A Device Mapper based Encryption Layer for TransCrypt

Sainath S Vellal

Department of Computer Science & Engineering

Indian Institute of Technology Kanpur

June 2008

A Device Mapper based Encryption Layer for TransCrypt

A Thesis Submitted

In Partial Fulfillment of the Requirements

For the Degree of

Master of Technology

by

Sainath S Vellal

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 *"A Device-Mapper based Encryption Layer for TransCrypt"*, by *Sainath S Vellal*, 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

Data security has come to be of utmost importance in the recent times. Several encrypting file systems have been designed to solve the problem of providing data security in a secure and transparent manner. TransCrypt is such an encrypting file system, which is implemented in kernel space, has an advanced key management scheme and is designed to be deployable in an enterprise scenario. It uses per-file cryptographic keys for flexible sharing and does not include even the superuser in its trust model.

Earlier, TransCrypt was implemented on the Linux kernel (version 2.6). In the implementation, several modifications were made to the existing kernel to embed the TransCrypt functionality. Such modifications also changed the file I/O behaviour in the kernel, in order to add a cryptographic layer to perform encryption and decryption on the file data. The kernel thus modified had several limitations with respect to functionality, maintainability and performance.

In this thesis, we propose a new cryptographic layer for the TransCrypt file system. This layer is implemented as a kernel module and does not modify any existing kernel code. The module uses the device-mapper infrastructure provided by the Linux kernel. The new layer addresses several limitations of the earlier implementation, and is robust and stable. Performance gains of over 90% were observed in read and write operations on large files with the new implementation. The design and implementation details of the new cryptographic layer and performance measurements are discussed in this work.

Acknowledgements

I wish to express my gratitude to my supervisors, Prof. Rajat Moona and Prof. Dheeraj Sanghi, whose guidance and support enabled this work. Discussions with them helped me immensely throughout the design and implementation of this work. I would also like to thank the Prabhu Goel Research Centre for Computer and Internet Security for providing me with the wonderful facilities and freedom that enabled me to undertake this project. I also thank Tata Consultancy Services, Lucknow for partially funding the project.

I would also like to thank my friends, Arun Raghavan and Satyam Sharma, for their co-operation and innovative suggestions. Thanks are also due to my classmates, who created an enjoyable and pleasant environment during my stay in the campus.

Finally, I am forever grateful to my parents, who have loved, supported and encouraged me in all my endeavours.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In the recent times, data storage has become increasingly common and more affordable. Archiving important data on storage mediums like USB disks and file servers is a very common usage scenario among desktop and corporate users. Data security is therefore of utmost importance, especially against data thefts, which impose risks of losing significant personal and organisational data [1, 2]. There is an acute need for a storage solution which uses strong cryptographic methods to protect data.

An *encrypting file system* provides the much needed solution to the problem of data protection. There are several encrypting file systems (see section 1.2) which provide security by encrypting and decrypting data transparent to a user. Although the different encrypting file systems address the problem of data security in different ways, a combination of features such as per-file encryption, flexible key-sharing and exclusion of superuser from the trust model makes the file system more secure and

customizable. *TransCrypt* [3] file system was created on the basis of these features to provide a very strong solution to the problem of securing data in a user transparent manner.

TransCrypt is an enterprise-class, kernel-space encrypting file system for the Linux [4] operating system, which incorporates an advanced key management scheme to provide a high grade of security, while remaining transparent and easily usable. The initial implementation of TransCrypt [5, 6] was carried out as modifications to the `ext3` [7] file system on Linux. Userspace packages specific to the `ext3` file system were also modified. The Linux kernel code undergoes changes periodically as new features and bug fixes are added to subsequent releases. Since a significant part of the TransCrypt file system includes modifications to existing Linux kernel code, changes to the code need to be tracked and updated for every kernel upgrade. Dependency due to modifications to the existing kernel code implies a constraint on the usage of TransCrypt over only the `ext3` file system. The need was felt for a improved TransCrypt file system which is independent of the modifications to the underlying native file system code. This had the potential to exploit the advantages of various other underlying file systems.

Desirable characteristics of an encrypting file system include performance and ease of use, apart from a high grade of security. If a user *perceives* the `read` and `write` operations to be slow on a TransCrypt file system compared to that on a normal filesystem, then a potential wide scale deployment of the encrypted filesystem would be hard to implement. The earlier implementation of TransCrypt was based on the modifications in the file I/O functionality in the kernel. It had several performace and maintenance related limitations. A need was felt to improve on the performance of TransCrypt file system, as well as to improve maintainability of the code by developing

an efficient encryption layer as a kernel module instead of incorporating modifications to the kernel code.

The changes to TransCrypt file system design proposed in this thesis address the maintainability and `read/write` performance issues. The new design attempts to enhance the user experience while reading and writing files on TransCrypt, while striving towards a layered architecture of the implementation, which is independent from the underlying file system. Such a design would then be able to exploit the advantages of other file system implementations.

1.2 Related Work

The solutions to provide data security in the form of cryptographic file systems in the kernelspace are primarily based on two approaches. *Volume encryption* and *File system level encryption*. In volume encryption approach, the data written to the storage device mounted as a volume is encrypted as a whole. A single cryptographic key is used to encrypt both data and metadata of all the files over the entire storage device. Example implementation of volume encryption includes `dm-crypt` [8] and FileVault [9]. File system level encryption approach is used to encrypt file system objects (files, directories and metadata), rather than the storage device as a whole. Different keys are used in this approach for different file system objects. eCryptFS [10], CFS [11] and EncFS [12] are some implementations that use the file system encryption approach.

`dm-crypt` and eCryptFS are two contrasting implementations which are included in the Linux kernel. `dm-crypt` uses the volume encryption approach, whereas

eCryptFS uses the file system encryption approach. The pros and cons of both these approaches, along with a short introduction to their working, are detailed below.

1.2.1 `dm-crypt`

`dm-crypt` is an encryption target (plugin) based on the device-mapper infrastructure included in recent Linux kernels. Device-mapper [13] is an infrastructure which can be used to create layers of virtual block devices over the real block devices to achieve striping, mirroring, etc. `dm-crypt` encrypts the data that is read from or written to the real block devices by transparently encrypting via the virtual device. It uses the native Linux CryptoAPI interface to carry out cryptographic operations. The usage scenario for `dm-crypt` is simple. A new virtual block device is created over the real block device which holds the file system to be mounted. While creating the virtual block device, a cryptographic key, along with the cipher to be used and the IV generation mode is specified. The file system on real block device is then mounted on the virtual block device. Thus, read and write operations on the file system intended for the real block device go through the virtual block device, where data is read from the real device and decrypted before returning the control to the applications and it is encrypted before writing to the real device. The cryptographic operations are carried out using the cryptographic information passed during the initialization of the virtual block device.

The data is encrypted at the generic block layer (explained in section 2.1.3) and just before the data for write operation is submitted to the I/O scheduler for further processing. This ensures a fast performance due to low overhead of I/O processing. The `dm-crypt` implementation also comes with some drawbacks, which are listed as

below.

- All the data that is written to the disk underneath (including metadata) is encrypted.
- It lacks an advanced key management scheme due to the usage of just a single key for the whole volume.
- Flexibility in file sharing is compromised.
- Incremental backups are hard to implement.
- Superuser is a trusted entity in the trust model for `dm-crypt`.

These limitations of `dm-crypt` restrict its use to a smaller segment of users, rather than it being used in large corporates where file sharing and incremental backups happen regularly. Though there is an effort [14] to introduce sharing of `dm-crypt` partitions, lack of a fine grained access is still a problem that is unsolved.

1.2.2 eCryptFS

eCryptFS [10] is a kernelspace cryptographic file system for Linux based on Cryptfs [15]. It is implemented as a stacked file system [16] which can be layered over an already mounted filesystem. ecryptFS positions itself between the VFS layer and the underlying file system layer. Thus, all the I/O to the file system is channeled through eCryptFS, where the data is encrypted or decrypted for write and read operations and is handed over to the lower file system. The native kernel CryptoAPI is used for all cryptographic operations.

A randomly generated cryptographic key is used to carry out encryption on the file for every file residing on eCryptFS. This key is stored on the disk in an encrypted format by contacting a userspace daemon to obtain the key-encryption parameters. The advanced key management scheme employed by eCryptFS is one of the distinguishing features suitable for an enterprise wide deployment of the file system. It provides a fine grain access control and flexibility to share encrypted files with individual users.

A seperate cache is maintained by eCryptFS for pages and file system metadata. Managing this infrastructure is an overhead on the performance of the file system. Further, eCryptFS addresses a very narrow threat model in which the superuser and the userspace are both trusted entities.

1.3 TransCrypt in a Nutshell

An overview of the TransCrypt architecture is illustrated in figure 1.1. The TransCrypt file system design considers the kernel to be the only trusted entity in the system. Hence, a major part of TransCrypt is implemented inside the kernel. Userspace utilities are required for certain key management tasks as described in the work by Abhijith Bagri [6].

When a user creates a file on the TransCrypt file system, a random file encryption key (FEK) is generated. The FEK is further encrypted with a special key known as the file system key (FSK), which is provided by the system administrator at the file system creation time. The so-formed *blinded* FEK is then further encrypted with the user's public key, which is obtained in the form of an X.509 [17] certificate

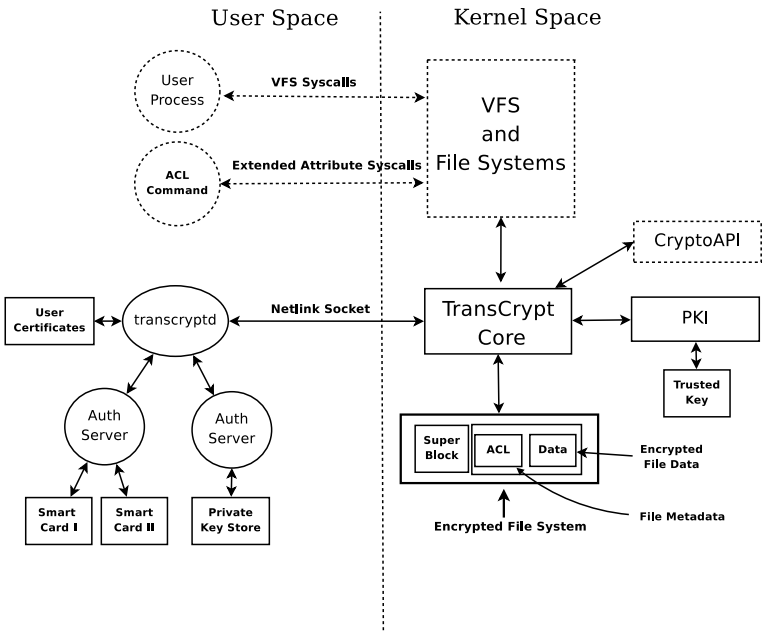

Figure 1.1: TransCrypt Architecture

through the `transcryptd` daemon. This blinded FEK encrypted with the public key of a user forms a "*token*" for the user, and is stored along with the file's metadata.

When an existing file is opened for reading or writing, the token is sent to `transcryptd` which forwards it to `transcrypt-auth` (auth server), which is an interface to the user's private key store (PKS). The PKS is a secure device that can store the user's private key and implement operations using the private key. The PKS can be a smart card or a secure USB key. The token is then decrypted using the user's private key, and the blinded FEK is sent back to the kernel. The kernel decrypts the blinded FEK using FSK which is known only to the kernel, and stores the key for subsequent reads and writes on the file in the kernel.

During file read and write operations, the key for file is obtained from the kernel keyring infrastructure (see section 3.1.2). The cryptographic operations in the kernel are carried out by utilising the native kernel CryptoAPI [18]. A seperate

communication module in the kernel implemented using the Netlink [19] interface is used to interface with the userspace utilities.

When a user, say A, wants to grant access to another user B for a particular file, the token for user A is authenticated with the PKS and the blinded FEK is obtained. This is encrypted with user B's public key obtained via `transcryptd` and a new token for user B is created and stored along with user A's token in the metadata of the file.

An unauthorised user will not be able to decrypt the token and obtain the FEK, since the blinded FEK is encrypted with the user's public key. The private key is available only with the original user. Blinding the FEK with FSK ensures that FEK is not leaked while in transit from kernel to userspace and back. Thus TransCrypt guarantees that only genuine, authorised users can access a file and no other user (even with superuser privileges) can deduce FEK for the file. A more detailed explanation of the TransCrypt components and the modifications to the kernel code is provided later in chapter 3.

1.4 Scope of this Work

TransCrypt's initial implementation involved modifications to the Linux kernel code. A major part of the modification included changing the file I/O behaviour to force encryption at a layer closer to the page cache. This implied a lower performance for `read/write` operations and a diversion from the optimised page cache behaviour in the kernel for I/O.

This thesis discusses the original design and implementation of the TransCrypt

filesystem in detail. It outlines the shortcomings of the earlier design of the encryption layer that resulted in read and write performance overheads. A new design of the encryption layer for TransCrypt utilizing the device-mapper (DM) infrastructure of the newer Linux kernels is presented in this thesis. The advantages of using the DM infrastructure is discussed. Implementation details of the new design are provided. Performance tests conducted on the new implementation against the earlier implementation, along with unencrypted file systems are plotted and the results are analysed. An attempt to make TransCrypt more maintainable by modularizing the encryption layer is presented in this work.

1.5 Organization of the Thesis

The rest of the thesis is organised in the following manner. In chapter 2, some essential kernel concepts and layers required for the understanding of the TransCrypt design in the Linux kernel are introduced. Also, the modifications made at the respective layers for TransCrypt are indicated in brief. In chapter 3, the TransCrypt architecture, along with its components involved, and the modifications to file I/O to plug in the cryptographic layer at the page cache are discussed in detail. In chapter 4, the advantages of using the device-mapper layer for plugging in the cryptographic operations for TransCrypt are described. The design and implementation details of the new device-mapper target for TransCrypt are also provided. In chapter 5, the details of the performance tests conducted with the new implementation are provided. The results are compared with the earlier implementation. As a conclusion, notes on the future work is provided in chapter 6.

Chapter 2

Background

Certain components of the Linux kernel were modified while implementing TransCrypt file system. In this chapter, we provide a brief introduction to the relevant components of the kernel. We also describe the device-mapper infrastructure, which is used by the new implementation of TransCrypt.

2.1 File I/O in the Kernel

The Linux kernel components can be differentiated into layers based on their functionality. Figure 2.1 illustrates such layers in the kernel involved in file I/O.

2.1.1 Virtual File System

The Virtual File System (VFS) is a subsystem of the Linux kernel which is positioned on the top of all other file system components. It groups the common functionality

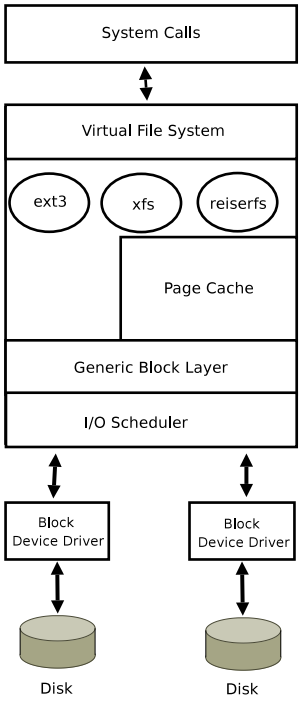

Figure 2.1: Kernel Components along the file I/O path [20]

amongst the underlying file systems and provides a common system call interface to the userspace. By providing a common interface, the VFS creates an abstraction layer over different file systems and provides a unified way of reading and writing files, creating directories etc. without having to know the file system specific interface.

The VFS stores information describing a particular file system in a *superblock* data structure. It keeps track of the files that are currently open through the *file* and *inode* data structures. The *file* structure is used to represent an open file for a process on the file system. It is a data structure local to the process. The *inode* structure represents the metadata of the file in memory. Additionally a *dentry* structure is used to cache the mapping between recently accessed file system paths and corresponding inodes, and hence speed up the file look up process.

The VFS also stores pointers to functions which operate on the VFS data

structures. When the file system is mounted, the VFS function pointers are populated with file system specific implementations. For example, the *file* structure has `file_operations` structure, which contains pointers to file system specific `read` and `write` routines. When an `ext3` file system is mounted, `ext3` specific read and write functions are assigned to the `file_operations` interface, which is shared by all the *file* structures across the mounted file system. During a `read` or a `write` operation on a file, the control is transferred to the file system specific `read` or `write` implementations using these function pointers.

2.1.2 Page Cache

Accessing data from disk is very slow compared to the access from physical memory. However, data access patterns tend to show *spatial* and *temporal locality*. Therefore, storing the disk data in physical memory is beneficial. It improves the file system performance by reducing the number of disk accesses.

The page cache is used by the kernel to manage disk data cached in physical memory. The unit of storage in the page cache is a *page*. Physical memory is divided into *pages*, which are the basic unit of memory management. `struct page` is the data structure used to describe attributes of a page in the physical memory. `struct page` also contains the virtual memory address of the physical memory page. The pages, apart from caching the data from the disk, also keep other data such as shared memory between processes in the kernel. The type of data that is kept in a page is indicated by an attribute in the corresponding `struct page` data structure.

The disk presents data in units of *sectors*. Managing data at the level of sectors is an expensive operation, since it requires free space management in units of sectors.

To avoid large data structure on the disk, file systems group contiguous sectors into a *file system block*. A *buffer* is a portion of physical memory which represents a file system block. The file system handles the mapping between disk blocks and buffers. The data that the buffer points to the file system data block stored in a page points to, is stored in a page. The size of a page is an integer multiple of the size of a buffer.

A page in the page cache can contain data from multiple buffers which are contiguous on the disk. Buffers which are not contiguous on the disk, and buffers representing metadata for a file, are stored in special pages called "buffer pages". The buffers are tracked in the kernel using a `buffer_head` data structure. The `buffer_head` structure is shown below.

```
struct buffer_head {
    struct page   *b_page;   /* page which has the buffer */
    size_t         b_size;   /* size of the buffer */
    char          *b_data;   /* start offset of the buffer */
    ......
};
```

Figure 2.1 illustrates the positioning of the page cache. It is located just below the file system implementations, and just above the generic block layer. The page cache can be accessed directly by the VFS layer (when a file is `mmap`ed) or through the upper file systems.

A user process views a file as a container for a stream of bytes. However, the file may be spread across several non-contiguous locations on the disk. A `read()` or `write()` operation on the file requested by an application as a stream of bytes may result in requests for fragments of data. The fragments of data can be spread over multiple file system blocks (buffers) and over multiple pages. If the data contained in the pages is also contiguous on the disk, then the entire request can be represented

by a single data structure. Such an aggregation is carried out in the kernel with the help of a BIO (`struct bio`) data structure. The BIO is a data structure used by all layers below the VFS to represent the I/O for a file. Additionally, a `buffer_head` data structure is used to track I/O completion on a BIO, from the layers above the generic block layer.

During a file `read` or `write` operation, the VFS functions invoke the file system specific `read` or `write` methods. The file system specific functions then call upon the page cache functions. During a `read()` operation, when the pages mapped to the file are accessed for the first time, page cache functions populate the pages with the data from the disk. The disk data is brought in units of buffers from the disk to the pages in the page cache. Subsequent `read()` requests for data are accessed via the page cache.

If the data present in the pages is already synchronized with the disk, then the data is returned back to the caller (system call). If the pages are not in sync with the disk, then the corresponding file system methods are invoked to fill the page content from the disk.

In the case of a `write()` operation, the pages containing data are marked *dirty*. The dirty pages are traversed by a seperate kernel thread (`pdflush`) periodically and written to the disk by calling the file system methods.

The file system methods group the pages containing contiguous data into a single BIO and send the BIO to the lower layer for further processing. If the page is a *buffer page*, then a seperate BIO is created for each buffer and handed over to the lower layer. Seperate `buffer_heads` are created to keep track of the state of each buffer in a buffer page.

Applications which implement their own caching mechanisms would take advantage of direct I/O to the block device. In the case of a direct I/O request, the page cache is bypassed and the generic block layer is accessed directly.

2.1.3 Generic Block Layer

The generic block layer is a layer below the page cache, which handles I/O requests for all block devices. It offers an abstract view of the block devices by providing general data structures which describe "disks" and "disk partitions".

In the earlier days, I/O requests by the applications were not as large as they are in the recent times. Therefore, I/O requests for a portion of contiguous virtual memory could be satisfied by contiguous physical memory. However, with the increasing demand for larger I/O by applications such as databases etc., the contiguous virtual memory was mapped to non-contiguous areas in physical memory. Therefore, the file data could be scattered over multiple non-contiguous physical locations in the memory, but still be contiguous on the disk. A data structure which could represent file I/O contiguous on disk by gathering all the scattered pages in physical memory was introduced in the recent kernel (version 2.6.9). This is achieved by keeping an ordered list of physical pages known as scatter-gather list. A data structure known as *BIO* is used to describe a signaler request of I/O and contains the scatter-gather list.

The BIO structure is a lightweight container for block I/O. The structure with some of its members is as shown below.

```
struct bio {
   sector_t                bi_sector;
   struct block_device     *bi_bdev;
   unsigned short          bi_vcnt;
   struct bio_vec          *bi_io_vec;
   bio_end_io_t            *bi_end_io;
   bio_destructor_t        *bi_destructor;
   ....
};
```

It has a pointer `bi_bdev`, which points to the block device involved in the I/O. The device may be a representation of a partition on the disk or the disk itself, if partitions are not used. `bi_sector` is set to the initial sector number of the disk block on this device corresponding to the I/O. `bi_end_io` represents a function which is invoked after the completion of the I/O.

The scatter-gather mechanism to address the data memory is implemented in a `bio_vec` data structure. The BIO structure contains a pointer to the list of `bio_vec` structures in `bi_io_vec`. The `bio_vec` structure is as shown below.

```
struct bio_vec {
    struct page *bv_page;
    unsigned int bv_len;
    unsigned int bv_offset;
};
```

Each bio_vec structure represents data which is contiguous on disk. It contains a pointer to the page (`bv_page`) which contains the data involved in the I/O. `bv_len` and `bv_offset` represent the length of the data and the offset in the page where the data begins respectively.

In the earlier kernels (versions before 2.4), the `submit_bh` function was used as the entry point to the block layer for all I/O. The `buffer_head` structure was

the data structure used to handle kernel input-output. In the current Linux kernel, the `buffer_head` structure is used to keep track of the state of the buffer within a page and as a wrapper for BIO to maintain backward compability. BIO is the data structure used for all I/O in the kernel.

Maintaining a BIO structure helps to keep track of the I/O in terms of pages, rather than buffers. This new scheme of having the BIOs as containers for I/O in the kernel has several advantages over the previous scheme (with buffers) as given below.

- BIOs are very lightweight and contain only as much information required to represent a block I/O as needed.
- The new scheme can perform scatter-gather I/O, since the BIO contains pages referring to different physical memory locations. Thus, the split I/O operations can be represented easily without having to maintain many bufferheads. Further, scatter-gather DMA operations can also be supported.
- The BIO represents I/O for regular pages (page-cached) as well as direct I/O.
- The lower block drivers can place the data (after I/O completion) in a process's user memory directly. This avoids placing the data in a kernel buffer and then transferring to the user memory.

During a read or write operation in the kernel, BIOs are created in the page cache to represent groups of aggregated contiguous disk data. These BIOs are handed over to the generic block layer by the page cache routines. Every block device driver has a *request queue* in which *requests* (which consists of a set of BIOs) are queued for further processing by the I/O scheduler. Upon entry to the generic block layer, the BIO is placed in a request queue of the block device driver. The block device

driver then invokes a strategy routine on the request queue and sorts the BIO requests accordingly. The strategy routine implements the disk scheduling algorithm as needed by the device. The requests are then processed in the sorted order by the disk controller. Once the disk controller completes an I/O operation, it notifies the block layer by raising an interrupt. The BIO's completion handler is invoked. This notifies the upper layers of the completion of I/O. Thus, the file I/O cycle is completed.

2.2 Device Mapper

Device Mapper (DM) [13] is a virtual block device driver which provides an infrastructure to filter I/O for block devices. It provides a platform for filter drivers (also known as *targets*) to map a BIO to multiple block devices, or to modify the BIO while it is in transit in the kernel. A few example targets include the software RAID implementations, Logical Volume Management [21] (LVM2) and disk encryption target [8] (`dm-crypt`).

2.2.1 Overview

Figure 2.2 shows the position of the device mapper layer in the kernel I/O architecture. It is positioned between the generic block layer and the I/O scheduler. Since DM itself is a block device driver, it registers the functions to handle block I/O with the generic block layer. These functions transform the received BIOs and pass them to corresponding functions from the target device drivers for further processing.

The DM block device driver exports a set of `ioctl` methods which are used by a userspace program (`dmsetup`, explained in section 2.2.2). `dmsetup` creates a mapping

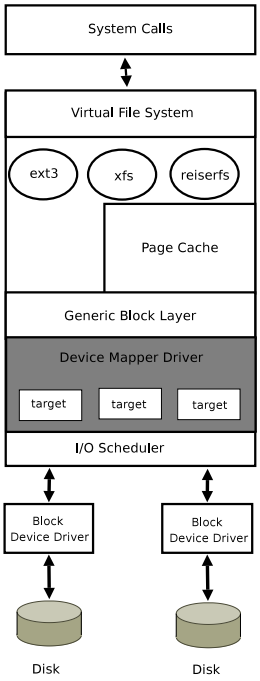

Figure 2.2: The Device Mapper Layer

between the sectors of the virtual block device and the sectors of the real block device. When this mapping is created, a data structure (*mapping device*) is generated, which stores all the information about the target and the underlying block drivers. The information regarding the underlying block drivers is stored in a configuration table in the kernel memory.

When the generic block layer receives a BIO for an I/O, the BIO is plugged into the request queue of the DM block driver. The DM driver now processes the BIO as follows.

1. The BIO is cloned and the end-of-I/O completion handler for the cloned BIO is set to that of DM's end-of-I/O handler.

2. The targets for the BIO are looked up in the list of targets, and the cloned BIO is handed over to the appropriate target implementation.

3. The target implementation processes the cloned BIO and modifies the data contained by the BIO.

4. The target driver directs the BIO towards the underlying block device that was mapped by the DM layer earlier, sets an appropriate end of I/O handler for the BIO and invokes the entry method `generic_make_request()` for the device driver.

5. Upon completion of the I/O request by the device driver, the cloned BIO's end-of-I/O handler invokes DM block driver's end-of-I/O handler, which then notifies the upper layers about the completion of I/O.

The above process is depicted in figure 2.3. The transformation (cloning) of BIOs at the DM driver might sometimes involve splitting BIOs and cloning, because a single mapping can be created to span multiple devices, which can have different hardware geometries [21]. When the mapping is removed after completing all the operations on the block device, the information stored about the target in the DM driver and the target module are also removed from the kernel.

2.2.2 `dmsetup`

`dmsetup` [22] is a userspace utility, which manages the virtual devices that use the device-mapper driver. A table specifying the target mapping for the virtual device is used while creating the device. This table maps each sector of the underlying device to the virtual device. The table is then handed over to the DM driver in the kernel, by issuing appropriate `ioctl` commands to create the device. In this way, the DM driver, and eventually the target, will have knowledge of the disk geometry

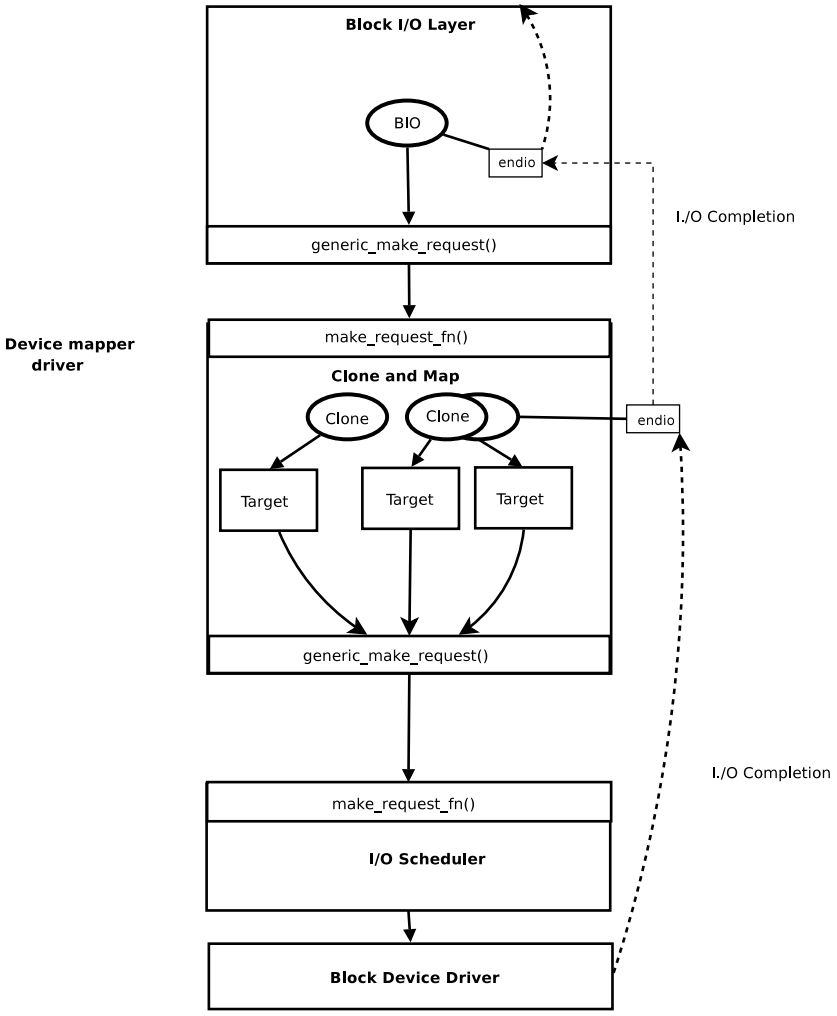

Figure 2.3: Device Mapper driver in action

needed to create the mapping, prior to starting the operations on the block device. `dmsetup` can also be used to safely pass arbitrary parameters to the target. For example, the cryptographic keys in case of targets performing encryption on data. The cryptographic key used to encrypt data can be stored in the target's private memory for further usage.

An example usage of `dmsetup` is as follows.

```
bash# dmsetup create device_name table
creates a virtual device
where device_name = transcrypt (creates /dev/mapper/transcrypt )
      table = <start sector>  <sector count> <target type> <arguments>

bash# dmsetup ls
lists all the mappings.

bash# dmsetup remove device_name
removes a mapping already created.
```

Chapter 3

TransCrypt Implementation

TransCrypt was originally implemented on the Linux kernel (version 2.6.11). It was later ported to kernel version 2.6.23. Modifications were carried out on the existing kernel components and userspace utilities during the implementation of TransCrypt. We describe the TransCrypt architecture that existed on the kernel version 2.6.23 in this chapter.

3.1 In-Kernel Architecture

TransCrypt is an encrypting file system implemented in the kernel space. Kernel is the only trusted component in the trust model of TransCrypt design which does not include even the superuser. Hence, the in-kernel architecture of TransCrypt plays a prominent part in the design of a secure encrypting file system.

Figure 3.1 illustrates the basic interaction between the kernel components that include TransCrypt to achieve permission checks, key storage and cryptographic

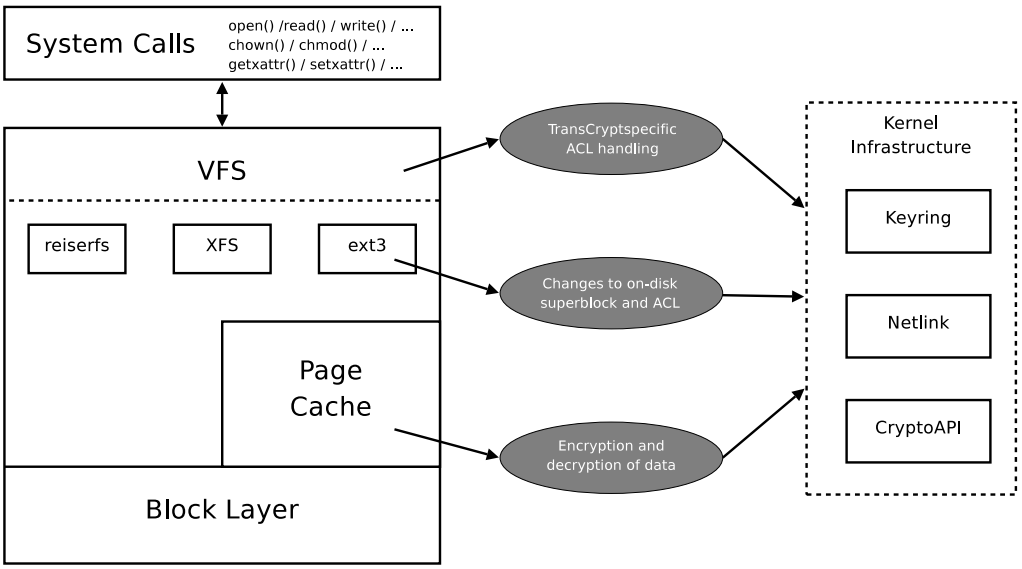

Figure 3.1: TransCrypt modifications in the kernel

operations. The kernel components modified in the earlier implementation of TransCrypt [5, 6] included the VFS, page cache and the `ext3` file system. These modifications and other related components introduced during TransCrypt implementation are presented below.

3.1.1 VFS Modifications

The VFS layer was modified to include changes for TransCrypt. The modifications included changing in-kernel and on-disk data structures for POSIX ACL [23] entries to include two new fields – *certid* and *token*. These fields were specified in the extended ACL of a file for a user. Since the usual behaviour of the POSIX ACL does not include the creation of extended ACLs by default, the VFS layer was modified to create extended ACLs on files. Further, changes to the VFS also accounted for the removal of *groups* and *other* permission checks, since the tokens in the TransCrypt file system are only user-specific.

3.1.2 Keyring

The kernel provides an infrastructure to store cryptographic keys in memory and search through them. These keys are subjected to access checks and are always available by keeping in unswappable kernel memory. TransCrypt utilizes this infrastructure to create a new container for its cryptographic keys, while the file system is mounted.

In the kernel, a `struct key` data structure is used to store a key. Every `key` has a "*description*", using which it can be searched. A *keyring* is a special type of key which contains a list of `keys`. Keyrings implement their own methods of handling the `keys` in this list including addition and removal of the `keys`.

A new TransCrypt keyring which acts as a container for TransCrypt related cryptographic keys was created as part of the implementation. When a file is opened for the first time on a TransCrypt file system, the FEK is retrieved, and stored in the TransCrypt keyring. The corresponding "description" of the key refers to the *inode* (i.e. the inode on which the operation is being carried out) the file. During the subsequent read and write operations to the file the FEK is obtained by searching the TransCrypt keyring for the key based on the file's inode. During an addition of the key, if the keyring is found to be full, the space is created for the key by searching and removing the unused keys. For this purpose a reference count is kept with the keys which is incremented at the time of open and decremented at the time of close. This process of removal of keys in the key ring is known as *key reaping*.

3.1.3 CryptoAPI

The in-kernel CryptoAPI [18] interface provides a generic infrastructure to use several cryptographic algorithms for carrying out symmetric cryptography and generating cryptographic hashes. TransCrypt uses this infrastructure to carry out encryption and decryption of file data stored in the page cache. Cryptographic keys used for this purpose are obtained from the TransCrypt keyring infrastructure.

The TransCrypt file system performs all cryptographic operations in the kernel. Therefore, the cryptoAPI was extended to include the asymmetric cryptographic operations (in particular, encryption and signature verification). The extended CryptoAPI is used by the token creation and modification logic in TransCrypt. The modifications also included the support for RSA [24] operations, and parsing and verification of X.509 certificates [17] by porting X.509 module from XySSL [25] library to the kernel.

3.1.4 `ext3` Modifications

The `ext3` file system is a popular file system native to Linux. The implementation of the `ext3` file system was modified to meet the needs of TransCrypt. Modifications to the `ext3` code included changing the `ext3_super_block` structure to accommodate the cryptographic hash of the file system key (FSK) and related parameters for Password Based Key Derivation Function [26] (PBKDF2). Other TransCrypt specific cryptographic parameters such as algorithms used for bulk and key encryption were also stored in the super block while creating the file system. The in-kernel and on-disk superblock structures were synchronized with the addition of new fields to hold TransCrypt specific data.

The file system behaviour was modified to check for the existence of TransCrypt volume. This was achieved by incorporating a flag in the `ext3_super_block` structure at the time of mounting the file system. All other TransCrypt modified layers in kernel could verify whether a file belonged to the TransCrypt file system or not by checking the presence of this flag and taking appropriate action. In this way, a TransCrypt specific behaviour was introduced in the kernel components.

Token creation during ACL initialization was also added to the set of changes in the `ext3` file system code. The cryptographic operations involving the tokens were handled through the TransCrypt modified CryptoAPI.

3.1.5 Page Cache

The cryptographic layer for TransCrypt performs encryption and decryption of file data inside the kernel. A strategic point to place this layer would have been in the VFS functions, just before transferring data between kernelspace and userspace. The file data would have to be encrypted and decrypted for every `read` or `write` operation in this design approach. Hence, the approach was discarded in favour of a design, where the cryptographic layer is placed just below the page cache. This design would help in storing data in plaintext in the page cache at all times and was efficient in terms of I/O performance.

During a file I/O, contiguous data spread over different physical memory locations, are aggregated into a single BIO. This BIO is submitted to the generic block layer for further processing. In the case of a page containing buffers, seperate BIOs are created for each buffer and seperate bufferheads are used to keep track of the buffers. The function `submit_bh()` is used to pass the buffers to the generic block

layer.

This normal behaviour in the page cache is modified in the TransCrypt implementation. Data belonging to the TransCrypt file system, irrespective of being contiguous or not, is treated as data belonging to a buffer. This data is split into blocks, each the size of a buffer, and submitted to the generic block layer in seperate BIOs. All the TransCrypt data passes through the `submit_bh()` function.

The cryptographic operations on file data were implemented in `submit_bh()` function in the modified TransCrypt implementation. Encryption and decryption in this function was implemented for buffer sized data. The *cryptographic context* for encryption/decryption, which is a structure used to describe the cryptographic operation parameters such as the key, encryption cipher, etc. in the kernel, was obtained from the TransCrypt keyring for the particular file.

Data being written to the disk was encrypted *in-place* before submitting to the lower layers. Data read from the disk, was decrypted before handing the control back to the upper layers. In order to keep the page cache populated with plaintext data, the in-place encryption was followed by a decryption for every `write` operation.

These modifications changed the optimized file I/O behaviour in page cache. Even though this approach for file data encryption worked, there was a penalty on the performance because of the changes in the page cache and in-place encryption.

3.1.6 Kernel-Userspace Communication

A TransCrypt enabled kernel communicates with the userspace in order to retrieve user certificates and to perform token decryption. To enable this communication

a framework was designed as part of the TransCrypt implementation [27]. This framework used `Netlink` [28] sockets to communicate with the userspace daemon (`transcryptd`).

Every time a file is created or opened on the TransCrypt file system, the kernel requests `transcryptd` for the certificate of the user and establishes a secure communication channel to the private key store interface to decrypt the token for the user. Requests for certificates and token decryption follow a secure communication protocol [6] and are sent over the netlink socket to the userspace.

A queue is maintained for the requests sent by the kernel. When the response for a request is received, the corresponding process which issued the request is woken up and the entry for the request is removed from the queue. This communication infrastructure is scalable during authentication of multiple processes opening multiple files on TransCrypt.

3.2 Userspace Components

TransCrypt has a set of userspace utilities which aid in the cryptographic key management of the file system. Some existing utilities were modified to support TransCrypt operations. Also, some new utilities were introduced.

3.2.1 `libacl` Modifications

POSIX ACLs for a file are manipulated using userspace utilities such as `setfacl`, `getfacl` and `chacl`. TransCrypt uses ACL utilities to handle the tokens for a par-

ticular file. These utilities internally depend on the `libacl` [29] library for their functioning.

The userspace utilities read the existing ACL entries and modify them, before writing them back to the disk. Since the in-kernel ACL structures are modified to contain TransCrypt specific information, the library (`libacl`) which handles the ACL manipulation in userspace, is also modified. This modification is required in order to synchronize the way the ACL structures are interpreted by the kernel and userspace.

3.2.2 `e2fsprogs` Modifications

`e2fsprogs` [30] is a suite of userspace utilities, which are used to create and maintain `ext2` and `ext3` file systems. The modifications in the `ext3` file system code in the kernel required corresponding changes to the userspace utilities in order to synchronize the data structures used by the kernel and userspace.

`mkfs.ext3` is the utility in `e2fsprogs` suite which is used to create `ext3` file system on a volume. The modifications in the TransCrypt implementation included changing the `mkfs.ext3` utility to embed a set of TransCrypt related parameters such as cryptographic hash of FSK, bulk and key encryption algorithms. This TransCrypt specific information is used to authenticate the file system mounting process. The changes also included modifications in the superblock data structure of the file system to indicate the presence of a TransCrypt volume.

3.2.3 Utilities

Userspace utilities which aid in the creation and mounting of the file system are implemented for TransCrypt. These utilities also include certain scripts which help a user in managing the certificates for TransCrypt. Private keys of the users can be managed using these helper scripts.

Amongst these utilities, `mkfs.transcrypt` and `mount.transcrypt` are the most prominent ones. `mkfs.transcrypt` is implemented as a wrapper over the `mkfs.ext3` utility. The user is prompted for a passphrase by `mkfs.transcrypt` while creating the file system. A cryptographic key (FSK), based on PBKDF2 [26], is generated using the passphrase provided. The modified `mkfs.ext3` is then invoked under the hood by `mkfs.transcrypt` with the cryptographic hash and PBKDF2 parameters.

`mount.transcrypt` is implemented as a wrapper over the `mount` utility for mounting TransCrypt partitions. It is used to prompt the user for a passphrase, convert the passphrase to a cryptographic key (FSK) and hand the key to the kernel.

3.2.4 Daemons

TransCrypt provides two userspace daemons which aid the kernel in managing the tokens and handling the certificates of the user. `transcryptd` is a daemon implemented in userspace which helps the kernel in handling the certificates, whereas `transcrypt-auth` is another daemon in userspace which acts as a private key store (PKS) interface to the user's private key.

`transcryptd` communicates with the kernel using the netlink communication infrastructure. It handles requests for certificates from the kernel. It also forwards requests from the kernel to `transcrypt-auth` and the response back to the kernel.

An encrypted session is established between `transcrypt-auth` and the kernel through a series of protocol exchanges. `transcrypt-auth` acts as a PKS interface. It handles the sessions that originate from within the kernel, decrypts the tokens and returns back the decrypted tokens to the kernel through `transcryptd`. In the current implementation, it reads the private keys of the users from a secure file store in the userspace. There are plans to extend this to a secure private key store such as a smart card in the near future.

Chapter 4

A New Cryptographic Layer for TransCrypt

TransCrypt[5, 6] was developed earlier as a simple and quick prototype implementation. Such a simple implementation had its own set of limitations.

The early prototype was based on modifications to the `ext3` file system and to the page cache in the kernel. The file I/O behaviour in the page cache was modified in this implementation. But the modifications conflicted with several optimizations in the original page cache. These optimizations were therefore not applied in the case of files belonging to TransCrypt file system. Additionally, TransCrypt could not support direct I/O operations, which are extensively used by applications such as databases etc., as the page cache was bypassed for direct I/O operations.

The cryptographic layer in the earlier TransCrypt implementation was developed as a set of patches to the existing kernel code. As time progressed, it became increasingly difficult to maintain the modifications for TransCrypt against the newer

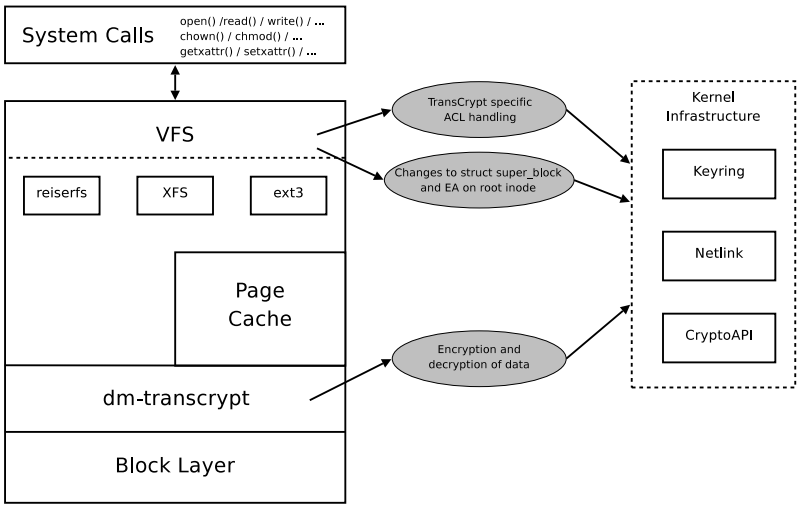

Figure 4.1: The new TransCrypt kernel

Linux kernels. Therefore, a new design for the cryptographic layer, which overcomes the limitations posed by the earlier design is proposed in our work. The modified design is based on the device-mapper 2.2 infrastructure of the Linux kernel. In our design the cryptographic layer is implemented as a kernel module (`dm-transcrypt`) to the DM infrastructure.

Parallel work by Arun Raghavan [31] moved the file system specific modifications which were implemented only for the `ext3` file system to a more generic VFS layer to make TransCrypt design independent of the file system. Figure 4.1 sketches the new modifications in TransCrypt induced by our work, in contrast to the earlier modifications (figure 3.1). As can be seen from the comparison, the proposed design removes the dependency on the page cache, thus making the implementation more clean and efficient.

The new design for the cryptographic layer for TransCrypt which is based on the device-mapper infrastructure is described below.

4.1 Design

The modifications in the earlier TransCrypt implementation affected the normal file I/O behaviour, which was otherwise optimized. The decision to opt for the new cryptographic layer has its roots in the advantages that the DM layer provided, compared to the older TransCrypt implementation. These merits are listed below.

- The new cryptographic layer is in a kernel module and is separated from the rest of the changes to the existing kernel. Any changes to the module could be easily maintained against different kernel versions. This would increase the maintainability of TransCrypt code.
- File I/O including direct I/O could be supported by the new cryptographic layer, since different I/O paths in the kernel converge to the generic block layer and DM infrastructure is placed under the generic block layer.
- The DM target can work on a set of large contiguous data, spread across multiple pages, as compared to smaller data blocks (buffers) in the earlier implementation. Therefore, the performance benefits of this functionality could be exploited.

The device-mapper infrastructure provides the file data in transit from the upper layers in the kernel to the target attached to a particular virtual block device. The TransCrypt cryptographic target (`dm-transcrypt`) receives the data and modifies the data by performing the encryption/decryption operations. Modified data is then handed over to the lower layer for the I/O operation.

Since TransCrypt uses separate cryptographic key for each file, `dm-transcrypt` needs to perform a look up for the right key to encrypt or decrypt data. The look

up is achieved by querying the TransCrypt keyring for the cryptographic context belonging to the file data. The cryptographic context for the data is set when the data is received by the target. In-kernel CryptoAPI is used to perform the encryption or decryption work, as required, using this cryptographic context.

When a volume with TransCrypt file system is mounted with encryption disabled, the data read by the application from userspace is presented only in ciphertext. Therefore, for such a case, `dm-transcrypt` uses the cryptographic context of a *null* cipher algorithm, which does not perform any cryptographic operation on the file data.

During a `read` operation, `dm-transcrypt` performs an *in-place* decryption on the data *after* the I/O completion by the lower layers. In the case of a `write` operation, data to be written to the disk is cloned into a set of pages maintained by the target. The cloned data is encrypted by the cryptographic layer *before* submitting it to the lower layer for I/O. After the encrypted data is written to the disk, `dm-transcrypt` is notified about the I/O completion. The cloned data is then destroyed, while the original request still contains plaintext data. The data is then returned back to the upper layers. Thus, cloning data during a `write` operation saves an extra decryption, as compared to an in-place encryption, which would need an extra decryption to keep the data in the page cache as plaintext. Since the cryptographic operations are expensive, a significant gain on performance is obtained as compared to the earlier implementation.

The cryptographic operations on data is performed per page in `dm-transcrypt`. In terms of performance, per-page encryption is better than per-buffer encryption as was done in the earlier implementation. This is primarily because it results in fewer

calls to the CryptoAPI.

4.2 Implementation

The new cryptographic layer for TransCrypt (`dm-transcrypt`) is implemented as a target kernel module for the device mapper infrastructure on Linux kernel 2.6.24.

The `dmsetup` utility is used to create a virtual block device which maps to one or more real block devices. The name of the target which uses this mapping is specified while setting up the virtual block device. Any further operations are performed on the virtual block device.

The `dm-transcrypt` module is loaded into the kernel, during `dmsetup` time. Upon loading, it registers with the DM infrastructure to receive notifications of I/O on the virtual block device to which the target is attached.

The device mapper driver, which forms the core component of the DM infrastructure, receives the I/O requests. The I/O requests are represented by a BIO structure. The BIO structure represents a set of contiguous disk data spread over multiple locations in physical memory. A layer in the kernel keeps track of a BIO that passes through it, by cloning the BIO structure and setting the end-of-I/O handler function in the BIO to point to the layer's own function. A reference to the original BIO is also embedded in the cloned BIO. In this way, when a lower layer calls the end-of-I/O handler after processing the BIO, the upper layers can be notified of the I/O completion.

The DM driver receives a BIO for each I/O request and clones the BIO structure. If the original BIO is meant for a virtual device on which the TransCrypt file

system is mounted, then the cloned BIO (DM BIO) is handed over to `dm-target`.

The DM BIO is processed further in `dm-target`. The target's implementation is defined by three methods — `constructor()`, `destructor()` and `map()`. These methods are registered with the DM driver, when the target is loaded in the kernel. The implementation details of `dm-transcrypt` are given below.

4.2.1 Constructor Method

The `constructor()` function of the target is invoked by the DM driver, when a mapping is created between the virtual device and the real device through `dmsetup`. It is used by the DM target to initialize any target specific data structures and memory. In case of the target being TransCrypt, following actions take place in `dm-transcrypt` when the `constructor` function is executed.

1. The arguments to the constructor function, which are passed through the `dmsetup` utility, are parsed.
2. The cryptographic context for the *null* cipher is setup.
3. The information about the underlying device is stored in a target-specific structure.
4. Kernel memory is allocated for the set of pages that `dm-transcrypt` would need while performing cryptographic operations.

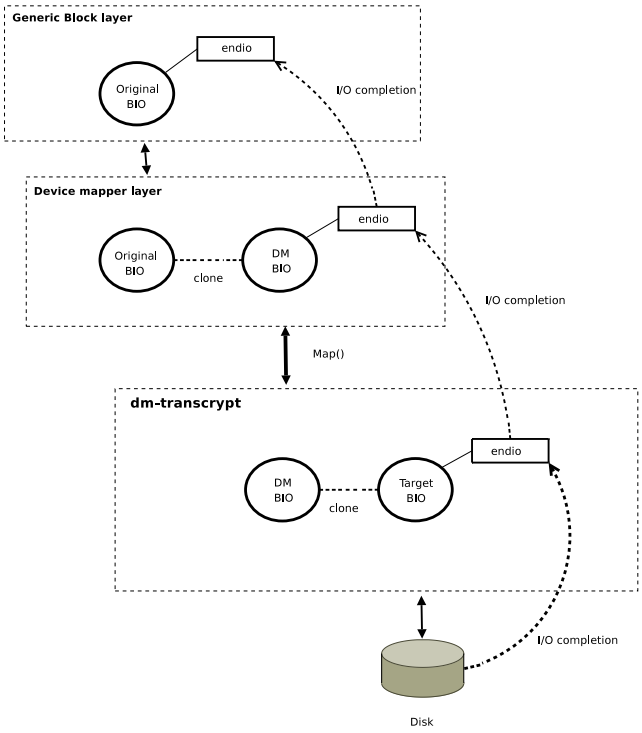

Figure 4.2: Target's `map()` function in action

4.2.2 Map Method

The `map()` function in the DM target implements the entire BIO processing logic. It is executed whenever a BIO is received by the DM driver from the generic block layer. In case of the target being TransCrypt, the BIO is cloned in the `map` function and is tracked by setting an end-of-I/O handler on the cloned BIO. The end-of-I/O handler of the target embeds the decryption logic of data.

Figure 4.2 illustrates the various events in the DM layer and the target. The flow of events inside the `map()` function can be distinguished based on the direction of the I/O request - an incoming I/O for `read` operation and an outgoing I/O for a `write` operation. The following actions take place during a `read` I/O request in `dm-transcrypt`.

1. The DM BIO is cloned. The cloned BIO (Target BIO) still refers to the original pages in memory containing data.

2. The device information in the BIO is set to the underlying device.

3. The target BIO's end-of-I/O handler is set to `dm-transcrypt`'s `endio` function.

4. The target BIO is then submitted to the I/O scheduler layer for further processing. The I/O scheduler submits the BIO to the underlying device driver, which performs the actual I/O from the disk.

5. After populating the disk data into the pages the BIO points to, the disk controller driver signals the I/O completion by invoking the `dm-transcrypt`'s `endio` function.

6. Data decryption is implemented in the `endio` function of `dm-transcrypt`. For decryption purposes, the cryptographic context corresponding to the file data is obtained by querying the TransCrypt keyring.

7. Data contained in every page in the scatter-gather list of the BIO is decrypted using the kernel CryptoAPI.

8. After the data decryption, the target BIO structure is destroyed and the end-of-I/O handler for the BIO received by the `map` function is triggered.[1]

9. The decrypted data in the BIO is passed to the upper layer, by the DM driver.

In a similar way, during the `write` operation data encryption is performed through the DM layer. The flow of events during a `write` I/O request is as given below.

[1]Note: The BIO structure is destroyed, but not the pages refered to by the BIO

1. The DM BIO is cloned. Data in the pages of the scatter-gather list of the incoming BIO are also cloned using a seperate pool of physical memory pages managed by `dm-transcrypt`. The cloned BIO (Target BIO) refers to the cloned data.

2. Per-page encryption is carried out in the `map()` function by obtaining the cryptographic context for the file data from the TransCrypt keyring.

3. The target BIO's end-of-I/O handler is set to `dm-transcrypt`'s `endio` function.

4. The target BIO is submitted to the I/O scheduler layer. The I/O scheduler submits the BIO to the underlying device driver, which writes the data to the underlying disk.

5. After populating the data represented by the BIO into the disk, the disk controller driver signals the I/O completion by invoking the `dm-transcrypt`'s `endio` function.

6. In the end-of-I/O handler, the encrypted pages represented by target BIO are returned back to the page pool and the target BIO is destroyed.

7. The end-of-I/O handler for the DM BIO received by the `map` function is invoked to signal the completion of I/O to the DM layer.

Thus, `dm-transcrypt` performs an encryption for every `write` request and a decryption for every `read` request for a file in the TransCrypt file system. The cryptographic keys are maintained separately for each file.

4.2.3 Destructor Method

After all the operations on the virtual block device are completed, `dmsetup` is used to remove the mapping between the virtual and the real block device. During this stage, `dm-transcrypt's` destructor function is invoked to perform the cleanup of data privately managed by the target. The pages reserved by the cryptographic layer and the *null* cipher are freed.

4.3 An Example Usage

The usage scenario of the TransCrypt file system has changed with the implementation of the new cryptographic layer based on device mapper. Even though it adds a small component of indirection with respect to the block devices used, the benefits of the new implementation far outweigh the minute usage mechanism. An example TransCrypt file system creation and usage is as shown below.

```
(Create a virtual block device /dev/mapper/transcrypt and the mapping)
root@host# echo <start sector> <sector count> <target name>  <real device> |\
                                                    dmsetup create transcrypt

(Create the TransCrypt file system on the virtual device with default parameters)
root@host# mkfs.transcrypt /dev/mapper/transcrypt

(Mount the created file system)
root@host# mount.transcrypt /dev/mapper/transcrypt <mountpoint>
Passphrase: XXXXXX

(Unmount the file system)
root@host# umount <mountpoint>

(Remove the virtual device and the mapping)
root@host# dmsetup remove transcrypt
```

Chapter 5

Performance Evaluation

The wide-scale deployment of any cryptographic file system is heavily dependent on how the users perceive the file system operations. Therefore, performance of a cryptographic file system is as important as the level of security that it provides. TransCrypt, being an enterprise-class cryptographic file system, demands the user experience to be as smooth as that on a regular file system.

File *reads* and *writes* are the most frequent operations in a file system. The performance of a cryptographic file system is therefore dependent on the speed of `read` and `write` operations. The earlier prototype implementation of TransCrypt [5] had severe limitations with respect to the performance of read and write operations. Tests conducted during the implementation showed a performance degradation of more than 250% when compared to a normal file system. Improvements were made over the prototype implementation, as TransCrypt evolved to a mature and stable state.

However, potential problems were identified even in this implementation which

hindered the maintainability and performance of TransCrypt. A new cryptographic layer based on device mapper (DM) was developed to ease the maintenance of the TransCrypt code against newer versions of the Linux kernel. This effort also provided performance benefits over the earlier implementation.

5.1 Experimental Setup

The cryptographic layer affects only the `read` and the `write` path of a file I/O. Thus, it was desirable to measure the read and write performance of the recently modified TransCrypt file system. Improvements in performance, if any, could be measured by comparing the results against the earlier implementation.

In order to measure how the user experience is impacted by the implementation of the new layer, it is desirable to compare the performance results with that of a regular file system. Additionally, the overhead of the DM layer on top of the new cryptographic layer could be quantified too.

Tests were conducted to obtain the `read` and `write` performance measurements of TransCrypt file system with the new cryptographic layer. The performance measurements were taken for the earlier implementation of TransCrypt in which the cryptographic layer was implemented in the page cache. The same set of tests were run on a regular file system (`ext3`) and on a TransCrypt file system mounted with TransCrypt specific options disabled thus bypassing the cryptographic layer.

All the tests were conducted on a freshly booted system with a clear page cache, in order to avoid the effects of caching. The tests were run on a machine having an Intel Core 2 Duo CPU running at 2GHz with 2GB RAM. Separate volumes, formatted

with `ext3` file system having a file system block size of 4KB were used to conduct the tests. On encrypted volumes, the AES block cipher with 128-bit keys were used. The tests were followed by a *sync* operation to flush the buffers to the disk.

The benchmark suite consisted of simple scripts to conduct the experiments. The *dd* utility was used by the scripts to read data from the appropriate input file and write data to the output file. The *time* command was used to measure the time elapsed in executing the scripts. This command reports the total time elapsed (in seconds) and the time spent by the CPU in the kernel mode while executing the specific process.

During an I/O on a large file, the time spent for `read` or `write` operations is significantly more than the time spent for the `open` operation by the kernel. Therefore the overheads due to `open` can be ignored. Thus, large file I/O gives an accurate measure of the time spent by the kernel in performing reads and writes. Further, work by Arun Raghavan [31] discusses the performance overheads caused by `open` system call which includes the time spent in token acquisition and the networking overhead.

The tests consisted of reading and writing a large file (512MB) in units of 16KB. Each test was repeated 10 times and the average of the readings was taken. During the tests to measure the performance of only `read` operations, a randomly generated file of size 512MB is read from the appropriate volume and written to */dev/null*. This ensures that the time spent in writing to the block device is minimized and only the time taken for reading is prominent in the experiment. Similarly, in the case of a performance measurement involving only write operations to a file, the input file is chosen to be */dev/zero* and the output file resides on the TransCrypt file system.

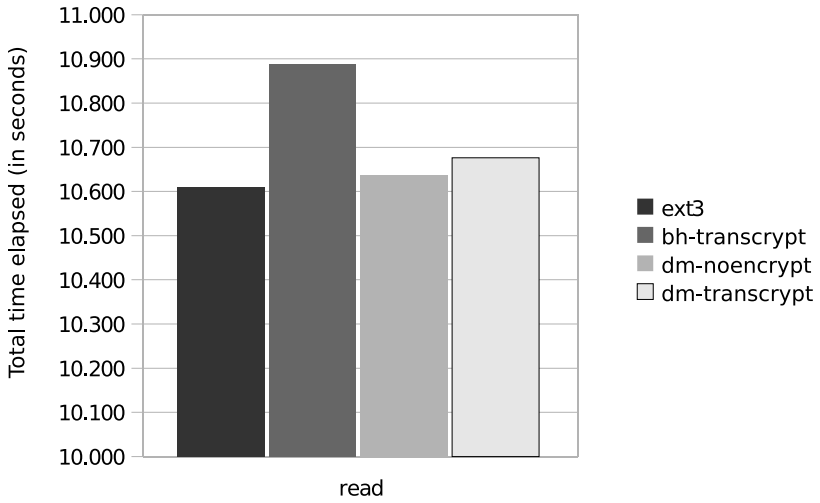

Figure 5.1: File `read` performance - Total time

Reading from */dev/zero* induces very minimal overhead, which can be safely ignored.

5.2 Results and Analysis

The times spent in performing the I/O during the tests are charted in figures 5.1 to 5.4, and the results are analysed separately for the `read` and `write` tests. In the figures, `bh-transcrypt` denotes the earlier implementation of TransCrypt with the cryptographic layer in the page cache. `ext3` refers to a regular `ext3` file system without any TransCrypt modifications. `dm-transcrypt` is the new TransCrypt file system with the DM cryptographic layer, whereas `dm-noencrypt` is a TransCrypt file system mounted with encryption/decryption disabled. All measurements reported in these figures are carried out Linux kernel version 2.6.24.

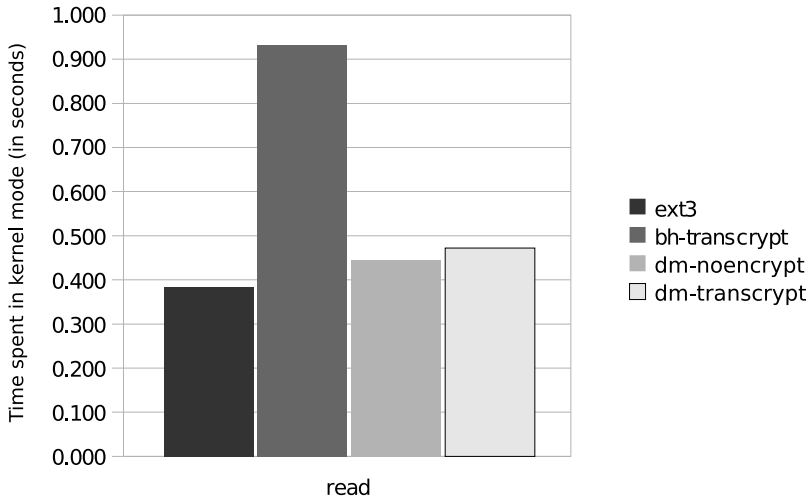

Figure 5.2: File `read` performance - time spent in the kernel

5.2.1 `read` Performance

Figure 5.1 illustrates the total time spent in reading a file of size 512MB from the different file systems. As expected, the `ext3` file system shows the best performance time for the `read` operation. `dm-noencrypt` performance is very close to the ext3. The minute difference between the read times between `ext3` and the `dm-noencrypt` is attributed to the overhead induced by the DM layer in managing the BIOs. This overhead is less than 1% and is negligible.

The decrease in performance of `dm-transcrypt` with respect to `dm-noencrypt` is because of the overhead introduced in cloning the BIOs and decrypting the data. `dm-transcrypt` is comparable in performance to the normal `ext3` file system. For a substantially large file, less than 2% decrease in the total read time is observed. However, the earlier TransCrypt implementation performs very badly under the read test. A 3% decrease in read time is noticed with respect to the normal file system.

The time spent by the CPU in the kernel mode in case of a read operation on

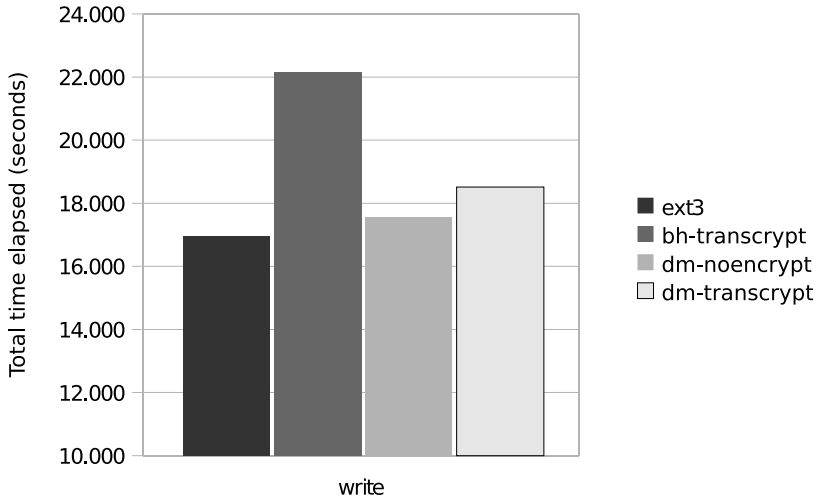

Figure 5.3: File `write` performance - Total time

the different file systems is shown in figure 5.2. This graph gives a good estimate of the actual time spent in doing the cryptographic work, whereas the total read time includes a measure of the time spent in waiting for I/O.

In `bh-transcrypt`, most of the work is carried out in the kernel to maintain the BIOs created by splitting the data into buffer sized chunks. A BIO is created and tracked by a bufferhead for every data chunk so formed. The combined time spent to create and track the data in the modified page cache layer results in a fairly large penalty of read performance for `bh-transcrypt`. By comparing the performance of `dm-transcrypt` and `bh-transcrypt` in the figure 5.2, we can see that the CPU spends about half the time in the kernel for `dm-transcrypt`, when compared to the time in the kernel for `bh-transcrypt`.

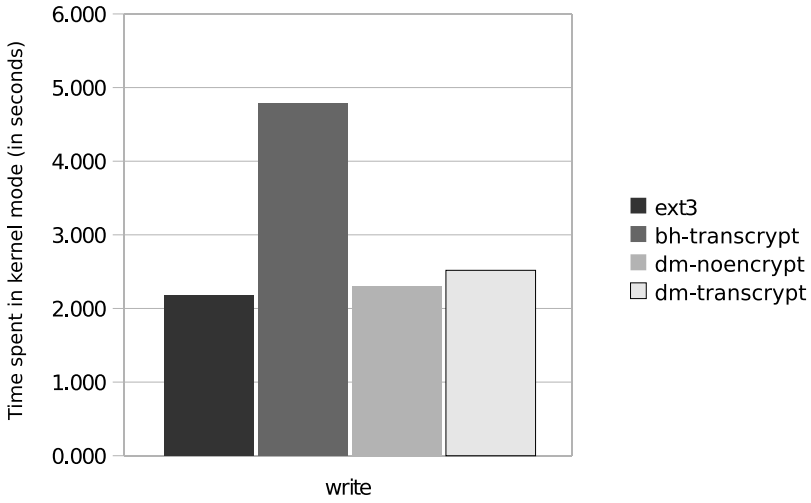

Figure 5.4: File `write` performance - time spent in the kernel

5.2.2 `write` Performance

Figure 5.3 sketches the total time taken by the `write` tests including the time to flush the data into the disk after performing encryption in various different file systems. Figure 5.4 plots the time spent by the CPU in the kernel for the same tests.

The optimization in file I/O behaviour inside the kernel is well exploited by the new implementation to provide a write speed which is very close to the normal `ext3` file system. This can be inferred from the total write time of `dm-transcrypt` with respect to `ext3` in figure 5.3. The performance penalty of `bh-transcrypt` is very high, both in the read and the write operation.

The time spent by the CPU on performing a write operation in the case of `bh-transcrypt` is twice as that of `dm-transcrypt` as shown in figure 5.4. The time spent in creating and tracking several BIOs and bufferheads adds significantly to the penalty incurred by `bh-transcrypt` in the case of a write operation. Additionally, `bh-transcrypt` performs an encryption followed by a decryption for every `write`

request.

In a similar way, the difference in the times spent in the kernel by the CPU for a `dm-noencrypt` and `dm-transcrypt` write operations is negligibly smaller and can be attributed to the encryption overhead.

Clearly, the new design of the cryptographic layer for TransCrypt is superior in performance compared to the older implementations. Also, both the read and write performance for dm-transcrypt are very close to the `ext3` file system performance.

Chapter 6

Conclusion

6.1 Summary

The ubiquitous use of computers for data storage has made data security a prime necessity in several scenarios, in the recent times. The TransCrypt file system provides a secure and efficient storage mechanism to address this need for data security at the enterprise level.

In this work, the dependence of TransCrypt's earlier implementation on the page cache has been addressed. A new cryptographic layer based on the device mapper infrastructure of the Linux kernel has been introduced. As a result of this, the performance of the new TransCrypt file system has significantly improved and is now comparable to the performance of a normal file system.

With this modularization, the cryptographic layer of TransCrypt can now be maintained against future versions of the Linux kernel without any modifications to the existing kernel code.

6.2 Future Work

With the conclusion of this work and with the introduction of the new metadata organization for TransCrypt [31], the TransCrypt implementation is at a relatively stable and mature state with respect to its kernel architecture. TransCrypt can now be deployed on individual hosts and servers with relative ease.

One limitation of the TransCrypt implementation is that the current model requires the users to access the files by logging into the host. In this scenario, TransCrypt could be extended to work over a network as a network file system. Additionally, a clear solution to the primary problem of effectively securing TransCrypt data against identity thefts, including access to the private key store of the user is yet to be devised.

Another area which requires some attention is the handling of ACL entries for *groups* and *others*. Currently these types of ACL entries are ignored by the implementation, although standard UNIX semantics mandate their implementation. Various potential solutions for handling the groups and other ACL entries could be explored in more detail.

It is also desirable to add metadata encryption to further enhance the level of security provided. Data and metadata integrity checks would help detect potential intrusions earlier. A significant performance improvement for file creation and open operations can be obtained by implementing caching of user certificates in the kernel. Finally, `transcrypt-auth`'s functionality can be extended to support smart cards for authentication.

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] Satyam Sharma, Rajat Moona, and Dheeraj Sanghi. TransCrypt: A Secure and Transparent Encrypting File System for Enterprises. In *8th International Symposium on System and Information Security*, 2006.

[4] The Linux Kernel Homepage. Website. http://www.kernel.org.

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

[6] Abhijit Bagri. Key Management for TransCrypt. Master's thesis, Indian Institute of Technology Kanpur, India, May 2007.

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

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

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

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

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

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

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

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

[15] 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.

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

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

[18] 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.

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

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

[21] LVM2 Resource Page. Website. `http://sourceware.org/lvm2/`.

[22] Dmsetup - low level logical volume management. Manual Page. `dmsetup(8)`.

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

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

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

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

[27] Sainath Vellal. Design and Implementation of a Kernel-Userspace Communication Framework for Transcrypt. CS697 Course Report, April 2007.

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

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

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

[31] Arun Raghavan. File System Independent Metadata Organization for TransCrypt. Master's thesis, Indian Institute of Technology Kanpur, India, June 2008.