) and stored in one of these slots. LUKS allows multiple copies of the key, encrypted with different passphrases, to be stored in these slots, thus permitting the volume to be shared between users.

1.2.2 eCryptfs

eCryptfs [5] is another encrypting file system in Linux, based on the Cryptfs [22] file system created by Erez Zadok, et al. For each file that is created, a random file encryption key (FEK) is generated, which is used to encrypt that file's data. The FEKs are stored on disk along with the respective files, after being encrypted using a userspace daemon.

eCryptfs has been implemented as a stackable file system — it is mounted on top of a mounted file system, and acts as a layer over this underlying file system. All file system operations are given to the eCryptfs layer which performs key management and encryption/decryption for the data before transferring the operation to the underlying file system.

Two methods of key and metadata storage are used. The first, more common method, involves changes to the file structure on the underlying file system as illustrated in figure 1.2. The file contains a header which provides the cryptographic

context for decrypting the data for any user with sufficient privileges to access the file. The file is divided into "extents", each of which is treated as a separate cryptographic unit, with its own initialization vector (IV). The IVs for a set of extents are stored in one preceding extent. These are followed by another set of IVs and the corresponding extents, and so on till the end of the file.

The second method of storage, which was introduced recently, involves storing the additional metadata (cryptographic context and IVs) in extended attributes (see section 2.4) if the underlying file system supports them and is mounted with the appropriate parameters.

1.3 TransCrypt

The overall TransCrypt architecture is illustrated in figure 1.3. The bulk of TransCrypt's functionality has been implemented in the kernel so as to be able to exclude `root` from the trust model. Userspace utilities are required for certain key management tasks.

When a user first creates a file, a random file encryption key (FEK) is generated for it. This is "blinded" with the file system key (FSK), which is a file system-specific key, known only to the kernel (and the administrator, who provides it). The user's public key is obtained in the form of an X.509 [23] certificate through the `transcryptd` daemon. The blinded FEK is then encrypted with the user's public key to create a "token", which is stored along with the file's metadata.

When the file is subsequently opened for reading or writing, the token is re-

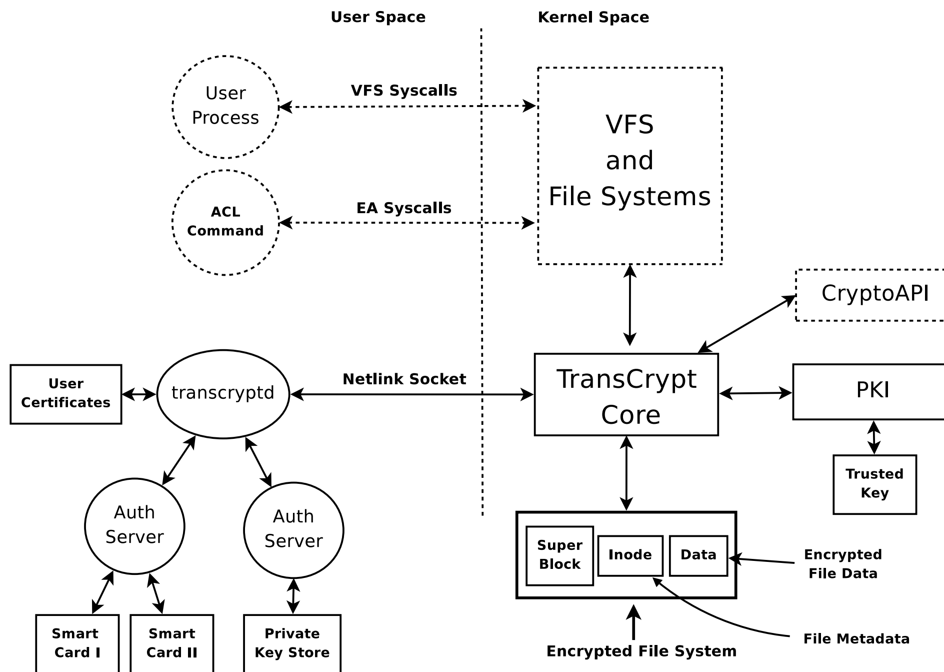


Figure 1.3: TransCrypt Architecture

trieved and sent to `transcrypt-auth` (the auth server) via `transcryptd`. `transcrypt-auth` is basically just an interface to the user's secure private key store (PKS). The token is decrypted at the PKS using the user's private key and then sent back to the kernel on the reverse path. The kernel then decrypts the FEK using the FSK and stores the FEK for use while reading from and writing to the file.

A file is shared in a similar manner. When a user a grants access to another user b for some file, the blinded FEK for that file is retrieved in a manner similar to when the file is opened. This is then encrypted with user b's public key, as was done during the initial token creation. This token is subsequently also written to the file's metadata along with the first token.

Since the blinded FEK is encrypted with appropriate users' private keys, no unauthorized user can decrypt the token. Having the FEK blinded using the FSK,

ensures that only the kernel can obtain the FEK for a file, and thus the contents of the file from the disk. In this manner, TransCrypt guarantees that only authorized users can access a file, and no user (including `root`) can even deduce the FEK of a file unless a brute force approach is adopted.

1.4 Contribution of this Thesis

A large part of TransCrypt's initial implementation involved changes to the `ext3` [11] file system. The changes were inherently bound to the `ext3` file system code in the form of changes to `ext3`'s on-disk storage format and in-memory data structures. The changes primarily involved the storage of the additional metadata required by TransCrypt, and corresponding changes to the userspace `e2fsprogs` and `acl` packages [24]. This approach presented several drawbacks, such as having to maintain TransCrypt code against a changing `ext3` code base (both in-kernel, as well as userspace utilities), as well as being unable to exploit advances in more recent file systems such as XFS, JFS, ReiserFS and `ext4` [25].

This thesis presents a method to make TransCrypt agnostic of the underlying file system. This is accomplished by moving all TransCrypt-specific metadata from file system-specific metadata storage to the more generic extended attributes [24] mechanism. The implication of this new implementation is that TransCrypt can now be used on any file system which supports extended attributes (most modern file systems do). In addition to giving the user a great deal of flexibility, this greatly enhances code maintainability since modifications to particular file system's code or existing userspace packages are no longer required.

1.5 Organization

This thesis is organized in the following manner — the motivation for TransCrypt in general and this work in particular, pointers to related work, and a brief discussion on what this work has achieved are presented in Chapter 1. In Chapter 2 the background required for reading subsequent sections is introduced. In Chapter 3, TransCrypt's overall design and implementation are described in detail. The design and implementation of this work, including a description of the previous system, its drawbacks, and how these are addressed by the new implementation are dealt with in Chapter 4. The testing methodology is discussed in Chapter 5, and the thesis is concluded in Chapter 6 with a performance evaluation and notes on future work.

Chapter 2

Background

The Linux VFS mechanism was modified and used for implementing the TransCrypt file system. In particular, modifications were made to permissions handling functionality, the `ext3` file system. and the page cache. The following sections cover these changes in detail.

2.1 The Linux VFS

The Linux kernel supports a very large number of file systems. Each file system implements its own disk storage structures and format, and sometimes its own caching policies as well. However, since almost all file systems present a standard UNIX files-and-directories hierarchy as their interface, there is a large amount of common functionality amongst them. The Linux Virtual File System (VFS) abstracts out this common functionality so that underlying file systems only need to implement the specific features that are unique to themselves. The architecture of the Linux VFS is

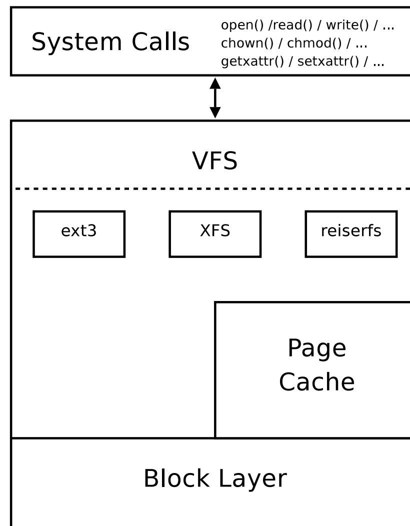


Figure 2.1: The Linux VFS

illustrated in figure 2.1.

The following functionality is implemented by the VFS.

- A standard interface for common file system operations (create, open, read, write, permissions manipulation, path name lookup, etc.)
- Common book-keeping functions such as maintaining state for each mounted file system and each file that is currently open and mapping each object to its underlying file system
- APIs to perform common tasks such as checking permissions for access control
- The `dentry` cache for speeding up directory lookups
- Callbacks on various file system operations (`inotify`/`dnotify`)

Interaction between userspace and the kernel begins when a process makes a system call. The system call results in a call to the corresponding VFS function. The VFS

passes this on to the appropriate file system-specific code, which implements the required functionality (the file system-specific code in turn might use some common VFS methods for tasks such as permissions handling).

The page cache is used extensively to enhance file system performance. For example, when reading data, the page cache is first looked up and a disk read is performed only if the required data is not present in the page cache. Similarly, when data is written, the corresponding page in the page cache is merely marked "dirty". A kernel-space daemon periodically flushes dirty pages to disk via the block layer, which writes out the data using the underlying storage driver.

2.2 Metadata Storage

In addition to data storage, file systems have to deal with metadata, such as file size, creation date and time, `uid`, `gid`, etc. These are stored on disk, as well as in memory, when the file is being used. The metadata can be global, for the file system as a whole, as well as local to each file.

2.2.1 On Disk Storage

While different file systems employ different formats for metadata storage, there is a large amount of commonality in data and metadata storage and layout on disk. The example of the `ext3` file system is presented here. Other file systems usually follow a similar pattern, but use different structures such as B+ trees.

`ext3` divides a partition into *block groups*, after leaving some space at the

Figure 2.2: `ext3` block group structure

beginning of the partition for other data such as the boot loader. The structure of a block group is illustrated in figure 2.2. Each block group contains a copy of important metadata and control structures. This redundancy greatly aids in improving file system reliability since this information is critical to the working of the file system.

The *superblock* contains block group information (like the number of blocks, inodes per group and the group number of this block group), inode management information (such as the number of inodes, and the free inode count), block management information (such as the free and reserved block count) and several other bits of metadata (like the volume name, pointer to the journal, and number of times the file system has been mounted since the last `fsck`). The on-disk superblock is represented by the `struct ext3_super_block` structure shown below.

```
struct ext3_super_block {
/*00*/  __le32  s_inodes_count;         /* Inodes count */
        __le32  s_blocks_count;         /* Blocks count */
        __le32  s_r_blocks_count;       /* Reserved blocks count */
        __le32  s_free_blocks_count;    /* Free blocks count */
/*10*/  __le32  s_free_inodes_count;    /* Free inodes count */
        __le32  s_first_data_block;     /* First Data Block */
        __le32  s_log_block_size;       /* Block size */
        ... several other members ...
        __u32   s_reserved[190];        /* Padding to the end of */
                                        /* the block */
};
```

The *file system descriptors* describe the block group, specifically the starting blocks for each of the remaining sections. The *block bitmap* keeps track of free blocks

in the group, and the *inode bitmap* does the same for free inodes. The actual inodes are stored in the *inode table* and finally the data blocks allocated to the inodes are stored in the remaining part of the block group.

Per-file metadata is stored in inodes. Each inode in the inode table contains information such as the mode (permission bits), the `uid` and `gid` of the file owner, the size of the file, creation and modification times, and pointers to data blocks. This information is stored in the `struct ext3_inode` structure shown below.

```
struct ext3_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_file_acl;     /* File ACL (actually xattr) */
        ... several other members ...
};
```

Additional metadata can be attached to an inode (and thus to the corresponding file) using extended attributes, as described in section 2.4.

2.2.2 In Memory Storage

Once the superblock and inodes have been read by the file system (while mounting and opening, respectively), common metadata is also stored in VFS structures for book-keeping. Changes to this metadata are made in memory and periodically flushed

to disk. Additionally, these structures also contain pointers to methods from the underlying file system to perform various operations.

The `struct super_block` structure contains common file system-specific information as given below.

```
struct super_block {
        /* List of all super_blocks */
        struct list_head        s_list;
        unsigned long           s_blocksize;
        struct file_system_type *s_type;
        struct xattr_handler    **s_xattr;
        /* all inodes */
        struct list_head        s_inodes;
        /* dirty inodes */
        struct list_head        s_dirty;
        /* Underlying device */
        struct block_device     *s_bdev;
        /* Filesystem private info */
        void                    *s_fs_info;
        /* Pointers to file system operations */
        const struct super_operations   *s_op;
        ... several other members ...
};
```

The operations that are provided by the underlying file system include functions for reading and writing the superblock, inode creation and deletion, etc.

The `struct inode` structure, which represents the in-memory (or in-core) inode, is intrinsically tied to both the VFS, the underlying file system and the memory subsystem (via the page cache). It stores several common pieces of metadata from the disk, as well as pointers to the owning `struct superblock`, the memory management structure (`struct address_space`) for its pages, as well as underlying file system-specific pointers for file and inode operations.

```
struct inode {
        struct list_head        i_list;
        unsigned long           i_ino;
        atomic_t                i_count;
        unsigned int            i_nlink;
        uid_t                   i_uid;
        gid_t                   i_gid;
        loff_t                  i_size;
        struct timespec         i_atime;
        struct timespec         i_mtime;
        struct timespec         i_ctime;
        blkcnt_t                i_blocks;
        umode_t                 i_mode;
        struct mutex            i_mutex;
        const struct inode_operations   *i_op;
        const struct file_operations    *i_fop;
        struct super_block      *i_sb;
        unsigned long           i_state;
        /* fs or device private pointer */
        void                    *i_private;
        ... several other members ...
};
```

2.3 Access Control

Of particular interest in this thesis is how file systems implement access control. Most file systems provide two mechanisms for access control — the standard UNIX user-group-owner permissions, and Access Control Lists (ACLs). The `CAP_DAC_OVERRIDE` capability [26] overrides these permissions (the purpose being to allow `root`, or any process with this specific capability, access to all file system objects). Since permissions checks are generally file system-independent operations, the VFS provides mechanisms to determine whether a given user has sufficient permissions to perform an operation on a given file system object or not. Most file systems use these func-

tions.

2.3.1 UNIX Permissions

Almost all modern file systems support the standard UNIX permissions paradigm. Each file system object (files and directories) has an owner, identified by its `uid` and `gid`. Permissions are specified by an octal triplet — the first octal digit specifies permissions for the owner, the second octal digit for the any user in the group specified by the `gid`, and the third octal digit for users who do not fall into either of these two categories ("others"). Each octal digit is a 3-bit number with the most significant bit specifying whether a user can read the object or not, the next bit specifying the same for writing to the object, and the last bit specifying whether the data in the object may be considered executable or not. For example. if the permissions for a file are represented by the number `640`, it is inferred that the owner of the file can read from and write to the file, but not execute it, users in the group specified by the `gid` of the file can only read from it, and others cannot access the file at all.

2.3.2 POSIX Access Control Lists

It was found that the standard UNIX permissions model is not sufficiently fine-grained for modern computing needs. The POSIX ACL [26, 27, 28] mechanism (hereby called "ACL") was born out of this need for a fine-grained mechanism to control access to file system objects. An ACL is a set of entries of the following form.

$$type : [id] : permissions$$

Here, `type` is an integer representing one of the following types of ACL entries.

- Owner: This specifies the permissions for the owner of the file. The `uid` of the owner is already part of the file's metadata, therefore the `id` of this entry is set to -1.
- Owning Group: Similar to the owner entry, but applies to the `gid` instead of the `uid.`
- Named User: Provides permissions for a specific user, as specified by the `id` field.
- Named Group: Similar to named user entries, but for groups. The group `gid` is specified in the `id` field.
- Mask: This is also called "group class permissions". Group class entries are defined as one of owning group, named user, and named group entries. This field defines an upper bound on the permissions that can be granted by the group class entries[1]. If there are no named user or named group entries, this is identical to the group permissions.
- Others: Specifies the permissions for all users who are not explicitly covered by any other ACL entry. Again, the `id` field is meaningless here.

When an ACL consists of only the first 3 kinds of entries, it is called a *minimal* ACL, since it can be represented entirely by the standard UNIX permission bits. An ACL

[1]For example, if there is a named user entry for user a granting "read" and "write" and a named group entry for group g granting "read", "write" and "execute", but the mask entry is set to only "read", then user a and users in group g will only be allowed read access. Even though they have greater access due to their respective entries, the mask entry sets an upper bound on what permissions may be given via these entries.

that also includes named user/group entries or a mask entry is called an *extended* ACL. In addition to the standard ACL that is used for access control directories may have another type of ACL called a *default* ACL. This ACL is used as the initial access ACL for all files and directories under this directory. The normal ACL is sometimes called an *access* ACL to distinguish it from the default ACL.

The `acl` package in userspace provides the `libacl` library for ACL manipulation by programs as well as the `getfacl`, `setfacl` and `chacl` utilities for users to manage ACLs. The library provides an API to read ACLs from disk into an in-memory representation, traverse the in-memory ACL, make modifications to it, and finally to write these back to disk if required. Some examples of how these utilities may be used is given below.

```
user@host $ getfacl foo
# file: foo
# owner: user
# group: somegroup
user::rw-
group::---
other::---
user@host $ setfacl -m u:someuser:r foo
user@host $ getfacl foo
# file: foo
# owner: user
# group: somegroup
user::rw-
user:someuser:r--
group::---
mask::r--
other::---
```

2.4 Extended Attributes

Extended attributes [29, 24] (also referred to as EAs or `xattrs`) provide a mechanism to store metadata on files and directories. They provide a generic means to attach arbitrary $(name, value)$ pairs to a given file or directory, and then subsequently look up a *value*, given a *name*. Extended attributes are supported by most modern file systems.

Extended attributes have been divided into four "namespaces" as follows. More namespaces may be defined if the need arises.

user: Extended attributes in the "*user*" namespace are used to attach arbitrary data to files and directories from userspace. Access control on these attributes is identical to that for the file.

system: Extended attributes in the "*system*" namespace are used to store system data such as ACLs and capabilities [30]. Permissions are determined by the implementation in the underlying file system.

trusted: Extended attributes in the "*trusted*" namespace are used to store metadata from userspace that is only accessible to privileged users (`root`, and users with the `CAP_SYS_ADMIN` capability).

security: Extended attributes in the "*security*" namespace are used in the implementation of security modules such as in SELinux [31].

Extended attribute names are prefixed by the namespace, separated by a period. For example, an extended attribute in the user namespace called "*foo*" would be named "*user.foo*".

The `attr` [24] package provides two tools — `getfattr` and `setfattr` — to allow reading and writing extended attributes and the `libattr` library for programs that need to manipulate extended attributes.

2.4.1 Implementations

Since the implementation of this work depends heavily on extended attribute storage, the implementation of extended attribute support in the kernel is presented. The VFS provides the following functions for EA manipulation.

1. `vfs_getxattr()` for looking up an attribute value by its name.
2. `vfs_setxattr()` for setting an attribute value by name. The parameters to this function determine whether the EA should be created, or replaced if it already exists.
3. `vfs_removexattr()` for deleting a particular attribute whose name is given as an argument.
4. `vfs_listxattr()` for looking up the names of all extended attributes on a file.

Each file system which supports extended attributes must support the "get", "set" and "list" operations. "remove" is just a special case of "set" with a null value.

Different file systems use different methods for storing the extended attributes on disk. The `ext3` file system tries to accommodate extended attributes within the on-disk inode. Once space in the inode runs out, a single separate file system block is allocated for extended attribute storage. A pointer to this block is stored in the inode

(in the misleadingly named `i_file_acl` member of `struct ext3_inode`). Files with an identical set of extended attributes share this block. Clearly, the space available for storage is limited by the file system block size.

ReiserFS stores extended attributes in a directory named *".reiserfs_priv/xattrs"* on the root directory of the inode. *".reiserfs_priv"* is not accessible from userspace. Each file system object in ReiserFS has a unique "object id". An extended attribute of the form (*name*, *value*) is stored in a file *".reiserfs_priv/xattrs/<object-id>/name"* which just contains *value*. Each attribute is allowed to be up to 64 KB in size.

XFS has several different forms of storage for these attributes. If the set of extended attributes is small enough, the inode is used. If there is not enough space in the inode, then a B+ tree is used. The XFS mechanism, although quite complex, is also one of the most efficient and scalable.

JFS stores extended attributes in a series of name-value pairs in the inode, if space is available. Otherwise, it allocates an extent for this purpose.

2.4.2 ACL Storage

In Linux, most file systems store extended ACL entries in an extended attribute in the system namespace. The ACLs are first packed and then stored in the *"system.posix_acl_access"* and *"system.posix_acl_default"* attributes for access and default ACLs respectively. The VFS provides functions to convert between the in-memory and extended attribute representation of ACLs. The in-memory representation of ACL entries is shown below. The on-disk representation is a file system-dependent representation of these entries, so similar changes also had made to the `struct`

`ext3_acl_entry` which serves this purpose.

```
struct posix_acl_entry {
        short e_tag;            /* The type of ACL entry */
        unsigned short e_perm;  /* The permissions granted */
        unsigned int e_id;      /* The uid/gid for the entry,  */
                                /* if applicable. -1 otherwise */
};
```

More details can be found on the website for the Linux POSIX ACL implementation [24] as well as Andreas Grünbacher's paper titled "*POSIX Access Control Lists on Linux*" [28].

Chapter 3

TransCrypt Implementation

TransCrypt was originally implemented as part of the `ext3` file system on the Linux 2.6.11 kernel, and was later ported to the 2.6.20 kernel and subsequent versions. This chapter provides a brief sketch of the original implementation of TransCrypt.

3.1 In-Kernel Architecture

The bulk of TransCrypt's implementation lies in the kernel. This is required because `root` is excluded in the TransCrypt trust model, and any process in user space is vulnerable if some user is able to gain super user privileges.

Since all file system access occurs through the VFS, the changes required were primarily in the VFS layer. Figure 3.1 illustrates how TransCrypt fits into the kernel in the context of the VFS architecture illustrated in figure 2.1.

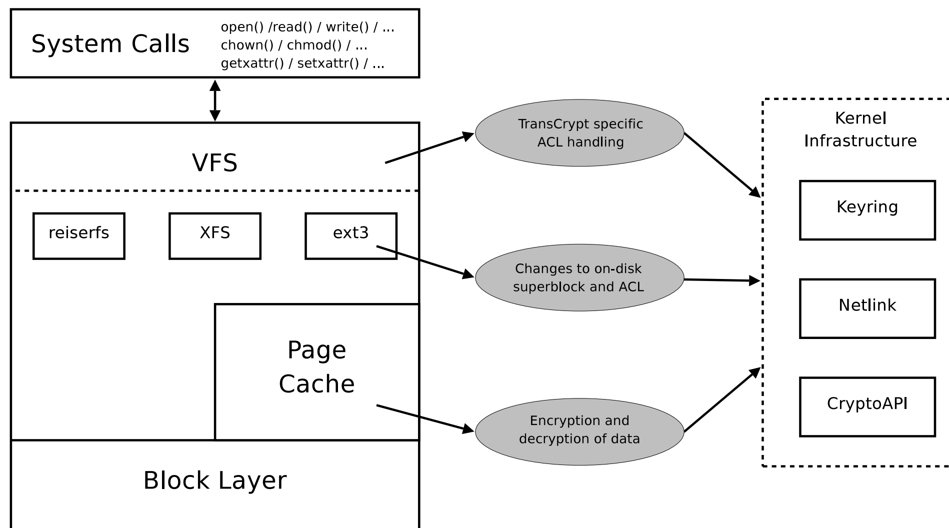


Figure 3.1: TransCrypt and the Linux VFS

3.1.1 VFS Core

At the uppermost layer, modifications were made to generic VFS functionality. Both in-kernel and on-disk structures for POSIX ACLs were modified to include two new fields, `certid` and `token`, which represent the certificate identifier for the user certificate that was used to create the token for the user and the token itself. These fields are only relevant to user and named user entries in the ACL. Changes were made in the in-memory `struct posix_acl_entry` data structure as well as in the functions to convert between the in-memory and packed extended attribute representations.

```
struct posix_acl_entry {
        short e_tag;
        unsigned short e_perm;
#ifdef CONFIG_TRANSCRYPT
        unsigned char e_certid[MAX_SIZE_CERTID];
        unsigned char e_token[MAX_SIZE_TOKEN];
#endif
        unsigned int  e_id;
};
```

An extended ACL is created for a file only if required. If a minimal ACL is sufficient to express the permissions for a file, the extended ACL is not created since the standard UNIX permissions contain all the information required. TransCrypt, however, requires that every file have an extended ACL, since the owner's token is stored in the corresponding ACL entry. The VFS provides functionality to determine whether a file requires an extended ACL or not, given a set of permissions. In the TransCrypt case, this functionality is forced to return "true" in all cases. Additionally, in the VFS, an extended access ACL is also not created unless the parent directory has a default ACL (if the parent directory does not have a default ACL, the `umask` is used). To address this problem, TransCrypt sets a default ACL on the root directory of the file system when it is first mounted, and this gets propagated to all files and directories.

Finally, since TransCrypt does not support ACL entries for groups and others, changes were made to the VFS code to disable the use of these.

3.1.2 ext3

One layer below the VFS, modifications were made to the `ext3` file system code. `ext3` was chosen because it is a stable and mature file system, yet its internal structure is relatively simple.

The first change was made to the in-memory `struct super_block` structure, where the `s_flags` member was modified to indicate whether it represents a TransCrypt file system or not. This is used to quickly determine whether a given `struct inode` represents a file on TransCrypt volume or not — a check that is performed

quite frequently.

File system-specific metadata was also added to `struct ext3_super_block`, which is written to the on-disk superblock. The "reserved" field at the end of the structure was used for this purpose. The metadata added include a flag to indicate that this is a TransCrypt file system, the algorithm used while encrypting the FEK for the purpose of blinding, the algorithm used for file encryption, the method used for generating initialization vectors for encryption, a hash of the FSK (which is used to verify that the FSK provided at mount time is correct) and the passphrase-based key derivation (PBKDF2 as defined in PKCS#5 [21]) parameters used to convert the file system passphrase to the FSK. The modified `struct ext3_super_block` is shown below.

```
struct ext3_super_block {
        ... several members ...
#ifndef CONFIG_TRANSCRYPT
        /* Padding to the end of the block */
        __u32   s_reserved[190];
#else
        __u32   s_reserved[190-21];
        __u32   s_transcrypt_flag;
        char    s_transcrypt_fsalgo[TRANSCRYPT_FSALGO_BUF_SIZE];
        char    s_transcrypt_pfalgo[TRANSCRYPT_PFALGO_BUF_SIZE];
        char    s_transcrypt_ivmode[TRANSCRYPT_IVMODE_BUF_SIZE];
        unsigned char
                s_transcrypt_fsk_hash[TRANSCRYPT_FSK_HASH_LENGTH];
        unsigned char
                s_transcrypt_fsk_salt[TRANSCRYPT_FSK_SALT_LENGTH];
        __u32   s_transcrypt_hash_num;
#endif
};
```

Another set of changes were made to deal with the creation of tokens at ACL

initialization time and token repacking. The latter involves recreating the tokens for all users in the ACL from the blinded FEK. This is required every time a file's ACL changes because the kernel is only provided the updated ACL from userspace, and not the list of changes from the original ACL.

3.1.3 Page Cache

At the lowest level, changes were made to the page cache (technically, the buffer cache, but these two layers have been merged) for encryption and decryption of data. As mentioned in section 2.1, all read and write operations go through the page cache (with the exception of `O_DIRECT` operations). While it would have been possible to encrypt the data before writing it to the page cache and decrypting it while reading, this would have been highly inefficient since every read and write operation would have required a cryptographic operation. Instead, encryption and decryption are performed just before writing to and just after reading from the disk respectively. Originally, modifications were made to the page cache itself. This involved short circuiting a more optimal code path through the `bio` layer. Subsequently, changes were made to move bulk encryption to a lower layer in order to create a cleaner implementation and improve performance [32].

3.1.4 Keyring

TransCrypt needs to store FEKs in memory for every open file. The kernel keyring infrastructure [33] provides a simple way to securely store keys in memory and search through them. Security is guaranteed by enforcing permissions checking for key and

keyring accesses and storing keys in unswappable kernel memory. A `struct key` data structure is defined to encapsulate a key. Every key has a "*description*" by which it can be identified. A *keyring* is a structure that contains a set of keys which can be searched for by their key description. Keyrings are implemented as normal keys which contain an array of keys, and can implement their own methods for operations such as adding, deleting and searching for keys. While there are a set of predefined keyrings for different purposes, none of these were adequately suitable for use by TransCrypt.

A separate keyring was thus defined for TransCrypt. For storage in this keyring, FEKs are described by a string of the form "<*inode-address*>". When the file is first opened, the FEK for that file is retrieved, and the key is stored in the TransCrypt keyring. Every time the FEK for a given inode is required for an encryption/decryption operation, the corresponding `struct inode` is used to form the description of the FEK and this is used to search the TransCrypt keyring for the FEK. Key removal from the keyring is done lazily. When the keyring structure, which has a finite amount of storage, becomes full, the keyring is iterated over, and keys that are no longer in use are discarded (this process is termed *key reaping*).

3.1.5 Communication with Userspace

As described in section 1.3, the kernel needs to communicate with the `transcryptd` daemon for retrieving user certificates, as well as with the `transcrypt-auth` server to perform token decryption. The kernel provides the netlink socket interface [34, 35] for communication with userspace using the familiar BSD sockets API. TransCrypt uses this API for communication with the `transcryptd`. Communication with `transcrypt-auth` is performed via `transcryptd`.

When the user creates a file, the kernel opens a connection to `transcryptd`. A request for the user's certificate (`GET_CERT`) is then sent. `transcryptd` receives this request, does a certificate lookup, and then sends a message with the certificate back to the kernel (`REPLY_CERT`).

When opening an existing file, the permissions checking code must retrieve the token and decrypt it. The kernel first gets the user's certificate as explained above. It then uses the user's public key to encrypt a random session key and sends this in a session establishment packet (`EST_SESS`) to `transcryptd`, which just forwards this to `transcrypt-auth` (this happens for all subsequent packets in the session in both directions). `transcrypt-auth` decrypts the session key and sends back an acknowledgment message (`ACK_SESS`). All subsequent messages are encrypted using this session key. The kernel, on receiving the acknowledgment, proceeds to send a key acquisition request (`KEY_ACQ`), which is basically a request to decrypt the token. `transcrypt-auth` decrypts the token and returns this to the kernel (`KEY_RESP`). The kernel now terminates the session (`END_SESS`) and `transcrypt-auth` acknowledges this (`KILL_RESP`). While currently `transcrypt-auth` accesses the user's private key for these operations, the protocol is designed in such a way that `transcrypt-auth` can use a more secure storage mechanism (like a smart card) to ensure adequate protection for the user's private key.

The TransCrypt communication framework is designed in the following way. When a message needs to be sent to userspace (and thus a reply must be waited for), the kernel sends the message from the user context of the process that caused this call to happen, and then puts the process to sleep. When a message comes from `transcryptd` to the kernel, a callback function (registered when the TransCrypt code is first loaded) is called in context of the process whose message is being replied

to. This callback function process the data received and then wakes up the sleeping process, which then carries on with its function. The synchronization is performed using the kernel "*completion*" infrastructure [36].

3.1.6 CryptoAPI

The kernel CryptoAPI [37] interface provides a generic infrastructure to use the numerous cryptographic algorithms provided by the kernel for such operations as generating cryptographic hashes, message authentication codes, and symmetric encryption and decryption. TransCrypt uses this API from the page cache layer for bulk encryption and decryption.

In addition to this, two sets of modifications were made for TransCrypt. First, CryptoAPI was extended to allow the use of asymmetric cryptographic operations. This included the support for the RSA algorithm [38], as well as a generic layer for asymmetric operations to support the implementation of other asymmetric algorithms in the future. The second feature added was support for X.509 certificate parsing and verification. For this purpose, the X.509 module from the userspace XySSL [39] library was ported to the kernel.

3.2 Userspace

TransCrypt requires support from userspace for multiple reasons. Firstly, the changes made to the on-disk storage format (superblock and ACL entries) require corresponding changes in userspace libraries and utilities that deal with these on-disk structures.

In addition to this, TransCrypt's key management also relies on userspace helper daemons. Several helper utilities and scripts have also been created to make using TransCrypt easier. The following sections briefly describe each of these userspace components.

3.2.1 libacl

As mentioned previously, `libacl` is a userspace library which provides an easy, implementation-independent API for reading and manipulating the ACL entries of a given file. `libacl` works by reading the appropriate extended system attribute ("*system.posix_acl_access*" or "*system.posix_acl_default*") and unpacking the value into memory. This in-memory version is exposed via a set of data structures which can be read, iterated over, and modified by the client program. When a program uses the API to write out a modified ACL, the library repacks the ACL into the on-disk format and writes it out to the appropriate extended attribute.

The library was changed in a manner similar to the kernel's POSIX ACL handling. Changes were made to the in-memory data structures that held ACL entries to accommodate the `certid` and `token` fields. The helper functions used to convert between the on-disk and in-memory storage also needed to be modified to accommodate these fields. Finally, APIs were added for userspace to populate the `certid` and `token` fields in manner similar to those used for modifying other fields.

3.2.2 e2fsprogs

`e2fsprogs` provides a number of userspace utilities to help manage `ext2` and `ext3` volumes. From the TransCrypt perspective, all parts of the package that dealt with the superblock needed to be modified. The superblock is populated by the `mkfs` utility when creating an `ext2/ext3`-based file system. As mentioned in section 3.1.2, TransCrypt added several fields to the on-disk superblock. The `mke2fs` utility was modified in order to also take each of the following TransCrypt parameters.

`--tcpt` To indicate that this is a TransCrypt volume

`--fsalgo` The algorithm used for encryption of the FEK with the FSK (e.g.: "*aes-cbc-128*" indicates use of the AES algorithm for encryption with the Cipher Block Chaining mode and 128-bit keys)

`--pfalgo` The algorithm used for encrypting file data using the FEK (follows a format similar to `--fsalgo`)

`--ivmode` The method used to generate initialization vectors (IVs) for blocks (currently only the value "*zero*", which uses an IV of all zeroes, is supported)

`--passphrase` Used to pass the salted hash of the FSK (derived from a passphrase provided by the administrator)

`--hash_num` One of the parameters used in converting the passphrase to an FSK (it is the number of "*n*" parameter to the PBKDF parameter which determines the number of times the passphrase is hashed)

These parameters are collected and used to populate the superblock fields in a local data structure. Eventually, these are written to the numerous on disk copies of the

superblock.

3.2.3 Daemons

The `transcryptd` and `transcrypt-auth` daemons are required by the kernel for performing key management tasks. These tasks are easily implemented in userspace without compromising the strict trust model employed, since attacks on these operations are either thwarted or detected.

`transcryptd` serves two functions — firstly, it is an interface to whatever user certificate storage mechanism is used by the organization deploying TransCrypt. Currently, a file-based storage mechanism is supported, but adding support for other mechanisms such as directory servers is trivial. Certificates are stored in a standard location (`/etc/transcrypt/certs/`) with a file (`/etc/transcrypt/certtab`) providing mappings between $(uid, certid)$ pairs and the corresponding X.509 certificate.

In addition to this, `transcryptd` forwards all other communications from the kernel to `transcrypt-auth` and back. When a packet is received, it first checks the type of the packet. If it is one that must be forwarded (that is, not a `GET_CERT`), it first checks if it is a session establishment packet (`EST_SESS`). If so, it spawns a new thread to open a TCP connection to `transcrypt-auth` and adds this connection to a table mapping the corresponding session to this thread. For other packets, `transcryptd` examines the packet's session identifier and then uses the table to look for the thread corresponding to that session. This thread is reused for this communication session.

By design, `transcrypt-auth` is simply an interface to a trusted private key store (PKS), such as a USB drive, smart card, trusted platform module, etc. Cur-

rently, TransCrypt implements a simple file-based PKS. `transcrypt-auth` keeps a file containing mappings between the user's *certid* and the corresponding private key file, which is used for the operations required.

3.2.4 Utilities

TransCrypt also provides a number of helper utilities to make administrative tasks simpler. There are scripts to provide an easy mechanism for adding and removing certificates from the certificate store. The same is also available for managing the private keys for `transcrypt-auth`.

Two wrapper programs are provided for the `mkfs` and `mount` utilities, called `mkfs.transcrypt` and `mount.transcrypt` respectively. `mkfs.transcrypt` is merely a wrapper over the `mke2fs` program. Its purpose is to obtain a passphrase from the user, convert it into an FSK using the PBKDF2 passphrase-based key derivation method, and then pass this and other relevant options to the modified `mke2fs` program, which initialises the superblocks on disk accordingly.

`mount.transcrypt` wraps over the `mount` utility. When used to mount a TransCrypt file system, it obtains the passphrase from the administrator. It then generates the FSK as was done at `mkfs` time, and then mounts the file system with the appropriate parameters (the FSK is *not* sent to the kernel over the mount command-line since this is globally visible in `/proc/mounts`). The kernel receives and parses the parameters, reads the on-disk superblock, and subsequently blocks while waiting for the FSK, which is sent to the kernel via a file in `configfs` [40]. The hash of the FSK is then calculated, and mount succeeds only if this matches the hash in the superblock.

Chapter 4

The TransCrypt-Enhanced Virtual File System

The earlier implementation of TransCrypt had certain limitations, which were accepted at that time in order to simplify implementation and develop a quick prototype of a working system quickly. The tradeoff made was basically that TransCrypt could not be used on file systems more advanced than `ext3`. The design also added to the amount of code maintenance required, since changes needed to be made to existing kernel code as well as userspace packages.

An effort is made in this work towards rearchitecting the TransCrypt in-kernel architecture in order to solve these problems. In particular, modifications are made to the way TransCrypt-specific metadata is stored in on disk, allowing TransCrypt to be implemented at a higher layer in the VFS stack than it previously was.

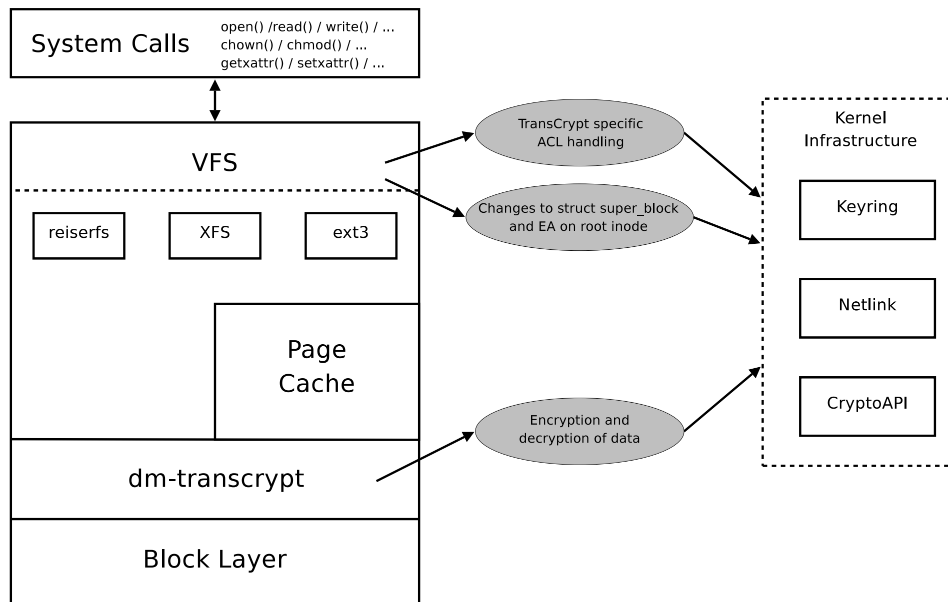


Figure 4.1: Modified TransCrypt and the Linux VFS

4.1 Design

The fundamental problem with the way TransCrypt was implemented earlier was that it was scattered across various layers in the file system stack. In order to support other file systems and truly be independent of the underlying file system, the implementation clearly needed to be moved to a higher layer in the VFS. Figure 4.1 illustrates the modified design as compared to the earlier design illustrated in figure 3.1. In the comprehensive implementation, a encryption and decryption was moved from the page cache into a separate layer has been added by Sainath Vellal [32] in the form of a device mapper-based implementation called `dm-transcrypt`, providing a cleaner implementation and better performance.

In this design, the handling of TransCrypt-specific metadata, at both the file system and per-file level, is now done in the VFS rather than in the file system code. This is enabled by the use of extended attributes.

4.1.1 File System Metadata

In this generalization, the file system-wide TransCrypt metadata was removed from file system-specific storage, both in-memory and on-disk to more general VFS storage. This was accomplished by moving attributes specific to TransCrypt in the `struct ext3_super_block` data structure to the generic `struct super_block` data structure of the VFS, in memory. In a similar manner, these attributes on disk were moved from the superblock into an extended attribute set on the root directory of the file system. The new information stored in these generic data structures, as given below, is almost the same as what was previously stored in the `struct ext3_super_block` fields, as described in section 3.1.2.

version: This field is an integer that roughly corresponds to the "*flag*" field used previously. The terminology is changed to "version" in order to allow future changes to be backward compatible if required. Code may be written to support old or new behaviour based on this version. This is currently set to 1.

pbkdf_n: This field is the "*hash_num*" field, renamed to more accurately reflect its purpose. It represents the number of times the passphrase is hashed by the passphrase-based key derivation function as defined in PKCS#5 [21].

pbkdf_salt: This corresponds to the "*fsk_salt*" field, and has been renamed to disambiguate its purpose. It contains the salt that is used at the time of passphrase-based key derivation function as defined in PKCS#5.

fsk_hash: This is identical to the field of the same name that was used in the earlier implementation and contains the hash of the FSK.

fsk_algo: This field corresponds to the the "*fsalgo*" field used earlier, and is of the form "<*algorithm*>-<*cipher-mode*>-<*key-length*>".

bulk_algo: This field corresponds to the "*pfalgo*" field used previously, but has been renamed for clarity.

bulk_ivmode: This field is corresponds to the "*ivmode*" field used earlier, and represents the method in which initialization vectors are generated for each block that is encrypted.

While each of these could have be stored in a separate extended attribute, this would have been inefficient. Firstly, reading all the data would have required seven different extended attribute to be read from the disk. Moreover, some file systems (such as `ext3`) have a limited amount of space for extended attribute names and values. This space would not have been efficiently utilized by storing seven different extended attribute name and value pairs.

Instead the data is packed into a binary blob and stored as a single extended attribute which is created by the `mkfs.transcrypt` program when initializing the TransCrypt volume. The kernel subsequently reads this extended attribute at mount time to populate the modified `struct super_block` in memory. The extended attribute can be modified only by the super user. This prevents a trivial denial-of-service attack by non-`root` users. Even if an attacker does gain `root` access, reading the information in the extended attributes provides no sensitive information, and modifying any of the data will merely make the TransCrypt volume unreadable, leading to a trivially detected denial of service which can be corrected after remounting the file system.

4.1.2 Per-File Metadata

As seen in the previous chapter, TransCrypt stores two metadata items for each file — the *certid* and the *token*. In order to store this metadata in a file system-independent manner, the new design takes an approach that is similar to the one used for file system-wide metadata. The $(certid, token)$ pairs for all users are stored in an extended attribute from the user namespace that is attached to the file.

The tokens are stored as a list of *token entries*. Each token entry consists of a triplet of the form $(uid, certid, token)$. Previously, the correspondence between a user and the corresponding token was implicitly established in the ACL entry using the *id* field. In the new implementation, since this is separated out, the `uid` must be explicitly stored with each entry.

This metadata is stored in aggregated form rather than as individual extended attributes (for example, one extended attribute for each user) is the same as for file system-wide metadata. Storing this data in aggregated form is more compact and efficient in terms of utilization of limited extended attribute name-value pairs.

4.1.3 Extended Attribute Caveat

It is worth noting that the use of ACLs and extended attributes is not yet ubiquitous, and support for these in userspace can be found to be lacking at times. While common tools such as `cp`, `ls`, Vim, etc. have had support for both ACLs and EAs added to them, others such as `rsync` and `scp` have not. The GNU `tar` utility is also a notable utility that does not support either ACLs or EAs. This is because there is no standard way to store these defined in the format used by `tar`. The `pax` utility, which uses

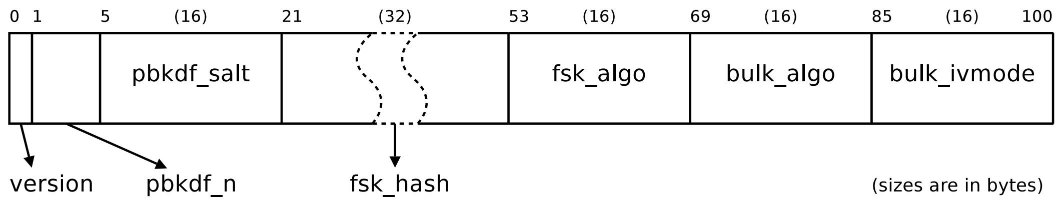


Figure 4.2: On-Disk EA format for TransCrypt Volume Parameters

a format of the same name defined by POSIX.1 [41] supports ACLs and extended attributes in the form of vendor-specific extensions. The `star` [42] utility, which is an improvement over GNU `tar` also supports both, and is available on most Linux distributions. In either case, when tools such as these are used to create archives for backup, care must be taken to ensure that the ACLs and (at least) TransCrypt-specific extended attributes are also backed up. Failing this, the data restored from a backup will not be readable.

4.2 Implementation

The design described here has been implemented over the TransCrypt 0.1 release, which, in turn, is built upon Linux kernel version 2.6.24.

4.2.1 Volume Parameters

The metadata items specific to TransCrypt, as described in section 4.1.1, are organized in a `struct transcrypt_volume_params` data structure. The corresponding parameters are no longer part of the `struct ext3_super_block` data structure. This structure is populated at mount time from the values stored in the "*user.transcrypt.params*" extended attribute, which is attached to the inode corresponding to the root directory

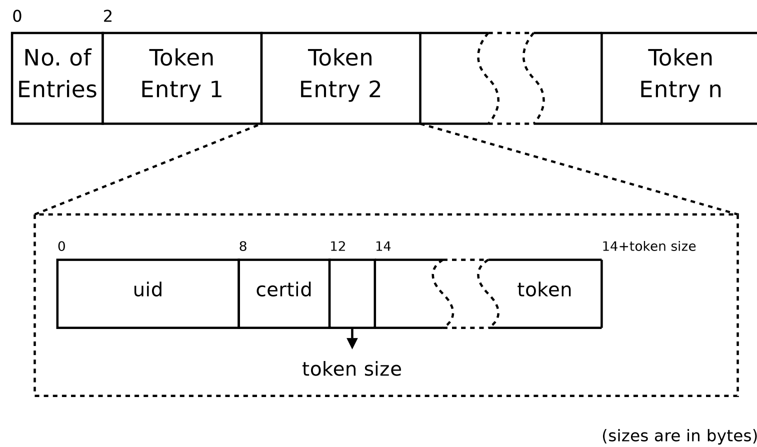


Figure 4.3: On disk EA format for tokens

of the volume. It is stored on disk in a packed format as shown in figure 4.2, with integer fields (specifically *pbkdf_n*) stored in little-endian format. Fixing the endianness ensures that the data is stored in a host-independent fashion.

The `mount` system call, after some preliminary sanity checks, results in a call to the `vfs_kern_mount` VFS function. If the "*encrypt*" option is passed to the `mount` command, the "*user.transcrypt.params*" extended attribute is read, unpacked into memory in the `struct transcrypt_volume_params` data structure, and finally attached to the `struct super_block` data structure. This is done using an `s_transcrypt_params` pointer field added to the `struct super_block` data structure for use by TransCrypt.

The metadata is then used for verifying the FSK, determining the algorithm to use for encryption/decryption, etc.

4.2.2 Tokens

The tokens for a file are also stored on disk in a packed format. For this purpose, an extended attribute named "*user.transcrypt.tokens*" is attached to that file. The on-disk format for the attribute data is illustrated in figure 4.3. All integer fields in this extended attribute are also stored in little-endian format.

The notable differences from the previous ACL-based storage are that the number of entries must now be stored separately, and that the tokens can now be of variable size. Previously the length of users' private keys were restricted by the maximum size of the token, since the token was of the same size as the user's private key. Storing the size of the token on disk allows users to have private keys of arbitrary and differing lengths.

The parts of the VFS affected by moving code out of the `ext3` implementation are as follows.

File creation: When a file is first created, a random FEK must be generated, the corresponding token for the owner of the file must be created, and then the "*user.transcrypt.tokens*" extended attribute must be initialized with this token. This is done after the file is successfully created by the VFS workhorse function — `open_namei_create()`.

File open: On receiving an `open` system call from a user process, the VFS ascertains that the user has sufficient permissions to open the file using the `may_open()` function. At this point, the "*user.transcrypt.tokens*" extended attribute is read and unpacked, and the token for the user is extracted. The FEK, once successfully acquired, is added to the TransCrypt keyring. If an error occurs while

trying to acquire the key, an error is returned and permission to open the file is not granted.

Changes to an ACL: Whenever changes are made to the ACL of a file, specifically to the owner and named user entries, the tokens might need to be changed. If a named user entry is added, the corresponding token must be added, and if a named user entry is removed, the user's token must be removed. To keep in sync with the ACLs for all files, the `vfs_setxattr()` function is modified (when the attribute being set is "*system.posix_acl_access*"), since this is the only path through which all ACL modifications are dispatched to the underlying file system. Once the ACL has been updated, the FEK is retrieved from the calling user's token, and tokens for all users are regenerated. While this is inefficient, it is the cleanest solution available. The same is done for the `chown()` system call as well.

4.2.3 Userspace Utilities

In userspace utilities, the primary changes are to the `mkfs.transcrypt` utility. While previously this was merely a wrapper to derive a key from a passphrase and pass it on to `mke2fs`, it must now perform a different task. `mkfs.transcrypt` takes the same parameters as before, with some additions. The newly introduced "*--fstype*" parameter is used to determine what the underlying file system should be. For example, calling `mkfs.transcrypt` with "*--fstype foo*" would call the file system-specific `mkfs.foo` utility to create the file system on the given volume. The wrapper then temporarily mounts this file system, creates the value for "*user.transcrypt.params*" from the rest of its command-line parameters, and sets the value for this extended attribute on the

root directory.

A "*--reinit*" parameter is also introduced. This skips the file system creation and only sets the extended attribute. This option can be used if, somehow, the extended attribute on the root directory is lost. All command-line parameters other than those mentioned here and in section 3.2.4 are passed on to the file system-specific `mkfs` command.

In addition to this, all changes to the `e2fsprogs` and `acl` packages are discarded. The default packages that are shipped with any Linux distribution suffice for use with TransCrypt.

4.2.4 Example usage

An example of how to set up TransCrypt on a ReiserFS partition is shown below.

```
root@host # dmsetup ... # (to set up dm mapping)
root@host # mkfs.transcrypt --fstype reiserfs \
                            --fsalgo aes-aes-128 \
                            --pfalgo aes-cbc-128 \
                            --ivmode zero
                            /dev/mapper/transcrypt1
Filesystem passphrase:
Retype filesystem passphrase:
...
root@host # mount.transcrypt /dev/mapper/transcrypt1 /mnt/foo
Filesystem passphrase:
```

Assuming `transcryptd` and `transcrypt-auth` are running, the TransCrypt volume (mounted at "*/mnt/foo*" in this example) may now be used as any other file system would be.

Chapter 5

Testing

While proper testing of software is important in general, it is particularly crucial in the context of file systems, since bugs in file system code can lead to data loss and corruption. In addition to testing manually, it is necessary to automate the testing process in order to catch errors as quickly and reliably as possible. Several automated test suites are available for file system testing in Linux, many of which have been aggregated by the Linux Testing Project [43] (LTP). TransCrypt is tested using the *fsx* [44] and *fsstress* [45] tools from the LTP suite, as well as a set of internally developed test scripts.

The *fsx* and *fsstress* tools execute a random set of operations, such as file and directory creation, file open and close, normal read and write operations, memory-mapped read and write operations, and asynchronous I/O. The TransCrypt internal test suite is used for unit testing of modules such as RSA cryptography, X.509 certificate parsing and verification, and for system testing, which involves creating a TransCrypt file system, mounting it and reading from and writing to it.

Since the core VFS code for handling permissions has been changed, a test suite has been created to ensure that these changes do not make the VFS behave incorrectly with respect to the defined behaviour for various permissions related operations. In the following sections, the test *strategy*, which defines what needs to be tested, is discussed. Subsequently, the test *methodology*, or how these test cases and scenarios are actually tested for, is covered.

5.1 Test Strategy

The various cases to be considered for testing are enumerated by first listing the set of file system objects that are affected by TransCrypt, and then defining the set of operations that can be performed on these objects that affect, or are affected by, the permissions of that object.

The file system objects that need to be considered in the TransCrypt case are regular files, and directories. No other objects are affected by TransCrypt. However, while testing it is must also be verified that the behaviour of other objects is not changed on a TransCrypt volume. The operations that change, or are affected by, file and directory permissions are listed below.

- Read the metadata of the given object using the `stat` system call
- Create a file/directory
- Truncate an existing file
- Opening a given file (for reading or writing)

- Change the file owner and group
- Change file permissions using `chmod`
- Add an ACL entry (particularly a named user entry)
- Remove a user from a file ACL
- Change the permissions granted by a named user entry
- Change the mask ACL entry
- Change the `umask`, which is used to determine the initial permissions of a file
- Access a file or directory through a symbolic link
- Grant programs special privileges using POSIX capabilities

The test suites include test cases for all these operations, both on files and directories. They also incorporate some basic sanity checks for correctness with regards to other types of objects such as symbolic links.

Tests usually consist of a set of operations followed by checks to verify that the operations were performed correctly. Some operations, such as file reads, make extensive use of the kernel data and metadata caches. This caching can sometimes hide a bug in the code from manifesting itself. For example, consider a scenario where a user a reads a file, causing the FEK for that file to be read into memory (and, thereby, cached in the keyring). Subsequently, say, a user b tries to open this file, without a valid token being present. It may occur that due to a bug, the cached FEK is used despite permissions checks failing. In order to catch such bugs, the file system is unmounted and then mounted again between the operation being tested and

the verification step for that operation. This causes all cached data and metadata for that volume to be flushed and discarded.

Finally, test cases must also verify that programs are not able to use any of the following capabilities [30] to override permissions checks for file read, file write and directory read operations.

`CAP_DAC_OVERRIDE`: This is capability instructs the kernel to bypass all permissions checks for a process.

`CAP_DAC_READ_SEARCH`: This capability tells the kernel to override read permission checks for file access, and read and execute permissions checks for directory access by the given process.

`CAP_FOWNER`: This allows the process to override permission checks which require the process' `uid` to be the same as that of the file or directory owner, excluding those operations covered by the two capabilities above. This basically includes permission to change the file permissions and add and modify ACL entries and extended attributes.

5.2 Test Methodology

It is clear that there are a large number of test cases to be considered, and writing scripts to perform these tests is a non-trivial task. In such cases, it is a common practice to create a *test framework* to simplify the creation of large test suites. Such a framework has been written to simplify the task of ensuring correctness of permissions handling in TransCrypt, using the Python `unittest` framework [46].

5.2.1 Python "unittest" framework

The Python `unittest` framework is the de-facto standard library used for test automation in the Python programming language. It provides mechanisms to define test cases, aggregate them into a test suites, run these tests, and, without any additional code, generate reports of successful and failed tests. A skeleton test case is shown below.

```
import unittest

class SomeTests(unittest.TestCase):
    def setUp(self):
        self.val1 = 2
        self.val2 = 3

    def tearDown(self):
        pass

    def testAddition(self):
        result = self.val1 + self.val2
        self.assertEqual(result, 5)
```

As can be seen, a test case is defined by extending the `TestCase` class. The `setUp` method allows the developer to define some common set up tasks to be performed prior to each test. Correspondingly, the `tearDown` method does the same for clean-up tasks that must be run at the end of each test. Several assertion methods, such as the `assertEqual` method used in this example, are provided to actually perform the required checks. Multiple tests can be defined as part of a single class.

The `TestSuite` class allows aggregation of several tests a single test suite, and the `TextTestRunner` utility class can be used to run a test suite and provide the results on standard output.

5.2.2 Common functionality

While creating the test suite for TransCrypt, it was noticed that a large number of tasks were common among the tests. This common functionality was identified for code reuse and increased maintainability and moved into a Python module named `transcrypt.py`. The main functionality in this module is provided by the `Volume` class. This class represents a TransCrypt volume, and provides an interface that allows several operations on the volume as described below. Objects of this class must first be initialized with the name of the device to be used for operations in the test case. The API provided by the `Volume` class is described below.

`mkfs`: This method is used to initialize the volume.

`mount`: This method is used to mount the file system. An optional parameter is used to specify the directory where the volume may be mounted. The default action is to create a temporary directory and use it as the mount point. Additional options to the `mount` command can also be supplied if required.

`unmount`: This method is used to unmount the file system.

`remount`: This method performs an unmount operation and then mounts the file system on the same directory that it was previously mounted on.

`createDir`: This method is used to create a new directory whose name is given as an argument. Optional parameters include a path with respect to the root directory of the file system under which the directory is to be created, the initial permissions for the directory, and a user whose identity is to be assumed while creating the directory. The last argument is required for test cases where

one user creates a file or directory and then grants access to another user. The default values for these parameters is as follows. The default path is assume to be the root directory of the volume, the default user is the user who has run the test suite, and the default permissions are determined by the current `umask`.

`createFile`: This is method similar to the `createDir` method, but is used to create a file. The parameters for this method are similar to those described for `createDir`. In addition to this, the initial contents of the file can also be passed as an argument. If this parameter is not specified, an empty file is created.

`grantUserAccess`: This method is used to grant a user access to a file or directory using the `setfacl` command. Parameters include the access permissions (read/write/execute) to be granted and, optionally, the user who is granting access (whose identity is assumed before running `setfacl`).

`revokeUserAccess`: This method is similar to the `grantUserAccess` method, but is used for access revocation.

`setAclMask`: This method is used to set a mask entry on a given file or directory. Parameters are similar to the `grantUserAccess` method.

5.2.3 Implementing Test Cases

The following snippet shows an example test case that uses the `Volume` class and ensures that the `root` user can create and read a file.

```
import unittest
import transcrypt
```

```
def PermTests(unittest.TestCase):
    def __init__(self):
        # perform some common initialization tasks

    def setUp(self):
        # mount the volume (self.volume)

    def tearDown(self):
        # unmount the volume

    def testRootReadFile(self):
        self.volume.createFile('file1', contents='Some string')
        self.volume.remount()
        # checkAccess() just checks if file can be opened for
        # the given operations
        self.assert_(self.checkAccess('file1', 'rw',
                                      contents='Some string')
```

The `setUp` and `tearDown` methods are used to mount the file system before each test and then unmount it after the test has been completed. The example test case uses the `Volume` class API to create a file and remount the volume, and then checks whether the file can be read. If the assertion fails, the `TestRunner` reports the failure in its output, along with the location of the failed assertion.

Several other tests have been written, and these follow the same pattern as the example given. Since the number of test scenarios can be large it is not possible to test for all such scenarios. The test suite currently contains tests that ascertain correctness of the core functionality, and is continuously being expanded.

Chapter 6

Conclusion

6.1 Performance Evaluation

While the new implementation of TransCrypt metadata storage provides numerous advantages, it is important to quantify the performance gain or loss incurred by the adoption of this new design. Some performance penalty is acceptable in view of the benefits derived, but it must be ensured that the actual impact on performance is within tolerable limits. In addition to this, it is also desirable to investigate the performance overhead incurred due to TransCrypt modifications as compared to the standard `ext3` file system.

Only a limited number of operations need to be considered for the purpose of evaluating performance in the context of this work. These are the set of operations that involve the permissions checking and the ACL code paths in the VFS, namely file creation, file open, ACL reading, ACL creation and ACL modification.

Almost all the operations above gain a significant performance enhancement due to caching. Specifically, each of these operations only deals with the file metadata, which, after the first operation on the file, would be cached in memory till it is flushed to the disk. It is not desirable to include disk operations in the performance measurements, since these are common to all measurements. Moreover, since the position of the disk head cannot be guaranteed to be the same at the start of any two operations, it is likely that the time spent on reading from or writing to the disk will be altered significantly between the two operations, thus skewing the results. Wherever possible, it is ensured that the cache is "warm" before measurements are taken, by performing a dummy run of the operation before running it again for the actual measurement. The first run ensures that the required data is in the cache for use in subsequent runs.

Measurements were taken on an Intel Core2 Duo E4400 dual-core CPU, running at 2 GHz, with a 2 MB L2 cache and 2 GB of RAM. The machine had a 160 GB, 7200 rpm Seagate SATA hard disk drive, on which a 3.5 GB partition was used for the file system being evaluated. One core on the processor was disabled in order to make performance more deterministic. On an SMP machine, a process may be migrated between processors by the scheduler. This would introduce indeterminacy in the measurements due to the overhead of migration. In addition to this, the timestamp counter (TSC) on each core, which provides a very accurate clock source tends to be unstable in an SMP environment.

The time taken to perform each of the following operations was measured.

`create`: File creation was tested by measuring the time taken to create a file and then flush it to disk. The underlying operation being measured basically involves

creating the file (i.e., initializing its metadata), generating a token for it, and writing this to the disk.

`open`: File open time was measured in a similar manner to `create`, by simply opening and closing an existing file. This tested the time taken to read a token, decrypt it via `transcrypt-auth`, and store the resultant FEK in the TransCrypt keyring.

`setfacl`: A named user entry was created for a given file in addition to the existing user entry in this operation. This involves reading the existing ACL, decrypting a token, generating a new token for the named user, and writing the new ACL and tokens to disk.

`getfacl`: This operation merely involved reading the entire ACL of the given file. The ACL consisted of the user, group and mask entries, and one named user entry.

Each operation was performed on 1000 files, 5 times, and the mean of these measurements was taken. A large number of files were considered because these operations complete very quickly on single files.

The result of the `create` and `open` operations is shown in figure 6.1. The Y-axis of the graph is in a logarithmic scale. TransCrypt introduces a significant overhead to both operations, as the kernel needs to communicate with userspace, as well as perform multiple asymmetric cryptographic operations. Some suggestions towards reducing this overhead are provided in section 6.3. Clearly, neither `create` nor `open` is significantly affected by the new design, compared to the previous design. The reason for this is that the `ext3` file system stores all extended attributes in a single block on disk. Thus, reading a single extended attribute containing the ACL

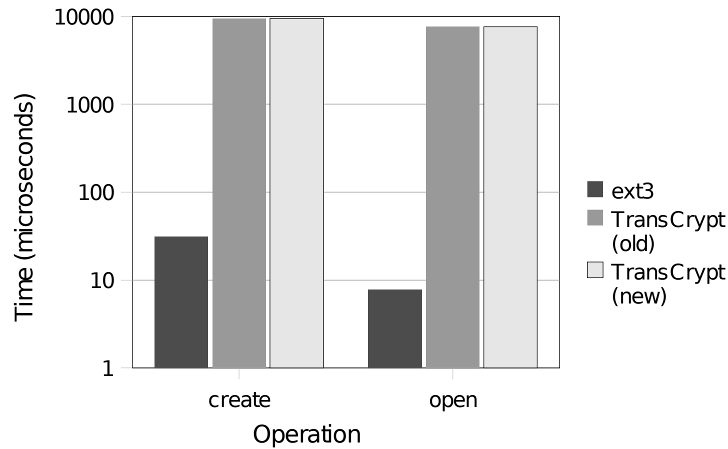
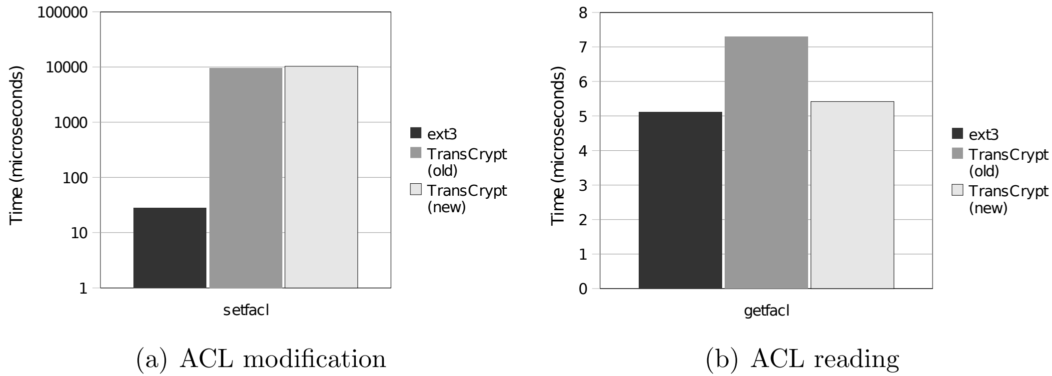


Figure 6.1: File create and open performance



(a) ACL modification (b) ACL reading

Figure 6.2: ACL reading and modification performance

and tokens, and reading both the extended attribute for the ACL as well as that for token, are functionally equivalent. The latter suffers from the minor overhead of having to read and unpack the token extended attribute separately.

Figure 6.2 illustrates the performance during the `setfacl` and `getfacl` operations. The graph depicting `setfacl` performance is on a logarithmic scale. The performance is comparable to `open` and `create`, as the operations performed are the same. The minor difference seen can be accounted for by the overhead of writing a separate extended attribute for tokens after the ACL extended attribute is written

out.

The `getfacl` operation, on the other hand, provides and interesting result. The new implementation takes about the same time as a normal `ext3` file system and about 75% of the time taken by the previous implementation. The reason for this is that the earlier implementation required the token to be unpacked and passed from the file system to the VFS and finally to userspace. This introduced an unnecessary overhead, since the tokens are not required at the time of `getfacl`. In the new implementation, this overhead was eliminated by storing the tokens separately and thus the `getfacl` operation works in exactly the same manner that it does on a normal `ext3` file system.

6.2 Summary

Data security is clearly crucial today, both to individuals and organizations, whether large or small. The TransCrypt file system provides a secure, scalable storage mechanism to cater to this need.

In this work, one of the primary limitations of TransCrypt — its implicit coupling with the `ext3` file system — has been resolved. Data that was stored in `ext3`-specific data structures in memory was moved to the more general VFS data structures. On-disk metadata from the `ext3` superblock and ACL were packed into extended attributes on the root directory and individual files respectively.

As a result of this, TransCrypt can now be run as a "cryptographic layer" of sorts over the many modern file systems which support extended attributes. More-

over, TransCrypt no longer requires any modifications to existing userspace utilities. Thus, not only is the footprint of TransCrypt on existing code reduced, but a constraint that might have been considered a barrier to adoption is also removed. This is accomplished with a negligible performance penalty.

In addition to this, a powerful test framework has been created, along with a number of test cases to ensure correctness and minimize the possibility of regressions being introduced by future modifications.

6.3 Future Work

With the conclusion of this work and the introduction of the new cryptographic layer [32], the TransCrypt implementation is at a relatively stable and mature state with respect to its kernel architecture. TransCrypt can, with relative ease, be deployed on individual hosts as well as servers which provide a central, secure storage for users in an organization.

The chief limitation faced today is that the current model requires users to access files by logging in to the TransCrypt host itself. The TransCrypt design is, by and large, amenable to use as a network file system. The primary problem that remains to be addressed, however, is that it is not possible for the TransCrypt kernel to differentiate between a genuine user and a user who has assumed the identity of another user. Potential solutions to this problem are yet to be devised.

Another area where there is scope for improvement is the handling of ACL entries for groups, named groups and others. Currently, these types of ACL entries

are simply ignored. This could potentially be a barrier to adoption, since in this particular case, TransCrypt does not adhere to the standard UNIX semantics. Several approaches exist to solve this problem, such as maintaining keys for each group, or expanding each group into its component users when group permissions are used. Each potential solution has its own limitations and needs to be fully explored before any one is adopted.

In order to enhance security even further, it might be desirable to add support for metadata encryption, as well as integrity checks for both data and metadata. `transcrypt-auth` is yet to be extended to support smart cards and other means of authentication. Finally, a significant performance improvement for file creation and open operations, as well as all ACL manipulations can be derived by caching user certificates in memory instead of sending a request to `transcryptd` every time a certificate is required.

Bibliography

[1] Arms dealers got Navy plans and deployment details. Website. http://www.indianexpress.com/story/8028.html.

[2] Symantec: Average Laptop Contents Are Worth Half A Million Quid. Website. http://www.digital-lifestyles.info/display_page.asp?section=cm&id=2960.

[3] Matt Blaze. A Cryptographic File System for UNIX. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[4] dm-crypt: a device-mapper crypto target for Linux. Website. http://www.saout.de/misc/dm-crypt/.

[5] Michael Austin Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. In *Proceedings of the Linux Symposium*, pages 201–218, Ottawa, Canada, July 2005.

[6] Apple Mac OS X FileVault. Website. http://www.apple.com/macosx/features/filevault/.

[7] How Encrypting File System Works. Website. http://technet2.microsoft.com/WindowsServer/en/Library/997fdd99-73ec-4041-9cf4-1370739a59201033.mspx.

[8] TrueCrypt - Free open-source disk encryption software for Windows Vista/XP, Mac OS X, and Linux. Website. http://www.truecrypt.org/.

[9] Satyam Sharma. TransCrypt: Design of a Secure and Transparent Encrypting File System. Master's thesis, Indian Institute of Technology Kanpur, India, August 2006.

[10] TransCrypt Filesystem Homepage. Website. http://www2.cse.iitk.ac.in/transcrypt/.

[11] Stephen Tweedie. The Extended 3 Filesystem. In *Proceedings of the 2000 Ottawa Linux Symposium*, July 2000.

[12] E2fsprogs: Ext2 Filesystem Utilities. Website. http://e2fsprogs.sourceforge.net/.

[13] Jim Mostek, Bill Earl, Steven Levine, Steve Lord, Russell Cattelan, Ken McDonell, Ted Kline, Brian Gaffey, and Rajagopal Ananthanarayanan. Porting the SGI XFS file system to Linux. In *ATEC '00: Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2000. USENIX Association.

[14] JFS project Web Site. Website. http://jfs.sourceforge.net/.

[15] Florian Buchholz. The structure of the Reiser file system. Website. http://http://homes.cerias.purdue.edu/~florian/reiser/reiserfs.php.

[16] James P. Hughes and Christopher J. Feist. Architecture of the Secure File System. In *Eighteenth IEEE Symposium on Mass Storage Systems*, pages 277–290, April 2001.

[17] EncFS: Virtual Encrypted Filesystem for Linux. Website. http://encfs.sourceforge.net/.

[18] Red Hat Inc. Device-mapper Resource Page. Website. http://sources.redhat.com/dm/.

[19] LUKS - Linux Unified Key Setup. Website. http://luks.endorphin.org/.

[20] Clemens Fruhwirth. New Methods in Hard Disk Encryption. Website. http://clemens.endorphin.org/nmihde/nmihde-letter-os.pdf.

[21] B. Kaliski. PKCS #5: Password-based cryptography specification version 2.0, 2000.

[22] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98, Department of Computer Science, Columbia University, 1998.

[23] S. Kent. Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management, 1993.

[24] Extended Attributes and ACLs for Linux. Website. http://acl.bestbits.at/.

[25] Avantika Mathur, Mingming Cao, and Suparna Bhattacharya. The new ext4 filesystem: current status and future plans. In *Proceedings of the 2007 Ottawa Linux Symposium*, pages 21–34, June 2007.

[26] Draft Standard for Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Programming Interface (API), draft 17 (withdrawn). Website. http://wt.xpilot.org/publications/posix.1e/.

[27] Draft Standard for Information Technology–Portable Operating System Interface (POSIX)–Part 2: Shell and Utilities, draft 17 (withdrawn). Website. `http://wt.xpilot.org/publications/posix.1e/`.

[28] Andreas Grunbacher. POSIX Access Control Lists on Linux. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, pages 259–272, June 2003.

[29] attr - Extended attributes. Manual Page. `attr(5)`.

[30] capabilities - overview of Linux capabilities. Manual Page. `capablities(7)`.

[31] Security-Enhanced Linux. Website. `http://www.nsa.gov/selinux/`.

[32] Sainath Vellal. A Device Mapper based Encryption Layer for TransCrypt. Master's thesis, Indian Institute of Technology Kanpur, India, June 2008.

[33] David Howells. [PATCH] implement in-kernel keys & keyring management. Linux Kernel Mailing List, August 2004. `http://lkml.org/lkml/2004/8/6/323`.

[34] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol, July 2003.

[35] netlink - Communication between kernel and userspace (PF_NETLINK). Manual Page. `netlink(7)`.

[36] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., third edition, 2006.

[37] Jean-Luc Cooke and David Bryson. Strong Cryptography in the Linux Kernel. In *Proceedings of the Linux Symposium*, pages 139–144, Ottawa, Canada, July 2003.

[38] B. Kaliski and J. Staddon. PKCS #1: RSA Cryptography Specifications Version 2.0, 1998.

[39] XySSL - Embedded SSL. Website. `http://xyssl.org/`.

[40] Joel Becker. [PATCH] configfs, a filesystem for userspace-driven kernel object configuration. Linux Kernel Mailing List, April 2005. `http://lkml.org/lkml/2005/4/3/112`.

[41] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6*. IEEE, New York, NY, USA, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6.

[42] Star - a very fast and Posix 1003.1 compliant tar archiver for UNIX. Website. http://cdrecord.berlios.de/private/star.html.

[43] Linux Testing Project. Website. http://ltp.sourceforge.net/.

[44] fsx. Website. http://www.codemonkey.org.uk/projects/fsx/.

[45] The ext3-tools package. Website. http://www.zip.com.au/~akpm/linux/patches/stuff/ext3-tools.tar.gz.

[46] Guido van Rossum. *Python Library Reference – unittest – Unit testing framework*. http://docs.python.org/lib/module-unittest.html.