JOHANNES KEPLER
UNIVERSITÄT LINZ

Author
Oliver Lorenz

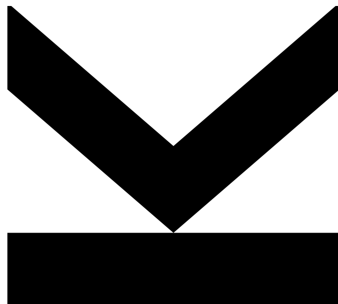
Submission
Institute of Networks and Security

Thesis Supervisor
Univ.-Doz. Dr. Gerhard Eschelbeck

Assistant
Thesis Supervisor
Dipl.-Ing. Dr. Rudolf Hörmanseder

October 2016

cfs: A Sophos SafeGuard Enterprise compatible file system for Linux



Master's Thesis
to confer the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

While in the past typical IT–environments were rather uniform, an emerging trend towards alternative operating–systems like Linux can be observed nowadays. This has only a limited impact on small and medium-sized businesses, but does affect large enterprises where centralized management clearly plays an important role.

With the advent of the Internet and increased utilization of evolving technologies, companies need ways to secure their data while it's on the move – typical clients not only include office workers, but also field personnel. Using encryption technologies is a logical step to secure sensitive information, but raises the question how to incorporate these techniques into an already existing infrastructure to provide optimal protection. This thesis aims to explore these already existing technologies for file–based as well as full–disk encryption on the Linux platform in terms of usability, performance and key handling, ultimately resulting in a proof–of–concept implementation that can be incorporated into available software products for different platforms.

Keywords: Linux File–Encryption Disk–Encryption

Zusammenfassung

Obwohl in der Vergangenheit die Mehrheit der IT–Landschaften durchwegs auf Betriebssystemen aus der Windows–Familie aufgebaut waren, kann gegenwärtig ein Trend in Richtung alternativer Betriebssysteme wie Apple's OS X oder Linux beobachtet werden. Im Gegensatz zu Klein– und Mittelbetrieben stellt dies vor allem Großunternehmen vor die Herausforderung, ein zentralisiertes Management der IT–Infrastruktur aufrecht erhalten zu können.

Daneben stehen Unternehmen vor der Aufgabe, Daten die sich durch die Erfindung und mittlerweile Omnipräsenz des Internets nicht mehr nur in Unternehmensnetzwerken befinden gegen unbefugten Zugriff zu schützen. Ein logischer Schritt um dies zu gewährleisten ist der Einsatz von Verschlüsselungsverfahren. Jedoch stellt sich hier wiederum die Frage, wie diese Verfahren in bereits existierenden Umgebungen eingepflegt werden können.

Ziel dieser Arbeit ist es deshalb, ausgehend von der Verschlüsselungssoftware *SafeGuard Enterprise* für Windows zu evaluieren, welche Möglichkeiten es gibt, Betriebssysteme wie Linux erfolgreich in die SafeGuard Umgebung einbinden zu können. Im Enterprise Bereich umfasst dies nicht nur Dateiverschlüesselung an sich, sondern auch Key Management, Datenintegrität oder Verhalten zur Laufzeit. Abschließend wird ein Machbarkeitsnachweis in Form eines Dateisystems implementiert, der vollständig kompatibel zu den SafeGuard Produkten ist.

Keywords: Linux Datei–Verschlüsselung Disk–Verschlüsselung

Contents

Contents vii

List of Figures ix

List of Tables xi

List of Code Snippets xiii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem statement . . . 2
- 1.3 Roadmap . . . 2
- 1.4 Linux . . . 4
- 1.5 SafeGuard Enterprise . . . 10

2 Evaluation 13
- 2.1 Cryptographic Algorithms . . . 13
- 2.2 Block ciphers . . . 14
- 2.3 Block Cipher – Modes of operation . . . 16
- 2.4 Password Expansion . . . 26
- 2.5 Secure storage . . . 26
- 2.6 Alternative Keystores . . . 33
- 2.7 The Virtual File System Layer . . . 39
- 2.8 Kernel–space Encryption . . . 44
- 2.9 User–space Encryption . . . 61
- 2.10 Interim results . . . 69

3 Development of shared component 71
- 3.1 Configuration file reader . . . 71
- 3.2 Event logging . . . 73
- 3.3 Generic object caches . . . 76

4 Development of a generic keystore interface 79

4.1 Keystore Interface 80
4.2 Architecture 84
4.3 Keystore–Interface message passing 86
4.4 External keyring sources 89

5 Development of user–space encryption using FUSE 91
5.1 Layout 91
5.2 File management sub–system 93
5.3 Password management 98
5.4 Memory management 102
5.5 Crypto core 106
5.6 Mountpoint bypassing 115

6 Runtime performance 125

7 Conclusion 135

Bibliography 137

A SafeGuard Enterprise file format 141

Glossary 143

List of Abbreviations 147

List of Figures

1.1 Chapter overview . . . 3
1.2 Selection of important Linux subsystems . . . 5
1.3 Linux Virtual Filesystem . . . 9
1.4 Overview of SafeGuard Enterprise . . . 11

2.1 CTR Mode encryption . . . 20
2.2 XTS Mode encryption . . . 21
2.3 XTS Mode Ciphertext stealing . . . 22
2.4 KDE Wallet Manager . . . 28
2.5 Interaction between processes and VFS objects . . . 41
2.6 Linux Layered file system access . . . 45
2.7 Linux Device Mapper . . . 51
2.8 eCryptfs architecture . . . 56
2.9 eCryptfs crypto overview . . . 56
2.10 Example: The container_of macro . . . 61
2.11 Filesystem in User–Space (FUSE) Architecture . . . 63
2.12 FUSE performance . . . 69

4.1 Keystore–interface architecture . . . 85

5.1 Slab allocator overview . . . 103
5.2 Slab allocator – slab layout . . . 104
5.3 IO request – read access . . . 112

6.1 Performance: raw . . . 127
6.2 Performance: FUSE passthrough . . . 129
6.3 Performance: FUSE memcpy–encryption . . . 130
6.4 Performance: FUSE AES256–CBC encryption . . . 131
6.5 Performance: Comparison . . . 133

List of Tables

2.1 AES modes of operation – speed comparison 25
2.2 KWallet: File format 29
2.3 KWallet: Resulting hash for password length ≤ 16 29
2.4 KWallet: Resulting hash for password length > 16 29
2.5 KWallet: Resulting hash for password length > 32 30
2.6 KWallet: Resulting hash for password length > 48 30
2.7 gnome–keyring entry format 31
2.8 gnome–keyring file format 32
2.9 LUKS header layout 53
2.10 LUKS key slot layout 54
2.11 Sample LUKS creation options 55
2.12 eCryptfs read method 58
2.13 eCryptfs write method 59
2.14 eCryptfs file header 61

4.1 Keyd / Keyd–runas interaction – pseudo code 88

5.1 notifyd recursive cache set up 118

6.1 Sample files generated for benchmarking 126

A.1 SGN key format 141
A.2 SGN padding header format 142
A.3 SGN file format 142

List of Code Snippets

2.1 IPC permissions . . . 34
2.2 Message queue object format . . . 34
2.3 Semaphore operation . . . 35
2.4 Sample Introspection file . . . 38
2.5 Kernel key type . . . 47
2.6 Kernel keyring users . . . 48
2.7 Data inheritance in C . . . 59
2.8 The container_of macro . . . 60
2.9 FUSE operations - sample . . . 64
2.10 RelFS main . . . 65
2.11 RelFS getattr function . . . 65
2.12 RelFS directory function . . . 66
3.1 libconfig grammar . . . 72
3.2 Logger backend functions . . . 74
3.3 Object cache – Base structure . . . 77
3.4 Callback operations for item caches . . . 77
4.1 Key daemon Lookup request . . . 82
4.2 struct ucred . . . 86
4.3 keyd helo message . . . 87
4.4 keyd lookup message . . . 87
4.5 Description of external key servers . . . 89
4.6 XML structure for key rings . . . 89
4.7 Keyd key structures . . . 90
5.1 File manager cache key . . . 94
5.2 File manager cache item . . . 94
5.3 ocbsecret data structure . . . 100
5.4 slab descriptor structure . . . 104
5.5 slab allocator slab initialization . . . 104
5.6 Context structure . . . 107
5.7 Crypto operations . . . 108
5.8 Per–file cryptographic data . . . 108
5.9 IO request . . . 111
5.10 notifyd info structure . . . 116

5.11 notifyd recursive path builder 118
5.12 inotify event structure 119
5.13 fanotify event structure 122
5.14 fanotify permission reply 123

CHAPTER 1

Introduction

1.1 Motivation

Early cryptography dates back thousands of years to ancient Egypt where it was used to confuse or obscure the meaning of inscriptions. Over time, cryptography has undergone fundamental changes in how it is applied – beginning from scytale transposition ciphers over trench codes in World War I and the world–famous Enigma machine in World War II to a great number of available ciphers today.

The invention of the computer, its rapid development and perfection have led to nearly all aspects of life are, at least at some extent, depending on the proper functionality of such systems. Coupled with interconnected networks, cryptography has regained attention as companies are facing the challenge of securing this ubiquitous experience for their users and the company's secrets on their own account. This typically implies management of user related data on the server side, but also covers all communication channels and ultimately the secure storage of data on the client side. In enterprise environments, these tasks are usually addressed by employing a wide range of products. Each tool typically serves a more or less complex purpose and only the combination thereof forms the IT infrastructure. In a heterogeneous environment however, maintaining interoperability becomes a high priority task. Commonly, if it comes to software support, Microsoft Windows and Apple OS X are the only fully supported platforms. If support for Linux is available at all, solutions only exist on a standalone basis or suffer from the lack of important features to use them in production environments. Besides operating systems used on workstations, mobile devices have become popular over the last couple of years. Although not covered by this paper, terms like Bring Your Own Device (*BYOD*) bring new security challenges for companies in the area of network security.

1.2 Problem statement

Windows and OS X operating systems do not suffer from aforementioned problems as they already have close ties to the vendor's own products or at least incorporate support for cryptographic toolkits in the kernel that allows easier integration of software on those platforms. This closed nature could be interpreted as an advantage when distributing software on those platforms, as one usually does not have to deal with compatibility issues, but simultaneously could also be considered as a major drawback as any public Application Programming Interface (API) could be changed, removed or otherwise restricted.

Linux on the other hand with its open design does not have such handicaps. Interfaces remain stable or at least could be back–ported in case support has been dropped. The issue here lies in the way how Linux works. The term *Linux* just refers to the kernel itself. It's up to the respective distribution to provide software packages for every day's needs.

The goal of this paper is to address the incompatibility issues. Based on a general overview of Linux distributions, the paper will evaluate their components in terms of usability for a cryptographic file system, ultimately resulting in a proof–of–concept implementation, fully compatible to the existing software suite available for Microsoft Windows and Apple OS X.

1.3 Roadmap

This section provides an brief insight of the chapters ahead. Figure 1.1 gives an initial overview of the areas that are covered by this paper.

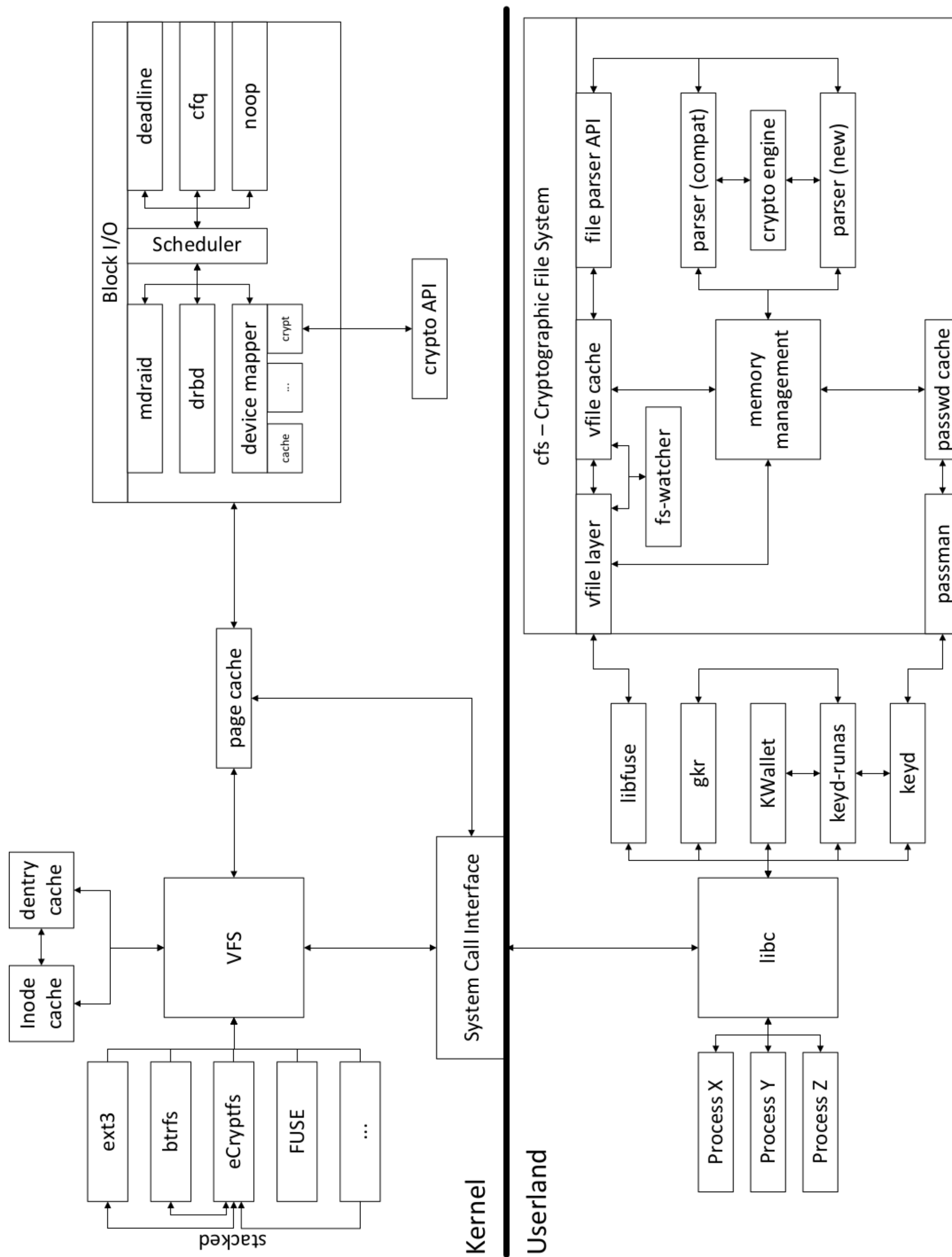
- Chapter 1 provides an initial overview of SafeGuard Enterprise as well as Linux. Starting from explaining common terms like Distributions, it also elaborates on important information like the various kernel subsystems this paper discusses and later uses in the prototype implementation.

- Chapter 2 addresses cryptographic basics like algorithms and modes of operation of block ciphers. Furthermore it provides a first overview of how to achieve persistent storage of sensitive information like encryption keys using native operating system functionality.

  Most importantly it also elaborates on the various layers the Linux kernel is composed of. Although the goal of this paper is to provide encryption in user space, chapter 2 serves as a guide where and how encryption is possible in kernel space.

- Chapter 3 is rather short an just describes shared components that have been implementation and used by all the tools resulting from this master's thesis.

Figure 1.1: Chapter overview



- Chapter 4 elaborates on the handling of key material and path based file encryption policies as sent by the SafeGuard Enterprise Server.
- Chapter 5 covers the main goal of this paper: implementing a file system in user space using FUSE. Since this is a rather complex task, the chapter describes each of the file system's components separately.
- Chapter 6 provides performance analysis of chapter 5' file system compared to native file operations.

1.4 Linux

The history of Linux began in 1991 when Finnish computer science student Linus Torvalds published his personal project – a new free operating system – on the Usenet group "comp.os.minix". What started as a small endeavor ended up being one of the world's largest open–source project with thousands of contributors around the world. Initially designed for the Intel 80386 processor, Linux is supported on all major platforms nowadays. The source code is freely available at the Linux Kernel Archives[1]. Since Linux is an open source operating system with a great many of developers, its development cycle is quite different from proprietary software. Contrary to that kind of software, the Linux kernel code can be obtained and changed by everybody due to its disclosed source code. Although the official source code tree is maintained by Linus Torvalds, companies as well as ordinary end users are able to submit patches to introduce new functionality or to fix existing bugs. This is an important factor for companies that rely on the stability of certain kernel interfaces and features, because needed functionality could either be back–ported once removed, or submitted to the Linux kernel code if initially missing – it simply breaks affiliation with specific vendors.

1.4.1 Architecture

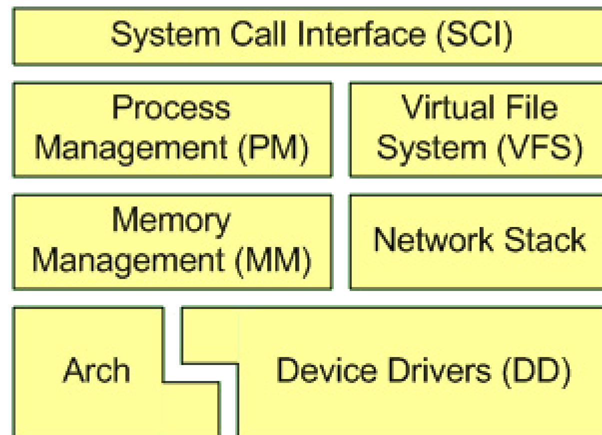
As for all computer systems, a basic set of programs is required in order to operate the system. The most important part of this set is the *Kernel*. It forms the basis for the operating system and is responsible for everything that happens on the system. It is loaded into memory during start–up and stays there until the machine is shut down. Due to the complexity, it is split into sub–systems, each serving a particular purpose. Figure 1.2 shows a selection of subsystems found in the Kernel which are also relevant for successive chapters in this paper:

System–Call Interface (SCI)

System calls are the main point of interaction between the kernel and user–space

[1] Located at https://www.kernel.org

Figure 1.2: Selection of important Linux subsystems [Jona]



applications and are used to instruct the kernel to do privileged work on behalf of a user process. Typical tasks are the creation of new processes or reading files from disk. However, users typically do not interact directly with the operating system. Instead, a common library is used for this purpose. This approach ensures portability across POSIX compatible operating systems and dispenses the user from setting up necessary CPU registers or system call numbers. While there are multiple libraries available, Linux uses the GNU C Library (glibc).

Process Management (PM)

After a system has started up, the kernel will create a single user–space process with Process Identifier (PID) 1 called *init*. This process persists throughout the system's uptime and acts as an ancestor to all other processes. The operating system has to keep track of newly created processes and their states to facilitate multi–tasking, as governed by the scheduler. When a new process is created using `fork()` – or to be more precise, the `clone()` system call, Linux copies its parent process, assigns a new PID and copies all data that persists across a `fork()` call and finally starts the process.

When the process has to be destroyed, either through a signal or a call to the `exit` function, the operating system cleans up existing memory mappings, per–process data, frees kernel memory used for housekeeping and notifies the scheduler to run the next runnable task. Process Management also covers interaction between processes. Linux offers various mechanisms to perform Interprocess Communication (IPC) as explained in later chapters.

Virtual Filesystem (VFS)

To provide support for various file systems wile encapsulating specific file system internals, Linux uses an abstraction layer known as Virtual Filesystem Switch (VFS). Since a vast part of this paper addresses file system specifics, further information about the VFS will be given in subsection 1.4.4 "File systems".

Memory Management (MM)

Linux uses a memory model called *Virtual Memory*. This approach is useful in multi–tasking environments as it not only provides fault isolation and information isolation, but also acts as an abstraction layer between processes and the physical memory and causes each task to see the *full* address space starting from address $0x0$ up to $2^{32} - 1$ or $2^{64} - 1$ respectively. When processes allocate memory, the kernel uses paging algorithms to map available physical pages into the processes' virtual address space. Modern CPUs provide hardware–support in the form of Memory Management Units (MMU). The kernel itself uses buddy allocation and slab allocation to satisfy these allocation requests.

Network

The operating system has to provide a subsystem that allows user–space applications to communicate with remote hosts. On a high–level, this support begins with supplying necessary system calls to access those functions whereas low–level implementations cover support for common network protocols. Furthermore, there must also be functionality in place that allows the delivery of packets to applications and vice versa.

Arch

Most of the kernel is platform–independent code, but especially during start–up and some low–level operations later on, the kernel has to execute code specific to the underlying hardware – the CPU in this case. The arch sub–system is a collection of supported CPUs and their respective low–level implementation for memory management, process creation or cryptographic operations.

Drivers

The largest portion of code in the kernel and responsible for providing hardware access to applications. Linux supports three types of device drivers:

- Character devices: allow direct access to the byte stream without buffering. Examples of character devices are the text console or serial ports.
- Block devices: allow access to data located at the underlying devices' block boundaries. Access to the data is cached in the kernel's page cache. Examples of block devices are hard disks. However, a hard disk is rarely accessed using the block device directly. Instead it is mounted and accessed through the resulting mount point.
- Network devices: allow access through the BSD socket interface and are generally not visible in the file system.

Each driver implements a common interface, allowing the user to perform typical operations like `read` or `write` on them.

As can be seen in figure 1.2, all functionality is directly included in the kernel itself. In addition, Linux provides support for Loadable Kernel Modules (LKM) that allow adding and removing functionality from the kernel at runtime. Related commands are `lsmod` to list loaded modules, `modinfo` to obtain additional information about a specific module, `modprobe` and `rmmod` to load and unload modules respectively.

1.4.2 Distributions

As mentioned before, the term *Linux* commonly refers to the core of the operating system. Since a computer system is of no use if only running a kernel, a fully–featured operating system also has to provide low–level applications / libraries that interact with the kernel itself thus making hardware access for user–space applications possible. On top of these libraries, users need applications that enable them to perform day–to–day tasks or management utilities for installing and removing applications.

Applications are typically called *packages*. A package can be installed, removed or updated by using a *Package Manager*. Unlike Windows, where software must be obtained from the respective vendor's web page, Linux collects available packages in one location called the *Package repository*. All the Package Manager needs to know is the address of such a repository and it then can provide necessary functionality for package management. These operations usually include

- installation of new packages
- removal of packages
- upgrade of out–dated or vulnerable packages
- search for packages in the repository
- add or remove a repository
- update package list
- perform system upgrades

This greatly simplifies software management on the system itself as the Package Manager also has access to all the packages' meta–information (e.g. state or version) and can therefore easily determine necessary steps (skip, upgrade) according to system policies. This is also advantageous in terms of system security. Microsoft for instance only provides automatic updates for Windows and other Microsoft products through *Windows Update*. Third party software requires the user to gather and install updates from various different sites, typically resulting in an aperiodic update cycle that puts the system at risk. With a Package Manager on the other hand, keeping a system up to date is quite simple and also includes security fixes, hence increasing system security by decreasing vulnerability life span.

The combination of the Linux kernel and a Package Manager along with a basic set of packages is called a Linux distribution. Since updates to the kernel are published according to a specific time window, a Linux distribution selects a specific kernel version, maybe applies distribution–specific patches to the kernel and provides support for that version over a longer period of time.

To date, Distrowatch[2] lists nearly three hundred available distributions. Due to the design of Linux, these distributions range from small–sized router firmwares to operating systems for super–computers. Not all of them are suitable for this research. Therefore, the main focus is limited to the following distributions typically encountered on a end–user workstation:

- Debian GNU/Linux
- Ubuntu
- Fedora
- Gentoo

1.4.3 Desktop environments

Since its invention, Linux had its strength in running on servers. As low resource consumption is a must–have property on those systems, Linux has never forced its users into a predefined system configuration to allow customization of the operating system to one's needs. A Command Line Interface (CLI) may be satisfactory on a server because tasks on such a system are usually confined to configuration tasks, but more importantly, the operator's experience is sufficient to perform them. Typical end–users however are greatly limited in their work if only a CLI is available.

In 1984, the X Window System (X) has been released by B. Scheifler at the MIT as a successor to the W Window System [Sch84]. Since then, X has undergone major rewriting and has become the de-facto standard for window management on UNIX platforms. In the late 1990s, when the number of Linux users increased, new desktop environments based on X were written, providing a rather uniform environment for the user. Common desktop environments available today include:

- K Desktop Environment
- GNOME Desktop Environment
- xfce

These environments usually consist of a window manager along with libraries for developers to provide a uniform look–and–feel for their applications. When installing

[2] http://www.distrowatch.com, accessed 01.12.2013

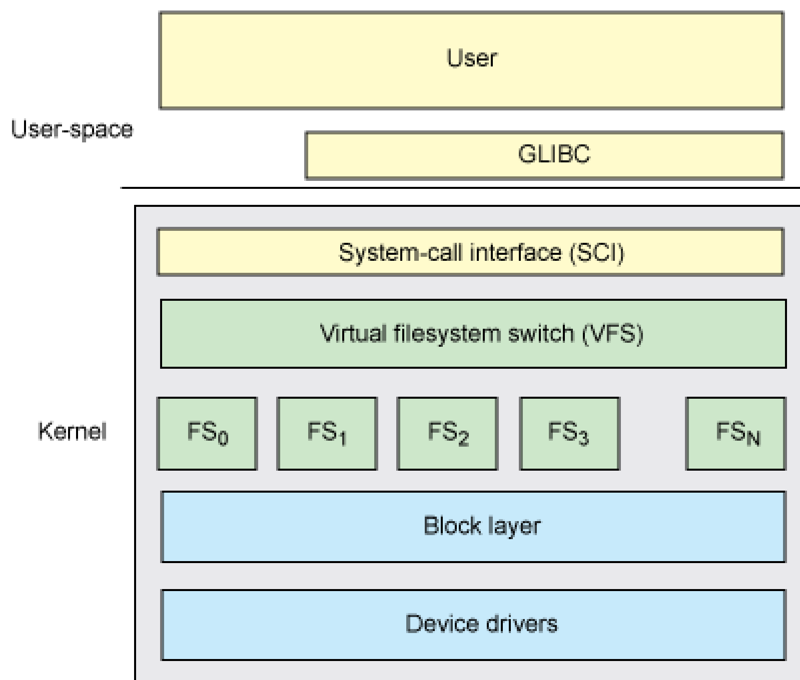
a desktop environment, the user does not only gain window support and decorations, but also environment–specific applications like file managers and image viewers.

A desktop environment can be obtained by downloading a distribution that already includes a specific environment or at runtime by using a Package Manager to install necessary packages. In addition, multiple environments can be installed simultaneously with the desired one selected at the login screen.

1.4.4 File systems

Every operating system is of very limited use if it does not support persistent storage of data. A file system is used to provide all that functionality. Its main task is to organize file access and establish a mapping from the logical view of files, like seen in a file manager to a specific, distinct layout on a block device. Figure 1.3 shows how Linux implements file system support. By adding another layer of abstraction, support for multiple file systems can be included in the kernel in a very efficient way. While each file system would increase the complexity of the kernel's API, the Virtual File System (VFS) layer eliminates this issue by exposing only a distinct set of functions to user–space. A file system like the Third Extended Filesystem (ext3) can then implement its own set of functions and only expose a reference to them to the kernel. The kernel keeps track of those structures and only has to pass user–space requests like `read` or `write` to the appropriate function implementation. The following simple command copies one file to another location to illustrate the power

Figure 1.3: Linux Virtual Filesystem [Jonb]



of the VFS layer:

$ cp /mnt/file /tmp/file

First of all, Linux uses the concept of *mountpoints* in contrast to drive letters encountered on the Windows platform. The Filesystem Hierarchy Standard (FHS) defines the initial folder layout, extended by user defined mountpoints. A mountpoint itself is represented by a folder that acts as a anchor for additional volumes like mapped network shares, removable devices and so on. Mount points can also refer to artificial data (e.g. the proc file system) which provide information about system resources. In addition, the VFS layer provides the necessary abstraction to attach any device with any supported file system to the system. Referring to the previous example, `/mnt` and `/tmp` do not necessarily refer to the same file system type. Instead, the kernel invokes the appropriate file system function to `read` the source file and uses the file system's `write` function to write to the destination. This makes the concrete implementation of both calls completely irrelevant to `cp`. The VFS layer also hides certain peculiarities as the next example shows:

$ ls /mnt/USBDRIVE

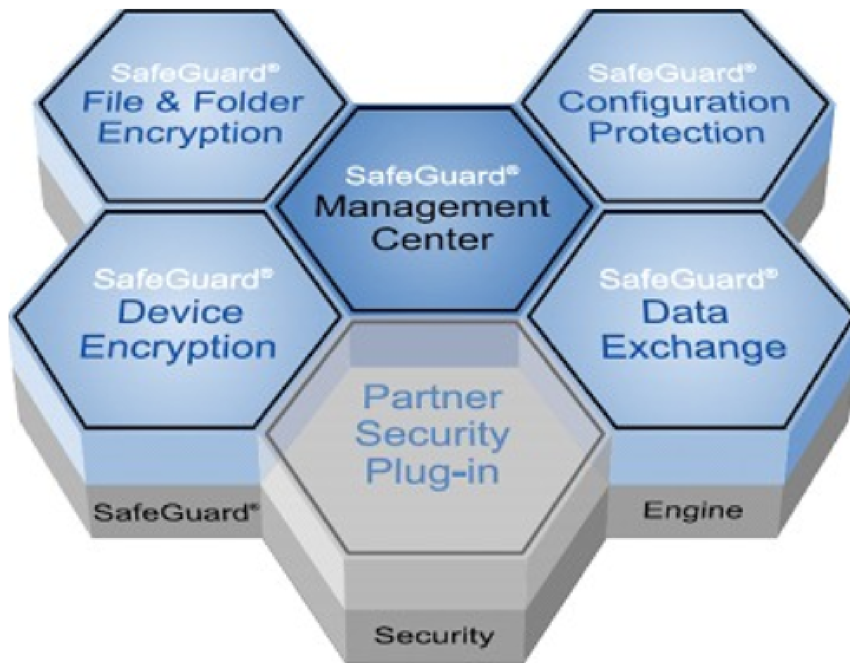
This command outputs a listing of folders and files in a directory. The *USBDRIVE* folder in this example refers to an externally connected hard drive. Assuming the file system is of type File Allocation Table (FAT) the kernel needs to know about FAT specifics so it can list the folder's content. In case of a FAT file system it becomes quite obvious why an abstraction layer is useful: the VFS layer uses a standardized model of how files and folders are represented. This allows operations like "list directory content" to stay simple as it merely calls the file system's implementation of "list directory content". FAT doesn't even use the inode model on disk, but this fact is encapsulated from the kernel. It simply calls the file system's implementation of "list directory content" which does all the heavy–lifting and passes back the contents in a format the VFS layer understands.

1.5 SafeGuard Enterprise

Sophos SafeGuard Enterprise is a well–established software suite for Windows server platforms offering encryption solutions for client systems running either on Microsoft Windows or Apple's OS X. It can either be used standalone or Active Directory integrated. Figure 1.4 shows the components of SafeGuard Enterprise. Features addressed by this paper are the Management Center in terms of basic key management and file and folder encryption in form of an evaluation of kernel–space and user–space tools that can be used for encryption.

The core component of SafeGuard Enterprise is the management center. Managed users are added either on a standalone basis or can be pulled from an Active Directory

Figure 1.4: Overview of SafeGuard Enterprise, Source: Sophos internal Confluence system



server which are then organized in containers – so-called Organizational Units. The administrator has the possibility to assign policies to those containers. A policy in turn is a set of commands and settings that the client enforced upon reception. While there are several hundred settings that can be changed in the management center, the device and file encryption policies are of most interest in this paper.

A device encryption policy interacts with the native disk encryption software on the client. On Windows platforms, it enables BitLocker Encryption whereas on Apple OS X it triggers disk encryption using FileVault2.

Policies that involve file encryption use a proprietary filter driver on Windows and a virtual file system on OS X platforms. Although a filter driver (kernel module) would be possible on Linux, this paper investigates possible user space implementations.

As one of the goals of this research is to provide compatibility to the encryption mechanisms, file format compliance will be of utmost importance. Tables A.1, A.2 and A.3 provide an overview of the file format as used by *SafeGuard Enterprise* and the resulting file system.

The research should result in a prototype of a file system implementation for Linux platforms with the possibility of reading and writing files created by SafeGuard components on aforementioned platforms. Furthermore, obeying to the Portable Operating System Interface (POSIX) is tried whenever possible to maintain portability not only among Linux distributions, but across all operating systems in the UNIX

family. For simplicity reasons, the SOAP–based server–client communication will be replaced by transferring simple XML documents containing necessary data. Additionally, disk encryption is also not part of this research as there is no native implementation of a on–the–fly full disk encryption under Linux.

CHAPTER 2

Evaluation

Chapter 1 has already outlined the foundation of this paper. Before an actual proof–of–concept implementation of a cryptographic file system can be constructed, peculiarities that stem from the nature of Linux need to elaborated in order to allow an efficient and extensible software design. This chapter will therefore first explain fundamental cryptographic terms before elaborating on suitable points of entry to the Linux operating system. The last part of this chapter outlines possible sources for key material and how they can be incorporated in the transparent file encryption process. The result of this chapter provides starting points for future kernel–space encryption solutions, but also serves as groundwork for the user–space file system implementation ("*cfs*") as specified in chapter 5, a prototype fully compatible to *SafeGuard Enterprise*.

2.1 Cryptographic Algorithms

In this paper, cryptographic algorithms are encountered in various encryption layers for file encryption, desktop–integrated keystore applications, and are part of transmission protocols between key servers and their clients. Their functional principles need to be understood in order to be able to draw conclusions about their benefits and drawbacks. It should be noted that this chapter only evaluates software and algorithms that will be encountered in already available software or in the final prototype as there are vast amounts of papers already available that describe cryptographic algorithms in depth.

Key–based ciphers can be categorized in two main categories: asymmetric and symmetric ciphers. In either case, they are used to transform plaintext into ciphertext by applying a *cryptographic algorithm*. For a given plaintext P, the cipher computes a corresponding ciphertext C by encrypting P.

$$C = E_{K1}(P) \tag{2.1}$$

The reverse operation, decryption, is performed on C and yields P

$$P = D_{K2}(C) \tag{2.2}$$

2.1.1 Asymmetric Algorithms

or public–key cryptography covers those algorithms that use different keys for encryption and decryption. The encryption key, or public key can be published and used by third parties to encrypt messages. The holder of the corresponding private (secret) key can then decrypt the message. Public and private key are mathematically related, but it is computationally infeasible to derive the secret key from the known public key due to the limitations of integer factorization and the calculation of discrete logarithms.

In general, public–key algorithms can be considered more complex in terms of required computational costs. Therefore, they are unsuitable for encrypting large amounts of messages. A common solution for this problem is to generate a random session key for Symmetric Algorithms and distribute it using public–key cryptography. Bulk encryption benefits from the throughput of symmetric ciphers while key distribution profits from the security of asymmetric algorithms in terms of key distribution over insecure channels. Such systems are known as *hybrid* cryptosystems.

Common examples of public–key algorithms include the Digital Signature Standard (DSS), Diffie–Hellman Key Exchange, or the RSA encryption algorithm (PKCS#1).

2.1.2 Symmetric Algorithms

Symmetric Algorithms are those types of ciphers where the encryption key can be calculated from the decryption key and vice versa, or where the same key is used in both cases. Equations 2.1 and 2.2 simplify to

$$D_K(E_K(P)) = M \tag{2.3}$$

Symmetric ciphers can be further classified in how they produce their output. *Stream ciphers* produce one bit of output for each bit of input, whereas *block ciphers* operate on blocks of fixed size. Throughout the software development as introduced in later chapters, focus will be on block ciphers as they are used in various parts like file encryption, key (un–) wrapping and integrity checking of ciphertext.

2.2 Block ciphers

A block cipher is an encryption function that operates on blocks of size specific to the algorithm. Modern block–ciphers like Rijndael use a block size of 128 bit, whereas older ciphers like Blowfish only use 64 bit. The block size also denotes the smallest unit of data the algorithm can process. When used, block ciphers take an input

block and transform it into ciphertext. In addition, the encryption process has to be reversible, so that the ciphertext can be transformed back to its plaintext form. The input and output of a block cipher always have the same length.

2.2.1 Blowfish

Blowfish is a Symmetric cipher designed by Bruce Schneier and described in [Sch96, p. 336] as a fast, compact, simple, variable secure, and optimized for applications where the key does not change often. The property "variable secure" refers to Blowfish's ability to work with keys varying in size from 32 bits up to 448 bits. The Blowfish algorithm consists of two parts:

key expansion:
a one–time process executed prior to the data encryption that computes 18 32–bit subkeys and four 32–bit S–Boxes with 256 entries each, giving a total of 4168 bytes of key material.

data encryption:
uses a 16 round Feistel network and pre–computed P and S–box values to generate the ciphertext. Blowfish uses a 64–bit block size per iteration.

The block size of 64–bit is the result of Blowfish's purpose to supersede the Data Encryption Standard (DES) which was already 17 years old at the time of the publication as the matching block size eased the upgrade from DES to Blowfish. To date, there are no known attacks on Blowfish, although its use is discouraged in favour of its successor Twofish – one of the five finalists of the Advanced Encryption Standard contest.

2.2.2 Rijndael

Rijndael, also known as Advanced Encryption Standard (AES) since its nomination by the National Institute of Standards and Technology (NIST) in 2001 is a block cipher that is specified for 128–, 192–, and 256–bit keys. The algorithm consists of an initial key schedule phase and the cipher algorithm itself. Furthermore, AES performs all its operations on arrays called *state*. A state consists of 4 rows. Each row houses Nb bytes, where Nb denotes the block size divided by 32. Hence, the state turns into a four by four array with each cell representing exactly one byte.

The key schedule is given by [oST01]. Its purpose is the calculation of sub–keys or round keys that are used during encryption and decryption. Depending on the used key size, the key schedule generates either 11 keys for 128 bit, 13 keys for 192 bit, and 15 keys for 256 bit keys. The number of subkeys therefore includes an additional key.

The cipher produces its output by subsequently applying *round functions* to the state array. Rijndael uses four different operations on the input to generate the output (from [oST01]):

1. Byte substitution using S–Boxes.
2. Shifting rows of the State array by different offsets.
3. Mixing the data within each column of the State array.
4. Adding a Round Key to the State.

These four operations make up a *round* in AES. As described in the specification, the last round is performed differently as it omits step 3. The result of the encryption or decryption operation is contained in the state array after the round functions have been performed.

2.3 Block Cipher – Modes of operation

2.3.1 Electronic codebook Mode (ECB)

Electronic codebook (ECB) mode is one of five recommended modes of operation for block ciphers as presented in [Dwo01]. ECB characterizes the most intuitive way of how to encrypt data. Plain text is split into blocks of suitable size for the cipher algorithm and then encrypted by applying the block cipher until all blocks are encrypted. For a given block of plaintext P_i, the corresponding cipher block C_i is calculated using:

$$C_i = E_K(P_i)$$

The resulting ciphertext is simply a concatenation of all previous cipher blocks. Due to the fact that each block does not have any relation to other blocks, encryption and decryption are parallelizable. Furthermore, it can be used when non–linear access to the encrypted data is necessary. Since blocks do not influence each other, encrypting the same data with the same key multiple times will always yield the same ciphertext. In the domain of data encryption, this could potentially be used to identify patterns as long as the same key is used.

Findings

Vast amount of research papers and literature exists on why ECB mode should not be used anymore in modern software. It is only mentioned as it unintentionally (hopefully) became the default mode of operation for KWallet.

2.3.2 Cipher–Block Chaining Mode (CBC)

To remedy the shortcomings of ECB mode, Cipher Block Chaining (CBC) mode adds a mechanism to chain cipher blocks together. Unlike ECB mode, where block P_i has no effect on subsequent blocks $P_{i+1..n}$, CBC mode uses the results of previous blocks to modify the current block before it is fed to the block cipher by using a XOR operation.

$$C_i = E_K(P_i \oplus C_{i-1})$$

It should be noted that for the case $i = 0$ no previous block $i - 1$ exists. Therefore, CBC mode requires the use of an Initialization Vector (IV) to perform encryption of the first plaintext block. The size of an IV is equal to the block size of the underlying block cipher. In addition, it is highly suggested that the IV is generated using a cryptographically secure pseudo-random number generator. The IV does not have to be kept secret, as it merely is just a placeholder for the non–existing block P_{-1}. Moreover, it also solves the problem of identical plaintexts resulting in equal ciphertexts. For decryption, the plaintext block P_i is computed as follows:

$$P_i = C_{i-1} \oplus D_K(C_i)$$

While ECB mode can be parallelized for encryption and decryption, the XOR operation in the encryption step renders parallelization for CBC encryption impossible due to the dependency of the previous block. Decryption on the other hand can be done in parallel as C_{i-1} and C_i are both available in step i.

This implies that for file encryption it is required that encrypted files are split into small blocks each having its own initialization vector, otherwise changes made to the beginning of a file (block C_i) would require re-encryption of the whole content as cipher block C_{i+1} depends on block C_i.

Findings

CBC is still widely used and supported by all cryptographic libraries available. A major drawback of CBC mode is that the size of the input data must be a multiple of the cipher's block size which for file encryption cannot be guaranteed all the time. Sophos SafeGuard Enterprise circumvents this problem by always using a buffer the size of the block size with a copy of the buffer after encryption being stored in the file header along with the exact size of the last block. A major drawback of that approach is that the file header needs to be refreshed during encryption (if appending to the file) and data needs to be read from the header during decryption of the last block. This might has a negative impact on performance as it breaks sequential file access.

Padding

A common issue when working with block ciphers is the fact that input data is mostly of arbitrary length. Block cipher modes like CBC and ECB however require that the

length of the input is a multiple of the block size of the algorithm. Sometimes this requirement cannot be fulfilled and the last block during encryption ends up only partially full. As a consequence, *padding* has to be appended during the encryption step to satisfy the length constraint

$$datalength \,\%\, blocksize = 0$$

A mandatory property of padding is its reversibility as it has to be removed during decryption. Zero padding for instance does not guarantee this behavior. The following example shall illustrate:

block size
: 16 bytes

message
: 24 `'0'` bytes

padding
: zero padding

padding length
: 8 bytes

While encryption would not cause any problems, *removing* the padding during decryption will run into problems as the plaintext contains `'0'` bytes at the ending. It is impossible to determine which of the zero bytes belong to the plaintext and which are part of the added padding.

An alternative to zero padding is PKCS #7 padding as described in [Hou09]. In this case, padding is added byte–wise. The value of each byte is defined by the total amount of padding that has to be added. When considering the same example from above, the padded last block would looks as follows when using PKCS #7 padding:

$$\texttt{MM MM MM MM MM MM MM MM}\underbrace{\texttt{08 08 08 08 08 08 08 08}}_{\text{padding}}$$

After decryption, the padding can be safely removed as the exact length is known. It should be noted however that PKCS #7 requires at least 1 byte of padding. Plaintext that ends on a block boundary therefore yields one additional block when encrypted.

Findings

Although padding would solve the problems of CBC mode as explained above, it can't be used during file encryption. Given a file with maybe millions of full blocks, encrypting each one would yield a block of padding which must be stored in the file. This only increases complexity of the read and write operations as offsets need to be recalculated to reflect the actual position within the file with respect to padding.

2.3.3 Counter Mode (CTR)

In contrast to ECB and CBC mode that generate ciphertext C by encrypting plaintext P with a secret key K, Counter Mode does not encrypt the plaintext directly. Instead it maintains an internal counter T which is then encrypted and XORed with the plaintext. [Dwo01] describes the process as follows:

$$O_j = ENC_K(T_j) \tag{2.4}$$

$$C_j = P_j \oplus O_j, \forall 1 \leq j < n \tag{2.5}$$

$$C_n = P_n \oplus MSB_u(O_n) \tag{2.6}$$

The fundamental principle of Counter Mode is given in [Smi11], which concisely describes it as a mode that "generates a key stream and applies XOR to implement a stream cipher." The key stream in turn is generated by encrypting successive values of a counter that will be increased after a block has been encrypted. With respect to Equation 2.4, CTR mode first calculates the jth output block by encrypting the jth counter value T. Depending on whether a full block (Equation 2.5) or a partial plaintext block is to be encrypted (Equation 2.6), either all of the output block bits are XORed with the plaintext or only up to the requested length. Because of Equation 2.4, it becomes evident why [Dwo01] demands that no counter value must be reused for a given key: in contrast to CBC mode where the cipher is applied to the plaintext, CTR mode only encrypts a counter value - the plaintext is never processed by the block cipher. Only the final XOR operation with the key stream yields the cipher text. Re–using the same counter value for a given (key, counter) pair would therefore produce the same keystream O.

$$C_1 = P_1 \oplus F(K, T)$$
$$C_2 = P_2 \oplus F(K, T)$$

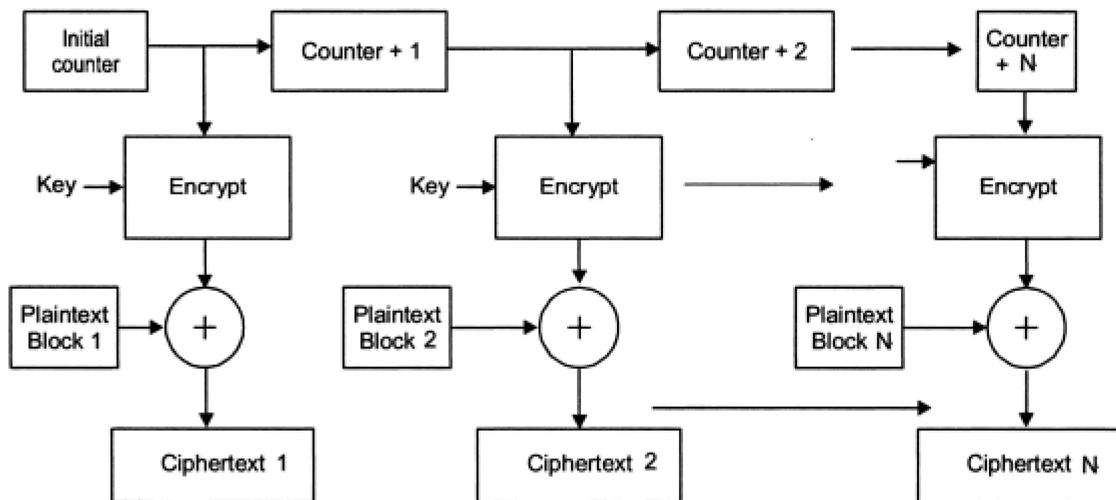
which is equal to

$$C_1 \oplus C_2 = P_1 \oplus P_2$$

Additional reuse of a (key, counter) pair adds more C and P block to the equation. Cryptanalysts can then employ various techniques to gain additional knowledge about the ciphertext and given enough input data, maybe even decrypt it.

In the simplest case the counter value is increased by one after each round. Although this might seem unorthodox and "insecure", it does not violate the requirements of [Dwo01]. The benefit of using Counter Mode is therefore that padding is no longer required as n bytes of input always produce n bytes of output. Figure 2.1 shows the basic algorithm. In addition, [Dwo01] specifies two possible ways of how the uniqueness of the counter is ensured. For a given underlying cipher like AES, the block size b is fixed at 16 byte or 128 bit respectively. Either the initial counter block starts with an arbitrary value and is incremented after every round, or the counter block is divided into two parts, a static nonce and a counter value. The specification

Figure 2.1: CTR Mode encryption [Pac08]



does not specify how the number of bits in b is split between nonce and counter. It should be noted however that the number of bits chosen for the counter limits the number of blocks that can be encrypted with the given key. This might become an issue when encrypting large amounts of data in a single file if the lower part of the counter is chosen too small.

Findings

When using CTR mode for file encryption, care must be taken to not violate the requirements of [Dwo01]. Given the fact that each file uses a random data encryption key with the IV being derived from the current block number within the file requires that a new IV is generated every time a block is overwritten. It is therefore recommended to store a 16 byte IV on a per–block basis and generate a new random IV upon encryption. As a result, offset and length calculations for read and write operations become more complex because the offset parameter of a read or write request has to be mapped to its real position within the file (that is, increased by the number of blocks before the requested offset multiplied by the block size).

2.3.4 XEX-based tweaked-codebook mode with ciphertext stealing (XTS)

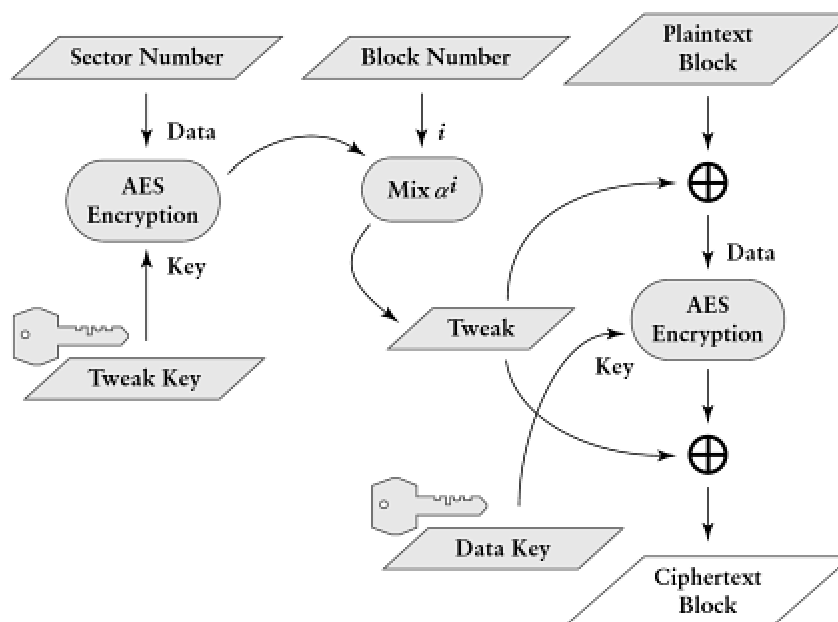
Presented in [IEE08] and approved by NIST in [Dwo10] XTS mode is described as a mode of operation used for "encryption of a data stream divided into consecutive equal-size data units, where the data stream refers to the information that has to be encrypted and stored on the storage device." XTS was specifically designed for transparent encryption, which reflects in its main properties as outlined by [IEE08]:

- It is *length preserving*. n bytes of plaintext generate n bytes of ciphertext, hence no padding is required.
- Encryption and decryption must be possible on individual data units. This property contradicts e.g. CBC mode where each cipherblock n depends on block $n-1$ during encryption.

The encryption process is shown in figure 2.2. As with Counter mode, encryption is not directly applied to the plaintext. Instead, the index of the data unit is encrypted using tweak key K_T. The result is *mixed* with the block number within the data unit to produce the tweak value. The outcome is then XORed with the plaintext before encrypting it with AES using the data key K_D and XORed again with the result of the AES output to produce the final ciphertext.

Since the file system prototype has built–in CBC mode support, the file block size as described before can be regarded as data unit. It should be noted however that due to the use of two encryption keys instead of splitting an existing 32 byte key, the effective key size for XTS if used in conjunction with AES–256 increases to 64 byte. As outlined in [LM08], the additional encryption key does not increase security, instead "the level of security would remain effectively the same." This fact implies that the file system prototype can only support XTS–128 without altering the *SafeGuard Enterprise* header format as the header only reserves 32 byte for the encryption key. Furthermore, XTS requires an input length of at least one full block of the underlying cipher – 16 bytes in case of AES. This is problematic for file encryption as very small files cannot be handled easily.

Figure 2.2: XTS Mode encryption [Pac08]

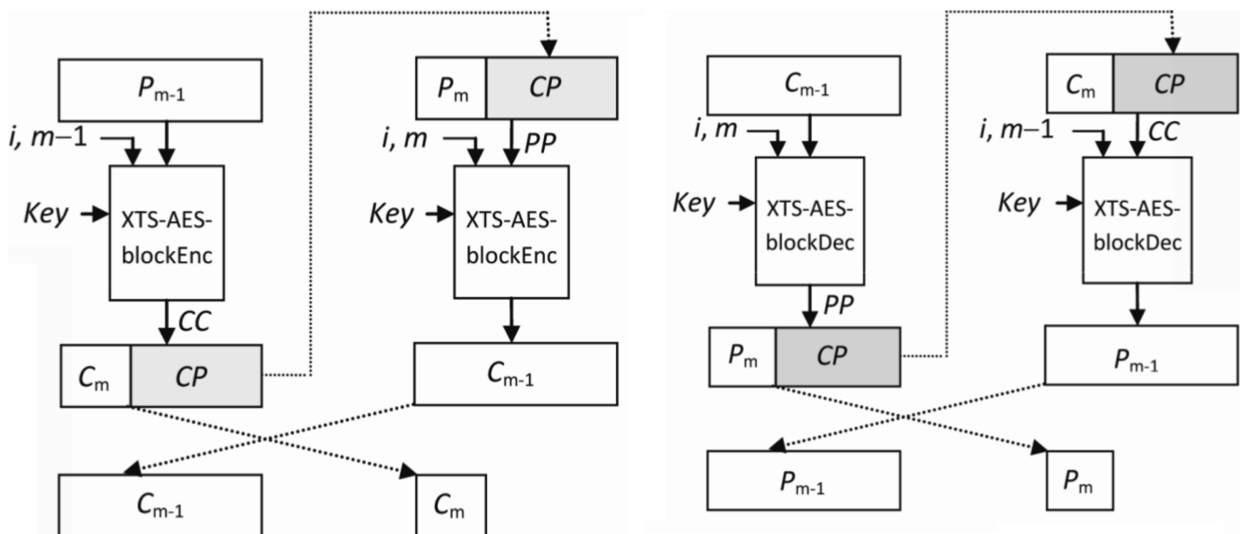


Tweakable block ciphers

Many modes of operation using block ciphers have certain requirements regarding their usage to prevent attacks of various kinds. Counter and Output Feedback more for example require the uniqueness of the initialization vector. Violating the respective restrictions will not only decrease security, it might even break security in general. Because key setup might be an expensive operation for a given block cipher, alterations to the input, output, or both would be a solution to mitigate attack vectors. [LRW02] however proposes a different approach by introducing *tweakable block ciphers* that can be regarded as extension to "conventional" block ciphers. Tweakable block ciphers add a special value that is cheap to calculate compared the fairly expensive key setup process. The block cipher still provides uncertainty through its symmetric key, but due to the tweak value also variability by embedding the tweak value in the process of transforming data by encryption or decryption.

Ciphertext stealing (CTS) applies to the length preserving property of XTS. CTS however is not limited to XTS. It can also be used in conjunction with ECB or CBC mode. CTS solves the problem of partial last blocks as addressed by Padding. When using CTS, the encryption step processes all full blocks except the last one. In case of XTS, the last full block is then encrypted together with the remaining partial block using two different tweak values ([IEE08, D.5]). Figure 2.3 shows this process. The encryption step is depicted on the left–hand side, decryption is shown on the right. During encryption, P_{m-1} denotes the last full block, P_m a partial block. The ciphertext CC is split in two parts: C_M and CP. CP is then appended to the last block P_m to expand the partial block size to the block size of the cipher (P_m therefore *stole* parts of

Figure 2.3: XTS Mode Ciphertext stealing (from [IEE08, 5.3.2,5.4.2]



the ciphertext) before applying the AES encryption. The output of the last encryption is stored at position $m-1$, whereas the partial result of CC makes up the last cipher (partial) cipher block. On decryption, the full block C_{m-1} is decrypted, which yields the plaintext block PP. Since the size of C_m is known, PP can be split accordingly to obtain the remaining input to the final decryption.

Findings

As will be shown later, XTS mode offers high throughput and therefore is a well–suited candidate for file encryption. Given the fact that it requires at least one full block during encryption special handling of operations that affect partial blocks shorter that 16 bytes are required. In such cases using XTS mode requires chaining with another mode that does not suffer from such restrictions, or the last block must be padded to at least 16 bytes and the exact file size stored in the header. This approach introduces the need to refresh the file header every time a block is added in the end (the most common write operation) to distinguish between data and padding. This of course breaks sequential file access and may has a negative impact on the overall write performance.

2.3.5 Galois/Counter Mode (GCM)

An aspect of file encryption left unregarded is the authenticity of ciphertext. Modes of operation for block ciphers discussed so far produce a certain amount of ciphertext that is written to disk and read and decrypted on demand. Neither during encryption nor decryption any authentication tags are generated or verified. With regards to the fundamental principle of information security, namely Confidentiality, Integrity, and Availability, the proposed solution only covers the former and the latter. To incorporate authenticated encryption, two approaches can be used:

1. Either employing a separate message authentication code (MAC), or

2. using a mode of operation that already incorporates authentication

Presented in [MV05], Galois Counter Mode (GCM) is an algorithm that provides "authenticated encryption and authenticated decryption." As the name suggest, GCM uses a combination of the already presented Counter Mode (CTR) for encryption and the *Galois* mode to provide authentication. Due to the use of CTR mode, the same rules and restrictions regarding key and IV generation and usage apply.

For authenticated encryption, GCM adds a forth parameter to the input which normally consists of a secret key, an initialization vector, and the plaintext. That additional parameter, commonly referred to as additional authenticated data (AAD) is then used to "protect information that needs to be authenticated, but which must be left unencrypted" ([MV05]). Consequently, the output of a GCM encryption yields

the ciphertext along with an authentication tag T. GCM decryption decrypts the ciphertext, but also maintains a MAC value T'. Given T and T', decryption is considered to be successful if the values match after the last block has been decrypted, hence the ciphertext has not been altered.

Given the nature of stream ciphers which allow random access, it is implied that only one such tag exists per file. In cases where a file is only partially decrypted, the tag cannot be verified without decrypting the whole file. As a consequence, a similar approach as for CBC mode can be used to make use of the authentication property. The file will be segmented in blocks, with each block consisting of the tag value and the encrypted data. The size of the data segment can be chosen at will, but should be within the recommendations of [Dwo07]. Given a random offset within the file, the file system can then calculate the corresponding segment location, decrypt the data block and verify the tag. Consequently, depending on how much data was requested, the verification of the tag imposes a certain amount of computational overhead as the whole data section has to be decrypted.

An additional use of GCM mode is the replacement of the MAC field in the file header. As can be seen in table A.1, each key slot contains an eight byte array that is calculated by using the first eight bytes when encrypting 64 zero bytes. It is used upon decryption to quickly determine if a given Key Encryption Key can decrypt the Data Encryption Key that is stored in encrypted form. By using GCM mode to encrypt and decrypt the Data Encryption Key, calculating the MAC value will become redundant as the GCM tag can be used to determine if decryption of an encrypted Data Encryption Key was successful. However, changing the modus operandi of the wrap and unwrap functions would break compatibility with existing files. Therefore, the proof of concept implementation will simply use a new header version to distinguish between files created with in traditional way and files using GCM mode.

Findings

In contrast to all other modes introduced in this section, GCM mode provides authentication of ciphertext during decryption. Although there are other modes supporting authenticated decryption like CCM or EAX, GCM mode is the most popular today. Two circumstances must be considered when using GCM: clearly it is used along with data to be authenticated, otherwise one could use CTR mode. The tag value obtained after encryption must be embedded in the encrypted file so it can be verified upon decryption. Due to the fact that the file header has a predefined size, tag values must be appended to the block within the data section of a file. Therefore, GCM suffers from the same limitations as CTR mode does in terms of file encryption: offset handling and random file access become more complex as the original data is interleaved with GCM tags. Accessing a file with an assumed block size of 512 byte at offset 1024 actually needs to access offset 1024 + 1 * GCM_PER_BLOCK_DATA. Additionally each 512 byte block only has one tag value, hence reading the first byte of a block requires

decryption of the full block to verify the tag value.

2.3.6 Comparison of cipher modes

The previous sections have shown that there are various modes available to use for file encryption. As the goal of this paper was to produce software that is compatible to the *SafeGuard Enterprise* file format, support for CBC and CTR mode is implicitly required. Due to the nature of CBC mode, the application already contains additional control logic to handle arbitrary I/O request, namely access to random offsets and handling of partial blocks. However, it is suggested that CBC mode should only be maintained for compatibility reasons. With Intel's introduction of the Advanced Encryption Standard New Instructions (AES–NI) for the x86 architecture and the `PCLMULQDQ` instruction for carry–less multiplication of two 64-bit operands, both described in [Gue12] and [GK12], tremendous performance improvements are noticeable when performing encryption with modes that make use of these new instructions. The sample source code in `extra/speed.c` [Lor15] can be used to obtain information about common supported block cipher modes. In its basic configuration, it encrypts 32 MB of data and calculates the average speed over 20 runs. Table 2.1 provides an overview of achievable encryption rates for a selection of CPUs. As can be seen, CBC mode cannot compete in terms of speed compared to other modes. Especially GCM mode benefits greatly if AES–NI is available due to the simplified multiplication in the Galois Field that is performed as part of the authentication step. GCM therefore provides an acceptable trade–off between speed and security if data integrity has to be maintained at the crypto layer instead of the software layer.

Although CTR mode offers the highest throughput among the tested modes, it does not protect the ciphertext from manipulation. Changing a byte at offset n in the ciphertext has a direct influence on byte n of the plaintext after decryption. Adding

Table 2.1: AES modes of operation – speed comparison

Processor			Mode [MB/s]			
Platform	Speed [GHz]	AES–NI*	CBC	CTR	GCM	XTS
i5-3320M	2.60	*Y*	369	1317	701	1290
		N	212	337	335	186
M 560	2.60	*Y*	388	832	495	830
		N	168	233	135	230
T9550	2.66	N/A	153	213	128	208

* disabled using `export OPENSSL_ia32cap=~0x200000200000000`

CBC–MAC to counter mode, hence turning it into CCM mode, performs exactly like CBC mode and is therefore omitted in table 2.1.

2.4 Password Expansion

Since the early days of authentication, passwords have played a major role in computing and especially cryptography. A password is usually a combination of characters, sometimes enforced through a policy, with a certain complexity obtained by mixing capital and lower-case letters, numeric, and special characters. To prevent brute–force and dictionary attacks, a policy might also enforce a certain password length. The problem in this case is the human brain. Complex passwords are easily forgotten, therefore users tend to use strings that are easy to remember but lack the required entropy in order to thwart password cracking. As a counter–measure, [Kal00] provides Password Based Key Derivation Functions (PBKDF) to derive a cryptographically strong key from a low–entropy password source such as user passwords. It does so by applying a pseudo–random function such as Secure Hash Algorithm (SHA-1) to the input along with a salt–value, repeated a certain number of times that governs the strength of the resulting key, but which is proportional to the computation time. Additional care must be taken on mobile phones as that number also has a direct influence on the battery charge level due to the high CPU load during the calculation. As a conclusion, the number of rounds should be the result of:

- a policy–defined minimum value
- frequency of required computation
- available hardware resources

On some platforms, usability is a concern. In such cases it would be possible to define the upper time bound that is acceptable for users and check if the resulting value provides the amount of security one desires.

[TBBC10] specifies the number of rounds (iteration count) as a value "selected as large as possible, as long as the time required to generate the key using the entered password is acceptable for the users".

2.5 Secure storage

As file encryption has to deal with cryptographic keys at some point, the following chapter will address problems that arise when implementing secure storage to maintain persistent storage of key material. First of all, it should be noted that *secure storage* is not limited to the process of writing data to a persistent storage device, but also extends to mechanisms for maintaining consistency and integrity. As mentioned

in chapter "Desktop environments", Linux is able to run various different desktop environments. Software on Windows or OS X platforms commonly uses the kernel API or a dedicated interface provided by the operating system to manage cryptographic keys. On Linux however, it is up to the user to install necessary applications and libraries. A typical installation of any desktop environment usually includes the required tools. Since K Desktop Environment and GNOME are the most common environments, the final implementation will only cover these types, but will offer an API that eases implementation of additional key storage providers. Furthermore, leveraging functionality provided natively by the desktop environment increases convenience for users as dialog and input boxes have their correct icons loaded and are also correctly themed. More importantly, the desktop environment already provides functionality to take care of certain operations like prompting the user to enter a password for a new keyring. This may seem trivial, but the problem will become more apparent throughout this chapter.

2.5.1 Terms

In this chapter, the term *keystore* refers to an implementation that offers functionality to persistently store data. A designated API offers ways of interacting with the library for the developer.

Each keystore consists of multiple *keyrings* or *keychains* which ultimately store the user's secrets. It is highly suggested that keyrings are protected using strong passwords. The API usually provides functions to lock and unlock keyrings as well as to create or delete them. To read and write data from or to a keyring, it has to be unlocked.

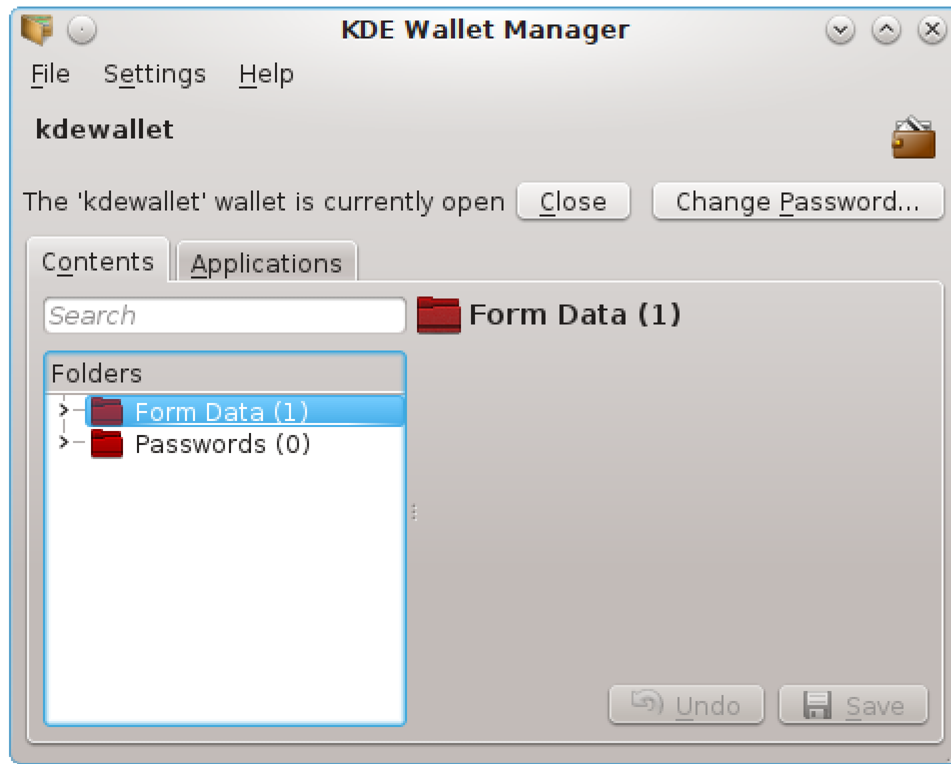
Additionally, when creating or opening keyrings the developer can pass the required password to the keystore or let the store itself take care of obtaining the correct password. Such an approach most likely uses D–Bus and therefore requires that the user's X session is set up properly. More details will be given in section "section 2.6 “Alternative Keystores”.

2.5.2 KWallet

KWallet is the default password manager for K Desktop Environment and is equivalent to the Keychain application on OS X and the Windows Vault on Windows platforms.

The KWallet sub–system manages a number of *wallets*. By default, only the initial wallet – *kdewallet* exists which gets unlocked during the users's login process and henceforth acts as the default destination for all newly added secrets. The KWallet main API offers the possibility to manage existing wallets (open, close, and change passwords) or to create new wallets. Figure 2.4 shows the *KDE Wallet Manager* main page.

Figure 2.4: KDE Wallet Manager



On–disk format

The information in the following section is taken from the package `kde-runtime` version `4.11.3-1`. References to source files in the following chapter are obtainable by downloading necessary libraries. On Debian, one would use

```
apt-get source kde-runtime
```

The KWallet file format is rather simple and implemented in `kwalletbackend.cc`. The backend service uses a `QDataStream` to manipulate data streams to and from the wallet. The file format is described in table 2.2. If the wallet backend needs to create new wallets, it uses `password2hash()` to perform the necessary password expansion. This function uses the user entered password and expands it according to the following rules:

1. Perform 2000 SHA-1 rounds into "block1"
2. If password size $>$ 16, perform another 2000 rounds into "block2"
3. If password size $>$ 32, perform another 2000 rounds into "block3"
4. If password size $>$ 48, perform another 2000 rounds into "block4"

Table 2.2: KWallet: File format

Offset	Size (b)	Description	Example
0	12	Magic number	`KWALLET\n\r\0\r\n` (static)
12	1	Wallet major version number	0 (static)
13	1	Wallet minor version number	1
14	1	Encryption algorithm identifier	0 (Blowfish CBC)
15	1	Hash algorithm identifier	0 (SHA–1)
16	4	Number of MD5 hashes	
20	n*20	MD5 hashes + NUL bytes	
20+n*20	*	Encrypted data	
*	20	Encrypted data SHA–1 checksum	

5. Depending on the password size, return hash according to table 2.3 – 2.6. The value denotes the block number from steps 1–4.

Passwords with $length \leq 16$ only generate a 20 byte key for the following Blowfish encryption with a total of 2000 rounds of non–standard PBKDF and it further is reasonable safe to assume that a user password as entered at the KWallet wallet creation wizard is shorter than 32 characters, so most KWallets are generated according to tables 2.3 or 2.4. Tables 2.5 and 2.6 are therefore only shown to illustrate the result and how the custom PBKDF distributes the generated blocks over the resulting 56 byte key. However, due to the low security standards offered by KWallet, it is recommended to always create wallets with a password as long as possible.

Table 2.3: KWallet: Resulting hash for password length ≤ 16

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1												

Table 2.4: KWallet: Resulting hash for password length > 16

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2								

Table 2.5: KWallet: Resulting hash for password length > 32

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3								

Table 2.6: KWallet: Resulting hash for password length > 48

1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2
2	2	2	2	2	2	2	2	2	2	2	2	3	3	3	3
3	3	3	3	3	3	3	3	3	3	4	4	4	4	4	4
4	4	4	4	4	4	4	4								

Security

As can be seen in table 2.3 the strength of the resulting wallet secret very much depends on the user entered password. In addition, the fact that the key generation scheme does not use any salt, makes it greatly vulnerable to pre–computed rainbow tables and brute–force attacks.

Even more worrisome code is located in the `Backend::sync()` function. The backend sets up the required initialization vector for the CBC encryption, but does completely fail to perform the encryption accordingly, thus turning CBC mode effectively into ECB mode as seen in the following function call:

```
rc = bf.encrypt(wholeFile.data(), wholeFile.size())
```

The `encrypt` function that should actually perform the CBC encryption first allocates a temporary, zeroed buffer of size equal to the input data size, XORs the buffer with the input data (which does not modify the data at all) and finally calls

```
rc = _cipher->encrypt(block, len)
```

with the full data block instead of splitting it into blocks and encrypting them in CBC mode.

Although not yet published in the official distribution packages, upcoming releases of KWallet will include support for GNU Privacy Guard (GPG) based encryption and decryption. To support GPG, cryptographic operations have been moved from `kwalletbackend.cpp` to `backendpersisthandler.cpp`. The newly added class `GpgPersistHandler` as child of `BackendPersistHandler` will then be used to delegate cryptographic operations to GPG and bypass the low–quality Blowfish code completely. The user has to enable GPG in the system preferences to make use of the new features.

Findings

It has been shown that the security of KWallet is somewhat unsettling. As mentioned before, ECB mode is unsuitable for secure data storage. Additionally due to the self–made password expansion it is suggested to collect all key material obtained from a SafeGuard Enterprise Server in a distinct keyring and use a strong random password to protect it. The password for the keyring can then be stored in the default login keyring which gets unlocked upon the user's login. It is believed that awaiting new versions with GPG support will not bring any added benefits as the typical end user in an enterprise environment does not have any GPG keys and has to use the default and broken implementation anyway.

2.5.3 gnome–keyring

Found on distributions running the GNOME desktop environment like Debian and its derivates (like xcfe or lxde) gnome–keyring is another keystore implementation. The application is split into two parts: the library `libgnome-keyring0` with the main API for the user to interact with and a daemon `gnome-keyring` that handles the requests from the library. The communication between the daemon and the library is based on D-Bus.

On–disk format

The information in the following section is taken from the package `gnome-keyring` version `3.4.1`. Compared to KWallet, the file format is more complex, as shown in table 2.8 but sets and preserves more keychain attributes.

During the user's login procedure, gnome–keyring starts its initial daemon that will wait for requests from `libgnome-keyring` and takes care of all necessary prompting for user–passwords.

Table 2.7: gnome–keyring entry format

Offset	Size (b)	Description	Example
0	4	ID	1
4	4	Type	1
8	4	Number of attributes	4
Array of *n* attributes			
0	*	Attribute name	"label"
*	4	Attribute type	0 (string)
*	*	Attribute value	"mykey"

Table 2.8: gnome–keyring file format

Offset	Size (b)	Description	Example
0	16	Magic number	`GnomeKeyring\n\r\0\n` (static)
16	1	Keyring major version number	0 (static)
17	1	Keyring minor version number	0 (static)
18	1	Encryption algorithm identifier	0 (AES)
19	1	Hash algorithm identifier	0 (MD5)
20	4	Keyring pretty name length = n	
24	n	Keyring pretty name	
24+n	8	Keyring creation time	1377793344
32+n	8	Keyring modification time	1377793354
40+n	4	Keyring flags	LOCK_ON_IDLE_FLAG
44+n	4	Keyring lock timeout	60
48+n	4	Keyring hash iteration count	3021
52+n	8	Keyring salt	
60+n	16	unused	
76+n	4	Number of entries = m	
80+n	*	Array of m entries	
*	4	Size of encrypted data	
*	*	Encrypted data	

Security

gnome–keyring uses a custom made PBKDF, implemented in `egg_symkey_generate_-simple()`. This function takes the iteration count and salt from the header and generates a key and IV suitable to use with AES-128. In case the secret collection is to be written to disk, no iteration count value exists yet, so `gkm_secret_binary_write()` will generate a random number in the range of [1000, 4096]. The encryption and decryption process uses CBC mode from *Libgcrypt* – a general purpose crypto library to encrypt the entries in the keyring. A MD5 hash value is included to verify the decryption process (located in the first 16 bytes of the "Encrypted data" section in table 2.8).

While gnome–keyring has to keep sensitive information in memory, it uses the `egg_secure_*()` functions to protect the memory from being swapped out to disk. The `egg-secure-memory.h` interface provides a custom memory management facility to perform secure handling of data. Additionally, free and malloc functions always return zeroed memory areas. Blocks of memory are organized in pools of twice the

size of PAGE_SIZE (usually 4 kB under Linux). Applications like gnome-keyring can then use the public API functions like egg_secure_alloc() or egg_secure_realloc() to obtain secure memory.

Findings

As is it the case with KWallet, gnome–keyring also uses a custom made key–derivation function to protect its keyrings. Using random salts and in the best case twice as many iterations during password derivation as KWallet does further increases the security of persistently stored key rings as it hampers brute–force attacks. Due to these facts it is rather fortunate that most of the Linux distributions include GNOME as their default desktop environment if they ship with a graphical user interface at all. Gnome in turns includes gnome–keyring. Therefore, no user interaction to install additional software is required.

2.6 Alternative Keystores – Linux IPC mechanisms

If different keystore applications are to be used or even self–implemented, a form of inter-process communication is needed in order to allow a graphical user interface to interact and exchange information with a persistent storage driver. The following section introduces a number of common UNIX inter-process facilities.

The * character in the following section should be substituted with one of three possible values: msg for message queues, sem for semaphores, and shm for shared memory segments.

***get()**
: creates a new resource of type *

***ctl()**
: perform control operations on the resource of type *. Operations include information retrieval about the resource, removal, or the setting of resource–specific options.

The resources created from one of the IPC instruments are commonly identified by a system–wide unique key, which can be of arbitrary value. To allow applications to work on the same resources, it is mandatory to use the same key for both applications. Therefore, the key value is derived from a combination of an arbitrary pathname and identifier through the ftok() function. According to the manual page, this function "convert[s] a pathname and a project identifier to a System V IPC key." Since the keys have to to be equal, the input to ftok() also has to be same in both cases. The pathname specified under ident must be existing and accessible as ftok() internally uses the stat() system call to derive the key. In addition, each IPC mechanisms has a set of permissions of type struct ipc_perm as shown in listing 2.1 attached to it.

Snippet 2.1: IPC permissions

```
1 struct ipc_perm {
2     uid_t          cuid;    /* creator user ID */
3     gid_t          cgid;    /* creator group ID */
4     uid_t          uid;     /* owner user ID */
5     gid_t          gid;     /* owner group ID */
6     unsigned short mode;    /* r/w permissions */
7 };
```

Using the creator and owner fields, access to the queue can be further restricted.

2.6.1 System V Message Queues

A message queue is an IPC mechanism provided by the kernel to exchange messages between processes. Typical operations performed on queue objects are `msgsnd()` to add an item to the end of the queue and `msgrcv()` to remove the head of the queue. Access to the queue is granted if the user performing the operation has write permissions for `msgsnd()` or read permissions for `msgrcv()`. A queue is identified by a positive *message queue identifier*. Application can provide this identifier to the send and receive operations to submit and retrieve objects of arbitrary type from the queue. The object is of arbitrary type, but should follow the base form shown in listing 2.2.

Snippet 2.2: Message queue object format

```
1 struct msgbuf {
2     long mtype;       /* message type, must be > 0 */
3     char mtext[1];    /* message data */
4 };
```

It is also possible to submit messages of length 0, hence submitting a structure that only consists of the `mtype` field. Both `msgsnd()` and `msgrcv()` are blocking calls as long as `IPC_NOWAIT` was not specified in the msgflg parameter. In terms of programming, the

```
int msgsnd  (int msqid, const void *msgp,
             size_t msgsz, int msgflg);
```

function is used to append a message to the queue. The `msgsz` parameter specifies the size of the data in `msgp`, *excluding* the size of the `mtype` field. If there is sufficient space available, the call succeeds immediately, otherwise the behavior depends on whether the `IPC_NOWAIT` flag was set or not. If set, the call fails with `EAGAIN`, otherwise it blocks until items have been removed and more space becomes available. On the receiving side, the function

```
int msgrcv  (int msqid, void *msgp, size_t
             msgsz, long msgtyp, int msgflg);
```

will remove the head of the queue and places the result in the buffer `msgp` of size `msgsz`. The `msgtyp` parameter can be used to specify which type of message should be removed.

2.6.2 System V Semaphore Sets

Semaphores are a vital part of multi–threaded applications and are one of the most fundamental principles when it comes to critical sections and resource utilization. Basically, a semaphore is a counter. Initially, the counter value is set to an arbitrary number. Every time a user requests access to a resource, the counter is decreased by one. When the user has finished his work, the counter is increased by one. As long as the counter is nonzero, the process continues working. When the counter reaches zero, which means there are no more resources available, the user has to wait until another user has finished his work and thus increased the counter. Therefore, semaphores are useful when there is more than one resource to manage. A *mutex* is a special semaphore with a maximum counter value of one. It is therefore useful when implementing critical regions[1]. Linux manages semaphores in terms of set. As before, the call `semget()` creates or retrieves such a set of n semaphores. To manipulate one or more items in a set, Linux provides two central functions:

```
int semctl  (int semid, int semnum,
             int cmd, ...);
```

is used to perform control operations on the set. Operations include retrieval, removal, information gathering about the semaphore counter values, and also the ability to set the counter values, as a newly created set of semaphores initially has undefined values. When the semaphore set is configured, one can use

```
int semop  (int semid, struct sembuf *sops,
            unsigned nsops);
```

to perform aforementioned operations on the set. An operation is characterized by a `struct sembuf` as described in listing 2.3.

Snippet 2.3: Semaphore operation

```
1 struct sembuf {
2     unsigned short sem_num;  /* semaphore number */
3     short          sem_op;   /* semaphore operation */
4     short          sem_flg;  /* operation flags */
5 };
```

The `sem_op` value determines the action that should be taken for this operation. If its value is

[1] A block of code that must not be executed concurrently

> 0 the value is added to the counter and the call returns immediately.

$= 0$ the caller is suspended until the semaphore counter reaches zero.

< 0 if the semaphore counter is greater than |sem_op|, add this value to the counter (effectively substracting it) and return immediately. If the counter is less than sem_op, wait until the counter increases.

The operations performed by `semop()` are atomically, which means that either each single operation succeeds or none of them is performed, the previous state is restored, and and error is returned. `nsops` determines, how many `struct sops` have to be performed. The smallest set obtainable by `semget()` is one.

2.6.3 System V Shared Memory Segments

Until now, processes used their own memory segments which remained isolated and inaccessible to other processes. When information exchange between processes is needed, IPC primitives like sockets or pipes have to be used to send and receive data. Another way of achieving the same goal is the usage of shared memory segments. A segment can be obtained by using `shmget()` which also takes a size argument. This call is similar to the `malloc()` system call. If the call succeeds, the memory segment has been created, but is still inaccessible until the application attaches it to its own address space using the `shmat` system call.

The return value of `shmat` can then be casted to the desired type and data written to to the memory segment. As shared memory is mostly used in some sort of producer–consumer environment, access to the memory has to be synchronized through primitives like semaphores to avoid data corruption.

When the process has finished its work, the memory segment has to be detached from the address space using the `shmdt` system call. It should be noted that the system call performs the detaching, but it still exists and the underlying memory in still use. Therefore, another call to `shmctl()` with the operation set to `IPC_RMID` is necessary to free the memory page. The actual destruction of the segment will be delayed until the attach/detach counter reaches zero, that is, when all applications have detached the segment from their respective address space. The advantages of using KWallet and gnome–keyring are clearly their native integration into the desktop environment they are written for and their out-of-the-box functionality.

2.6.4 D–Bus

Although there are other libraries available, *cfs* will only support KWallet and gnome–keyring. This section however should introduce the required interfaces if one wants to use custom keystore implementations. This is important as events that require user interaction become problematic to handle and a certain logic has to exist that is able

to attach to the current X session to prompt the user for input[2]. Due to the architecture of *cfs* as introduced in later chapters, launching applications from background applications with no additional user context brings the question of how to find and connect to the user's desktop session to display a graphical interface.

D–Bus or *dbus* provides an open–source inter-process communication layer for applications. It was originally designed and published by Red Hat Inc. and is now actively maintained by the freedesktop[3] community. D–Bus extends the primitive inter-process communication facilities that are typically available under Linux like socket, pipes, or shared memory. It is installed by default on virtually any workstation that runs a graphical user interface. The core architecture of dbus includes two messaging buses:

System bus
: A system–wide messaging bus that allows any application to export services that any other application can use. Only one instance of a system bus exists at runtime.

Session bus
: A per user / per session bus. For each X login, a new session bus is created for applications to communicate.

The dbus architecture offers more flexibility than Linux' built–in IPC mechanisms. [Pal05] and [PCL+03] describe the design of dbus as based on the following entities:

Message
: The most fundamental concept of dbus is a *message*. It is comprised of a header and a body with a maximum size of 128 megabyte. The header describes endianness, message type, flags, and body size, but also contains an array of $(key, value)$ pairs that characterize the recipient.

Object
: An abstract representation of an object in the given language (e.g. Qt object, GLib object). Instead of identifying an object through its address in memory, dbus uses object paths to refer to a specific instance. This makes it possible to work on object without mandatory knowledge of how the object is handled at the remote side.

Object path
: An identifier for an object instance. A format path is similar to a file system path and has the format `/org/myapp/settings/windowstyle`. The message header does include the recipient's object path in the header fields too. For separation purposes, one can use a namespacing schema for applications – e.g. object paths

[2] for example creating or unlocking a wallet
[3] http://www.freedesktop.org

starting with `/org/kde/` are used to identify applications part of the K Desktop Environment project.

Methods & signals
An object would be worthless if one cannot interact with it. Therefore, dbus provides the concept of methods. Methods are identified by a name (e.g. `Set`) with optional parameters and may return a value. Signals on the other hand are automatically initiated by the object on a certain condition (e.g. `onChange`) and broadcasted on the bus. Interested parties may catch those events.

Interface
A named set of functions and methods, identified by a name with format `org.myapp.setting`.

Service
Registered by an application during start–up. A service is identified on the bus by a unique name, e.g. `org.myapp.setting`. A powerful feature of dbus is the support of service activation: if an application executes a call on a non–existing service, dbus has the ability to map the service to an application and start it.

To get an overview of what an ordinary Linux installation includes, one can use `qdbus [--system]` and `dbus-monitor` to inspect the buses. The system–bus typically lists services like Avahi[4] or UPower[5]. The session–bus on the other hand provides notification daemons (`org.freedesktop.Notifications`) or access to the user's keyrings through `org.gnome.keyring` or `org.kde.kwalletd`. The actual services provided depend on what packages have been installed on the system.

For languages with static binding like C however, a dbus object needs to be declared at compile-time as the compiler needs to "see" its definition. To obtain necessary definitions, `libdbus-glib-1-dev` provides `dbus-binding-tool`. It uses XML–style to describe objects including their methods, signals, and interfaces. Listing 2.4 shows an example that can be used as input to `dbus-binding-tool` to generate a C header file. A detailed description of the Introspection Data Format is given by [PCL+03]. The method `GetInt32` as specified in line 5 for example expects a matching C function `gint32 setting_get_int32(Setting *s)`.

Snippet 2.4: Sample Introspection file

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <node name="/org/myapp/Setting">
3 <interface name="org.myapp.Setting">
4   <annotation name="org.freedesktop.DBus.GLib.CSymbol" value="setting" />
5     <method name="GetInt32">
```

[4] D–Bus capable implementation of DNS service discovery and multicast. http://avahi.org

[5] Successor to the Hardware Abstraction Layer (HAL). Exposes an interface to power sources currently present on the bus. http://upower.freedesktop.org

```
6        <arg type="u" name="val" direction="out">
7          <annotation name="org.freedesktop.DBus.GLib.ReturnVal" value=""/>
8        </arg>
9      </method>
10     <method name="pGetInt32">
11       <arg type="u" direction="out" />
12     </method>
13     <method name="SetInt32">
14       <arg type="u" name="val" />
15       <arg type="u" direction="out" >
16         <annotation name="org.freedesktop.DBus.GLib.ReturnVal" value="err"/>
17       </arg>
18     </method>
19 </interface>
20 </node>
```

An interface to custom keystore implementation would then consist of a prompter service to prompt the user for input as well as accessor functions to the keystore's data like passwords or certificates.

Findings

It is highly recommended to use native keystores like KWallet or gnome–keyring to permanently store data in keyrings. If it is unavoidable to have custom protocols and interfaces in place, dbus seems to be the most widely pre–installed Inter–Process Communication and Remote Procedure Call facility among the most common Linux distributions. However, various implementation attempts of simple interfaces have shown that due to the complexity of the standard and peculiarities of software implementations (libdbus or glib's GIO GDBus) it is advisable to use one of the aforementioned facilities.

2.7 The Virtual File System Layer

Linux supports a variety of file systems. Due to the fact that each of them requires a different implementation and operates on different (internal) data, directly including them in the kernel would bring major drawbacks:

- A wealth of different data structures scattered across the whole I/O subsystem.
- Important data structures like a `struct inode` would need a union type that includes a member for each supported file system to allow each of them to store auxiliary data. The size of the union would be governed by the file system with the largest amount of auxiliary data. This approach wastes a lot of memory, especially when considering the number of inode structures loaded in memory at a given time.

- Re–implementation of common features for each file system.

As a consequence, the Virtual File System (VFS) layer was introduced to provide an abstraction layer to file systems. The result is a common interface for all file related functions with the VFS merely acting as a dispatcher that forwards system calls to specific file system implementations. The VFS layer is build around four major data structures:

`struct superblock`
: Refers to a mounted file system. For disk–backed volumes, the structure contains volume–specific information. On NTFS partitions, it is referred to as Master File Table (MFT).

`struct inode`
: Carries general information about a file system object like a file or a folder. Data associated with this structure includes ownership, size, permissions, various access timestamps and an inode number that is used to uniquely identify the file object within the file system.

`struct dentry`
: Links inode objects to path names. dentries do not exist on disks, but are kept in memory to speed up path lookups. Each dentry has a parent dentry, a name, and an inode associated with it.

`struct file`
: Denotes the link between a process and an inode. Maintains information that are specific to the process, not the file itself like access mode or current offset.

The result of the introduced abstraction layer is denoted as the *common file model* of Linux and the interaction between the components is depicted in figure 2.5. File system implementations like ext3 or FAT are required to follow this model. When considering the FAT file system, the power of VFS becomes more evident as these types of file systems do not use the concept of inodes. Instead, they must be constructed on demand. The VFS layer however does not need to take this implementation detail into account. It simply forwards a `read()` call to FAT which will then perform all necessary operations to get the job done. An example shall illustrate:

Userspace
: `read(int fd, ...)`

System call
: `read(int fd, ...)` in `read_write.c`

VFS call
: `vfs_read(struct file *file, ...)` in `read_write.c`

VFS dispatch
`do_sync_read(struct file *file, ...)` in `read_write.c`

VFS dispatch
`file->f_fops->read(file, ...)` in `read_write.c`

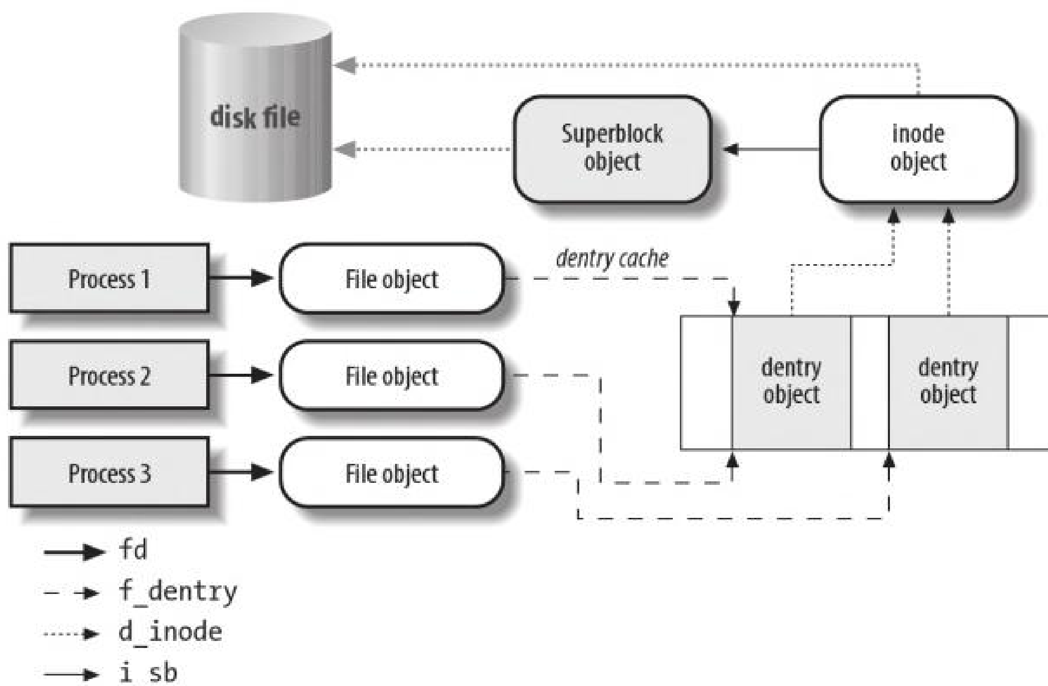
FAT call
`read()` function specified in `struct file_operations` in `fat/file.c`

In addition, the common file model does not only apply to files and directories. If a socket is to be created using the `socket()` system call, an (anonymous) inode is created. Instead of assigning file system operations as seen in the example above, the kernel assigns `socket_file_ops` to the inode's `f_ops` field. This makes it possible to use the `read` and `write` system calls on socket just like on ordinary files as they eventually translate to the appropriate functions suitable for inode types. Therefore, the VFS also brings convenience as it unifies the exposed I/O API to userspace.

2.7.1 dentry cache

Because I/O operations are performed on a frequent basis and require vast amounts of directory traversals and name lookups, the kernel tries to perform these operations as quickly as possible. Functions that operate on pathnames first need to translate them into dentry objects. That is, for a given path like `/bin/echo`, a dentry object for

Figure 2.5: Interaction between processes and VFS objects [BC05]



the root `/`, a dentry for the subdirectory `bin` and lastly for the `echo` entry need to be generated. It is therefore reasonable to keep dentry objects in memory while they are in use. Additionally, it makes sense to keep them even longer in memory as users tend to repeatedly access files. In such cases the corresponding dentry objects would be re–constructed multiple times which can be easily avoided by keeping them in a cache. As described by [BC05], the kernel's dentry cache consists of two kinds of data structures:

- A set of dentry objects, separated by their state
- A hash table to maintain a mapping from paths to dentries

In addition, the cache also maintains a Least Recently Used (LRU) list that contains dentry objects that are not in use. If the system is under pressure and running low on memory, the kernel frees those dentry objects.

2.7.2 File systems

The VFS layer provides all necessary abstraction to hide specific file system implementations from callers. However, as each request eventually passes through a distinct file system, the VFS needs references to the functionality a specific file system provides. To allow file systems to announce their presence, the VFS offers two functions:

`register_filesystem`
: The kernel maintains a list of known file systems. This function adds a given file system described by a single parameter of type `struct file_system_type` to the list.

`unregister_filesystem`
: Removes a given file system from the list.

During registration, the file system provides a structure containing its name along with pointers to functions that can read and discard a superblock. Once the file system has been registered, it can be used when invoking the `mount` system call. During system startup, the kernel will automatically mount the root directory (`/`) that contains applications and initialization scripts to ensure proper functioning. At runtime, either as a result of scripts or directly performed by users, additional devices can be mounted and linked to directories below the root directory. The destination directory of a mount operation is known as mountpoint. As a consequence, the content of the destination directory is hidden as long as a device is mounted, including its subdirectories. During application execution, the root directory may differ from the *real* root directory of a system as applications can make use of the `chroot` system call to define

what they consider as their root directory. Changing the root directory of a process does not affect other processes or the system itself.

A file system by itself does not trigger any events. Instead, its operations are invoked as a result of system calls concerning the VFS layer. As the following sections contain specific information about file system internals that typically refer to the file system in a mounted state, it shall now be explained how this state is achieved. The life of a mountpoint starts by executing the system call `mount()`. This call takes a source device, a destination mountpoint and a set of mount flags and options and passes them on to the kernel. The `do_mount` function in `namespace.c` performs sanity checks on the parameter, but also checks if the user is allowed to mount a file system. Before the file system is actually mounted, `security_sb_mount` will check that the user has permission to mount a device. This call will translate to either a Tomoyo[6] or SELinux[7] function call. Eventually, `do_new_mount` is invoked that performs a series of operations. First, `vfs_kern_mount()` is called which executes the `mount()` function each file system has to expose. The returned dentry object refers to the new root directory. This function also takes care of reading superblock objects from disk (through `mount_bdev`). After the superblock has been set up, `do_add_mount` adds the mount point to the current process' namespace which completes the mount operation. After that, the mounted file system is ready to serve VFS requests issued from well–known system calls like `open`, `read`, `write`, `close` and so forth.

Findings

Although the goal of this research is a file system in userspace, it is highly recommended that the prototype implementation *cfs* is moved to kernel space when using it in production. As has been shown, the VFS layer is very powerful and further already contains other stackable file systems that sit between the VFS and other mounted file systems which can and should be used as a starting point. Additionally, from a file system's view, the VFS layer has limited complexity due to its location on top of facilities like the block layer or the device mapper compared to other kernel subsystems. Because the VFS layer already provides multiple default implementations of read of write, new file systems can focus on what is known as `address space operations` that handle pages instead of file chunks. Cryptographic operations should become simpler due to aligned access.

[6]a Linux kernel security module by the NTT Data Corporation which implements Mandatory access control (MAC).

[7]man 8 selinux descibes SELinux as "an implementation of a flexible mandatory access control architecture in the Linux operating system."

2.8 Kernel–space Encryption

The following chapter will outline facilities in the Linux kernel that are useful for developing a cryptographic file system. Starting with an overview of generic APIs that give access to often used functionality, the main part of this chapter will address possible starting points when deploying kernel space file systems.

As the VFS layer already provides a sophisticated structure for various file systems, it is possible to implement encryption directly in kernel space. Figure 2.6 provides a basic overview of the VFS layer along with components participating in I/O handling. Due to the fact that the kernel has access to each device on the machine, encryption can be efficiently performed not only on a per–files basis, but also on block devices. This implies that for file encryption, a file system needs to be implemented. It has to be compiled either as a module or directly included in the kernel, where the former is preferred as it does not require any kernel source code modifications. At runtime, users are then able to make use of the file system by mounting it to a directory. With regards to the VFS layer, major parts of the implementation however need to address the issue of interacting with the underlying block device. Consequently, file encryption would only take place in a rather small part of the file system. The rest would cover inode and superblock management and interacting with the VFS layer. As a result, cryptographic file systems can be implemented on top of already existing ones, with the advantage being that they only have to process data in term of cryptographic operations and can safely ignore how the underlying file system handles passed data.

2.8.1 Crypto API

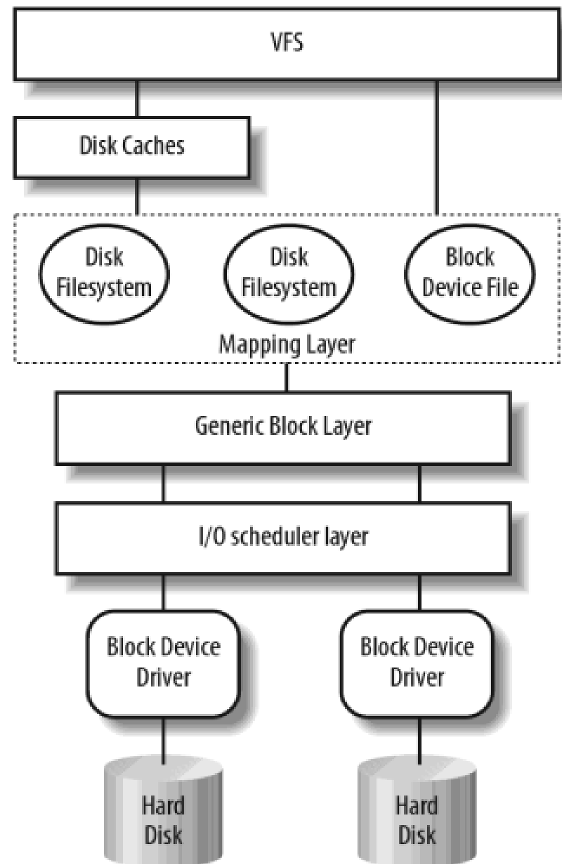
During the development of Linux version 2.6, a new API has been added to the kernel to address cryptographic tasks. Mainly developed by Herbert Xu, it was originally used to provide cryptographic support for IPSec. Due to its generic design however, systems like the WiFi stack or file systems also started to utilize its functionality.

On the highest implementation level, the API supports three types of algorithms: ciphers, digest and compressions. Out–of–the box supported ciphers are Rijndael and RC4; supported digests include MD5, SHA–1, SHA–224, and SHA–256; and supported compressions are LZS and Deflate. Block cipher modes of operations as presented in section 2.3 are created automatically for supported algorithms.

As for file systems, algorithms that should be accessible through the Crypto API need to be registered as well. A module that implements a certain algorithm can use `crypto_register_alg` at load time to announce its presence. If it has to be removed, `crypto_unregister_alg` will perform the reverse operation. At runtime, the kernel exposes a list of available algorithms under `/proc/crypto`[8].

[8] One can read that list using `cat /proc/crypto`

Figure 2.6: Linux Layered file system access [BC05]



The Crypto API operates on units of pages and performs all of its work in so–called scatter–gather operations, which denotes a process where data from multiple sources is processed by a single function call. Applications that perform read and write tasks typically run into similar problems as one has to issue multiple calls to read data into different buffers. This approach however adds additional overhead for each operation as each read or write call invokes a system call which entails a switch to kernel space, additional data copying operations and so on. An example is given below:

```
1  struct header {
2      uint16_t h_version;
3      uint32_t h_length;
4      uint32_t h_datalen;
5  } hdr;
6
7  uint8_t *data;
8
9  /* set up header and data */
10
11 write(fd, &hdr, sizeof(hdr));       /* write the header */
```

```
12 write(fd, data, hdr.h_datalen);     /* write the content */
```

While this example contains valid code, the two `write` syscalls add additional overhead at runtime. In cases where `fd` is connected to a socket, additional packets may be sent across the network even though they would fit in one. Sending data of such a type can be tedious as it results in poor performance and parsing complexity as it requires intermediary memory and the use of `memcpy`. Although the packets arrive in the order they were sent, additional packets may have been interleaved by the kernel which requires additional logic on the server–side.

To remedy additional costs and complexity, one can use vectored read and writes as illustrated in the following code snippet:

```
1  struct header {
2      uint16_t h_version;
3      uint32_t h_length;
4      uint32_t h_datalen;
5  } hdr;
6
7  uint8_t *data;
8
9  /* set up header and data */
10
11 struct iovec to_send[] = {
12     { .iov_base = &hdr,  .iov_len = sizeof(hdr)   },
13     { .iov_base = data,  .iov_len = hdr.h_datalen }
14 };
15 writev(fd, to_send, sizeof(to_send) / sizeof(to_send[0]));    /* send both */
```

which only requires one system call and further performs the operation in an atomic manner.

With respect to the Crypto API, [Mor03] emphasizes once again that the "primary purpose of scatter-gather in the kernel is to avoid unnecessary copying of data." Data in kernel space could stem from different sources and need to be processed fast and efficiently. For instance incoming traffic from network cards at 10 GBit/s that needs to be decrypted should not be copied to different locations until decryption is performed. In such cases the Crypto API needs to process data as fast as possible.

When it comes to encrypting data from user space, pointers to data are passed to kernel space. Before encryption can be performed, the address needs to be translated to a page address. The kernel provides `virt_to_page` and `offset_in_page` that translate a buffer of a given length into a scatterlist. If translated, one can use the main API which consists of the following functions to lookup and instantiate crypto algorithms:

`crypto_find_alg`
Tries to find a given algorithm

crypto_alloc_base
: Finds and allocates an algorithm. The find routine first tries to locate an already loaded algorithm. If unsuccessful, it loads the module containing the algorithm. Otherwise the crypto manager is invoked to construct one on the fly.

crypto_alloc_cipher
: Instantiates a cipher.

crypto_alloc_hash
: Instantiates a hash algorithm.

crypto_alloc_comp
: Instantiates a compression algorithm

Although the naming might be confusing, this API also allows access to hash and compression algorithms like *LZO* (Lempel–Ziv–Oberhumer compression).

2.8.2 Key Management

With the crypto API in place, subsystems have a single point to contact if they need to perform cryptographic tasks. Naturally, if it comes to encrypting and decrypting data, key material has to be supplied to conduct the operation. The key management facility however is not limited to cryptographic keys. Due to its generic nature it permits the storage and lookup of any token that follows a certain structure.

In kernel space, a key is always instantiated using a key type. As for each service provider, specific register and unregister functions exist. For key types, one can use `register_key_type` and `unregister_key_type`. However, a type has to be created in kernel space. It is not possible for user space applications to register their own key types. Registered key types carry a set of operations that take care of key specifics like instantiation, read, update or destruction. A key type is represented by a structure `key_type` as defined in `key-type.h` and partially shown in listing 2.5.

Snippet 2.5: Kernel key type

```
1 struct key_type {
2     const char *name;
3     size_t def_datalen;
4     int (*instantiate)(struct key *key, ...);     /* Create a new key */
5     int (*update)(struct key *key, ...);          /* Modify a key */
6     long (*read)(const struct key *key, ...);     /* Read key data */
7     [...]
8 };
```

As can be seen, above-mentioned functions operate on `struct key` as defined in `key.h`. This structure represents an arbitrary token with the following properties:

1. a unique serial number

2. an owner
3. a set of permissions
4. an expiration date
5. a type and description
6. an arbitrary payload

In addition, each user has a structure `key_user` assigned that keeps track of allocated resources as shown in listing 2.6.

Snippet 2.6: Kernel keyring users

```
1 struct key_user {
2     atomic_t       usage;      /* for accessing qnkeys & qnbytes */
3     atomic_t       nkeys;      /* number of keys */
4     atomic_t       nikeys;     /* number of instantiated keys */
5     kuid_t         uid;
6     int            qnkeys;     /* number of keys allocated to this user */
7     int            qnbytes;    /* number of bytes allocated to this user */
8     [...]
9 };
```

A set of keys is commonly referred to as *keyring*. By default, various keyrings exist with the most important ones being:

- thread-specific: each thread has its own keyring. During search operations, thread–specific keyrings are searched first.
- process-specific: each thread group (process) has its own keyring as well. It is searched if a secret was not found in a thread–specific keyring during search operations. *Note:* process-specific keyrings are replaced during `fork` and `exec`.
- session-specific: persists across `fork` and `exec` even if `exec` executes a set-UID or set-GID binary.
- UID-specific: each user has its own keyring. It is accessible across all processes running as the same UID the keyring was created for.

User–space applications have access to keys stored in a keyring by using various system calls introduced along with the keyring API as defined in `keyctl.c`:

`add_key(type, description, payload, length, keyring)`
Create a new key or keyring. To create a new keyring, `type` has to be set to *keyring* and the value of `description` specifies its name. New keys can be created by using *user* as type followed by a unique name in the `description` parameter. The newly added key will be linked to the keyring `keyring`. If a key

with description `description` already exists, it will either be updated if the key type provides an `update()` function or replaced. The keyring `keyring` must be present already. Further, custom key types that have been registered using `register_key_type` also denote accepted values for the `type` parameter.

`request_key(type, description, callout, destination)`
Searches keyrings for a key of type `type` that matches `description`. If no key could be found and `callout` is set, the user–space application `/sbin/request-key` will be called with `callout` as the first argument to construct a key. Otherwise an error is returned.

`keyctl(option, ...)`
Multi–purpose function to manipulate key rings. A total of 17 options is currently available. They range from common options like search and read up to the manipulation of timeout settings or keyring merging.

Due to the complexity of the key management system, well–commented example source code is included in `extra/key_mgmt`. Especially with the use of the callout parameter during `request_key`, a kernel space encrypting file system has a powerful, yet easy way of requesting keys from user space. In such cases, the kernel facility would only be used for keeping track of keys while user space is responsible for obtaining them from various sources. With regards to section Secure storage, it allows the connection between user space key storage facilities for permanent storage with kernel space secure management of keys while they are in use.

Findings

The kernel's crypto API is very powerful as it already provides support for all state–of–the–art ciphers and hashes. Kernel modules can rely on their presence and don't need to provide their own implementations. Due to their presence in the mainline kernel they automatically receive maintenance. This is very important as it leads to very performant implementations. Using

```
grep -rn crypto_register_  /src/linux/arch/x86/crypto
```

where `/src/linux` refers to the root directory of the Linux kernel source code, it can be seen that numerous algorithms are registered. Those algorithms make use of hardware features like AES–NI or SSE which further increase their performance.

A limitation of the kernel keyring is that it only exists in kernel space. This implies that a reboot for example empties the keystore and it must be re–constructed as necessary from user–space. As mentioned in chapter "Secure storage", one needs a combination of KWallet or gnome–keyring and the kernel keyring.

2.8.3 Device–Mapper

The device mapper forms the basis for the underlying encryption layer *dm–crypt*. It is a framework in the Linux kernel available since version 2.6. Commonly, it allows the mapping of a block device onto another, therefore acting as a low–level volume manager. When establishing such a mapping, the device mapper creates *virtual* block devices that can be used in I/O operations. Before elaborating on the nature of dm–crypt, the environment it is operating in needs to be understood.

As previously explained, the VFS layer handles I/O requests from user space and invokes file system specific operations to handle these request. As a result, code complexity in the VFS layer is moved towards specific file systems. The very same principle holds true for I/O facilities underneath the VFS layer.

To increase I/O performance, Linux uses a facility called the *page cache* that aims to reduce the number of disk I/O by storing data in memory that would otherwise require rather expensive read–in from an underlying device. When issuing read requests, the kernel checks if the page cache already contains data for a given file and if so, returns it immediately. Likewise, write operations are first inserted into the page cached and the associated pages are marked *dirty*. Regardless of the operations, the page cache needs to *sync* its dirty pages with the block device eventually. It does so by submitting a `struct bio` to the generic block I/O layer, which holds an array of pointers to memory pages that need to the written to disk. It should be noted that the write operations are performed asynchronously. The sender of a write request has no knowledge about when data is finally written to disk. Therefore, the submitter has to specify a callback function `bi_end_io` that is invoked when the request has been processed. As depicted in figure 2.7, the submitted bio objects are inserted in the request queue of the block driver. In case of the device mapper, the bio objects are cloned and the `bi_end_io` callback is changed to the device mapper's end–of–IO handler. This is necessary as the device mapper may need to perform special end–of–IO actions that would otherwise be impossible to carry out. Furthermore, in case of software raid targets, the bio object has to be written to multiple devices, so an I/O operation is only considered to be successful if all objects have been written.

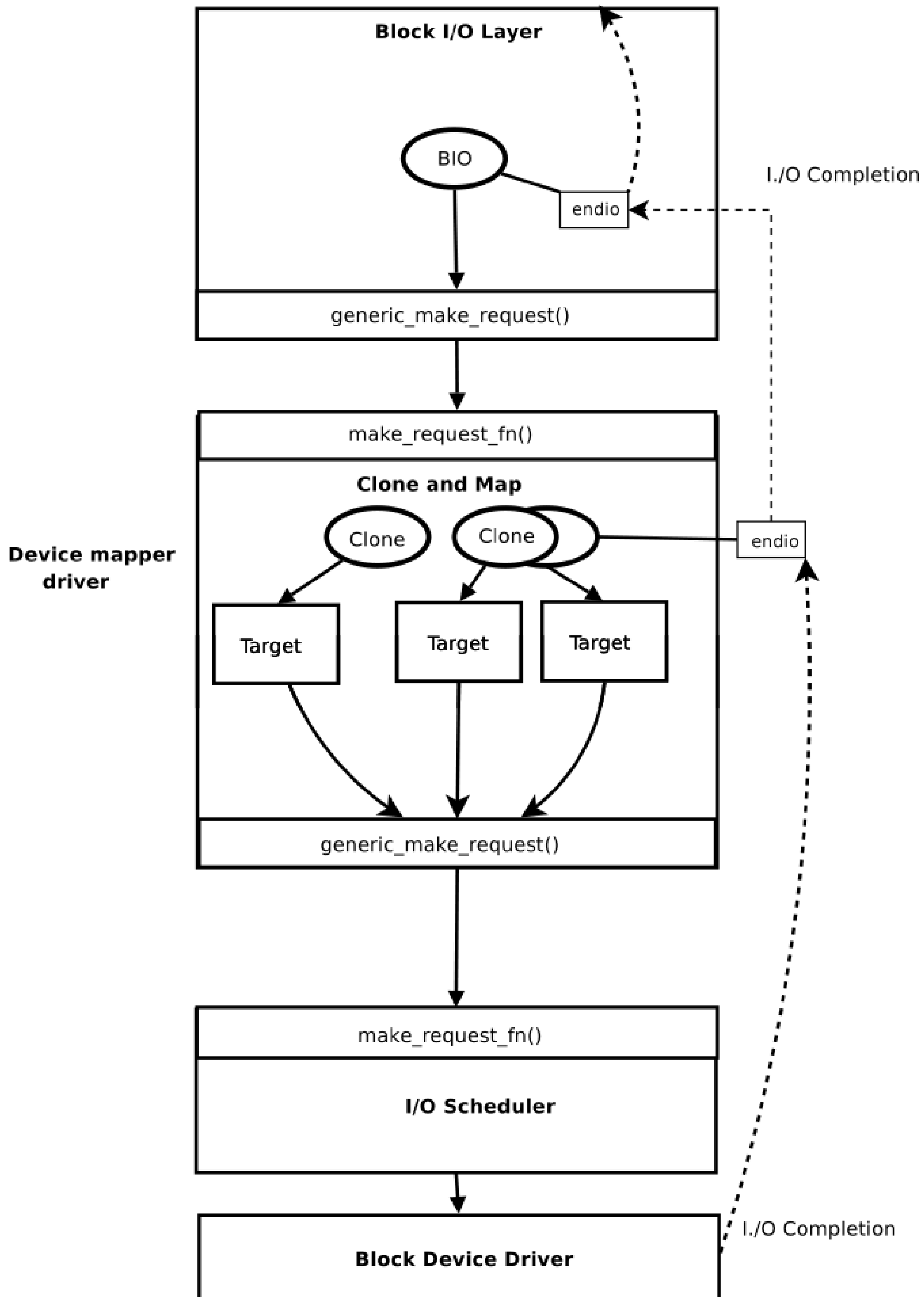
Depending on the scheduling strategy of the underlying device, requests might further undergo reordering or coalescing to maximize throughput[9].

As mentioned above, the device mapper creates virtual block devices and acts as a common framework for filters (*targets*). A filter has access to the request data and is able to replicate or modify it while it passes the filter. Common targets supported on recent kernel versions include:

`dm-zero`
: Acts similar to `/dev/zero`. That is writes are discarded and reads always return

[9]this strategy is implemented by I/O schedulers which can be viewed and changed by reading or writing to `/sys/block/sdX/queue/scheduler`.

Figure 2.7: Linux Device Mapper [Vel08]



byte value `0x00`.

`dm-raid`
: Implements a software Redundant Array of Independent Disks (RAID) target that supports RAID levels 1 (mirroring), 10 (striped mirrors), 4 (dedicated parity disk), 5 (block-level striping with distributed parity), and 6 (block-level striping with double distributed parity).

`dm-crypt`
: Provides transparent encryption using the kernel crypto API.

`dm-multipath`
: Allows the configuration of multiple I/O paths between server nodes and storage arrays, therefore increasing availability and performance.

In addition, it is possible to stack device mapper targets. For instance, using the crypt target to encrypt data, but further utilize the RAID target to distribute data to various locations.

Due to the location of the dm–crypt target in the File I/O hierarchy, block device encryption offers the fastest I/O rates as there is very little overhead in the process. Furthermore, without knowledge of the encryption key, virtually no information about the file system or individual data is exposed to a third party. Drawbacks of such a solution include:

- The partition must be pre–allocated. Resizing might become a tedious task.
- Each file is encrypted with the same key.
- Backup applications typically access files once the partition has been mounted. This implies that the files will be backed up in the clear, therefore requiring additional encryption for the backup itself.

To perform cryptographic tasks, the dm–crypt target uses the kernel Crypto API as introduced in the previous section. In addition, starting from version 1.6, supported cipher modes for the userspace tool cryptsetup can also be benchmarked to select a cipher that yields maximum I/O rates when reading and writing from the target device. On Linux, one can use `cryptsetup benchmark` to start the evaluation.

2.8.4 LUKS

Initially when using the dm–crypt target, a password is used to derive a master key. Each time the volume is mounted, the user needs to enter that very same password again to unlock the volume as no information about cryptographic properties is written to disk. As explained in section 2.4 “Password Expansion”, user passwords either

suffer from low entropy that weakens or nullifies the benefits gained through encryption, or if satisfying rather complex password policies, end up on a post–it note on the screen which has the same effect. Therefore, a standard for an on–disk format for encrypted hard drives known as the Linux Unified Key Setup (LUKS) has been developed. Described in [Fru11], LUKS introduces a dedicated area at the beginning of partitions to store information needed in order to regain access to encrypted data. The main header used by LUKS is outlined in table 2.9. Each key slot is comprised of various fields as shown in table 2.10.

The user supplied password is used during the initial partition setup to encrypt the randomly generated master key that is used for bulk data encryption. Further, the header includes a checksum of the master key to decide whether a given user password decrypted the master key successfully during the mount operation. A total of eight user keys can be set to allow access to the encrypted data.

An interesting feature of LUKS is the handling of key changing events. Commonly, if a password is to be changed, the old values are simply replaced with new content. The problem however arises when using modern hard disks as they handle inaccessible sectors due to aging differently compared to obsolete models. Bad blocks are simply remapped to spare sectors. Due to the small size of the master key however (typically 16 or 32 bytes), such a remapping would affect the whole key as it easily fits into one sector which would then become inaccessible and may be recoverable during a forensic examination. Therefore, LUKS artificially inflates the size of the master key to distribute it across multiple sectors. When later erasing the key, the probability of destroying a single part of the key increases with the number of *stripes*.

Table 2.9: LUKS header layout

Offset	Size (b)	Description	Example
0	6	Magic value	{'L', 'U', 'K', 'S', 0xBA, 0xBE }
6	2	Header version	1
8	32	Cipher name	"aes"
40	32	Cipher mode	"cbc-essiv:sha256"
72	32	Hash algorithm specification	"sha1"
104	4	Offset of bulk data	2056
108	4	Number of key bytes	32
112	20	Master Key checksum	0x19 0x90 0x3B ...
132	32	Master Key salt	0x9C 0x86 0x7A ...
164	4	Iteration count for PBKDF	48375
168	40	Key unique identifier	691b2e50-...
208	384	Storage for eight key slots	

Table 2.10: LUKS key slot layout

Offset	Size (b)	Description	Example
0	4	Slot state information	0x00 0xAC 0x71 0xF3
4	4	Iteration count for PBKDF	193814
8	32	Salt for PBKDF	0x68 0xEC 0xB3 ...
40	4	Key Material offset	8
44	4	Number of anti–forensic stripes	4000

As a generic hash function like SHA–1 is used to inflate the key, a single missing piece is sufficient to render the recovery process pointless.

To ease the setup of LUKS encrypted partitions, the *cryptsetup* tools have been extended to support LUKS volumes. On Linux, one can use a variety of commands as described in `man 8 cryptsetup` to create and mount volumes, but also to manage key slots. These operations cover addition, removal, and key change of specific slots in the LUKS header.

Encrypting a partition is made easy using the aforementioned cryptsetup tools. The following example demonstrates the use of dm–crypt and LUKS on a loop device:

1. `dd if=/dev/urandom of=/tmp/secret.img bs=1M count=16` creates a 16 MB file that will hold the encrypted data
2. `losetup /dev/loop0 /tmp/secret.img` associates the file with a virtual block device
3. `cryptsetup` *`options`* `luksFormat /dev/loop0` will prompt for a password and write the header to disk (or to the container file in this case).
4. `cryptsetup luksOpen /dev/loop0 lo_file0` opens the LUKS device and creates a mapping in /dev/mapper.
5. `mkfs.ext4 /dev/mapper/lo_file0` initializes a file system on the given device.
6. `mount /dev/mapper/lo_file0 /mnt` mounts the container to the given location.

I/O operations on the mountpoint will then result in data being encrypted and decrypted on the fly as they pass through the block I/O layer. As mentioned before, LUKS supports up to eight key slots for a given container. These keys can be added and removed using one of the following commands:

- `cryptsetup luksAddKey /dev/loop0`
- `cryptsetup luksRemoveKey /dev/loop0`

- `cryptsetup luksChangeKey /dev/loop0`
- `cryptsetup luksKillSlot /dev/loop0` *slot number*

Table 2.11 shows various options that can be used during creation of the encrypted container / partition in step 3. Built–in defaults can be viewed by issuing `cryptsetup -help` at the command line.

Findings

LUKS is an ideal solution for disk encryption as it hides as many properties of the partition as possible. Without unlocking the volume, an adversary can only learn very little about stored information. However, on a workstation encrypted partitions are unlocked most of the time as the user is using them. Due to the fact that encryption on the device mapping layer does not offer per file encryption, any user with appropriate UNIX permissions has access to encrypted files. In terms of performance, encryption within the device mapping layer would be desirable.

A major drawback of LUKS is that it does not support on–the–fly initial encryption like Apple's FileVault or Microsoft's BitLocker do. Instead, the volume must be reformatted which is something the user does not want, but that is exactly what is necessary if full disk encryption should be supported on the SafeGuard Linux client if a policy for full disk encryption should be enforced.

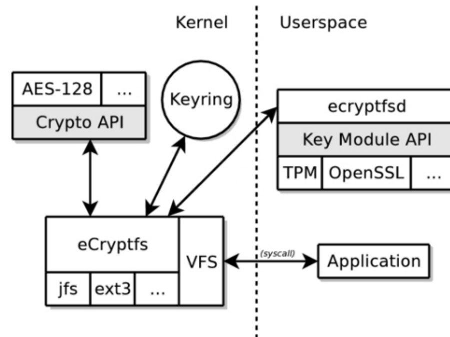
2.8.5 eCryptfs

The enterprise Cryptographic Filesystem for Linux (eCryptfs) is an encrypting file system situated in the VFS layer of the kernel. It provides transparent file encryption on top of existing file systems. The term *enterprise* refers to its design goal of versatile key management through the support of multiple key material sources such as Trusted Platform Module (TPM) or the GnuPG keyring [Hal05]. Figure 2.8 shows the components involved in the file encryption process. On module load, eCryptfs registers itself as a file system with the kernel VFS layer and allows users to mount directories. When data is accessed, eCryptfs passes the data on to the underlying file system (the *lower file system*) while performing necessary cryptographic operations on the data. Due to the fact that eCryptfs is a stackable file system, it has access to

Table 2.11: Sample LUKS creation options

Name	Description	Example
cipher	Selects the cipher algorithm to use	aes-xts-plain64
key-size	Specifies the key size	256
uuid	UUID for device to use	"a random string"

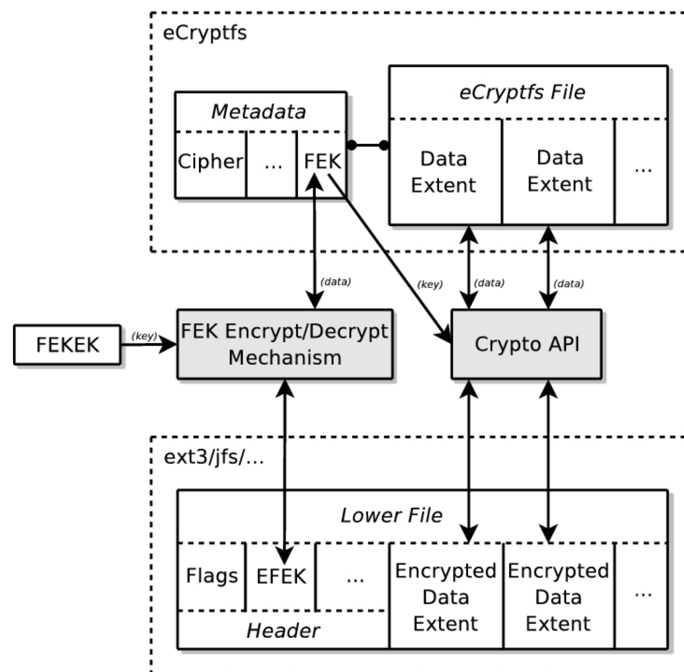
Figure 2.8: eCryptfs architecture [Hal07]



individual files. Compared to dm–crypt, this allows a per–file encryption with both a random encryption key and initialization vector.

For key management, eCryptfs maintains a bi–directional communication channel with the per–user user–space daemon *ecryptfsd*. Figure 2.9 shows the encryption and decryption process that is performed when receiving system calls like read or write. As there is virtually no documentation of eCryptfs available, this section shall also serve as a guide for further implementations. This should especially be helpful if the prototype as introduced in chapter 5 "Development of user–space encryption using FUSE" is moved to kernel space to achieve better I/O rates. References to source code encountered in this section refer to source code shipped with the Linux kernel under `fs/ecryptfs` if not stated otherwise.

Figure 2.9: eCryptfs crypto overview [Hal07]



As mentioned before, support for eCryptfs can either be obtained by inserting the kernel module *ecryptfs* using `insmod` or `modprobe`, or by compiling a kernel with support directly included[10]. Recent kernel versions however already include a module for ecryptfs.

Initialization

The Initialization routines of eCryptfs are located in `main.c` and perform a series of initialization routines:

- create a series of caches for authentication tokens, file headers, dentries, and inode structures
- create various entries in `/sys` that can be used to obtain information about the state of eCryptfs.
- start a kernel thread that is used to access inodes in the lower file system.
- initialize the messaging system. That is, set up data structures that contain information about ecryptfsd sessions in user space. In addition, a miscellaneous device is created in `/dev` that is used to communicate with these daemons.
- The file system is made available for mounting using `register_filesystem`.

As long as no directory is mounted using eCryptfs, the file system does not perform any actions. To make use of it, one has to mount a file system.

```
mount.ecryptfs /tmp/1 /mnt
```

will make the contents of `/tmp/1` available under `/mnt`. Files accessed through `/mnt` will therefore be encrypted and decrypted on the fly. This also becomes obvious when creating empty files in `/mnt` using the `touch` command. While the files show a size of 0 bytes when inspecting `/mnt`, `/tmp/1` shows that each of them already contains data. This is the result of eCryptfs having already generated random file encryption keys and initialization vectors and has written that information in the file header.

eCryptfs I/O

As with all file systems, the functionality of eCryptfs is accessed through multiple sets of callbacks invoked by the VFS layer when file I/O needs to be performed. The mount system call as stated before will initialize a `struct superblock` object that will carry two important pointers to the following data structures:

[10]directly included refers to kernel builds with `CONFIG_ECRYPT_FS` set to `'Y'` instead of `'m'`.

superblock operations `s_op`

Superblock operations are used to perform low–level file system operations like creating or destroying an inode. Additionally, they include methods to read and write to and from inode objects.

dentry operations `s_d_op`

dentry objects represent the file system tree structure as can be seen with tools like `ls` and further contain a link to an underlying inode object.

When user space applications use VFS system calls like `open` or `stat`, the VFS layer needs to lookup the dentry structures within the dcache, or if not found there, construct one. The dentry operations a file system has to specify provides functions for the dentry cache to keep information up to date. Since eCryptfs is stacked on top of an existing file system, it simply relays the call to the lower dentry – e.g. an ext3 inode.

Additional to the superblock operations, eCryptfs also has to inform the VFS layer how to handle common operations like `read` or `write` system calls. It does so by setting the inode's file operation pointer to a set of callbacks that implement the main eCryptfs logic. The VFS can then simply call the file's `read` or `write` functions which in turn trigger the encryption or decryption of a page when necessary. It should be noted however that cryptographic operations take place on a per page basis. Since the VFS contains lots of complexity, the call stacks for both `read` and `write` are shown in Tables 2.12 and 2.13. To maintain state information between calls, eCryptfs uses the structure `ecryptfs_crypt_stat`, defined in `ecryptfs_kernel.h`. This structure

Table 2.12: eCryptfs read method

Level	Name	Description
1	`read` system call	read from a file (descriptor)
2	`vfs_read`	does some sanity checks and invokes the file system's `read` method
3	`do_sync_read`	synchronous read
4	`ecryptfs_read_update_atime`	eCryptfs' read method as specified in the `struct file_operations`
5	`generic_file_aio_read`	generic file system read routine
6	`do_generic_file_read`	generic file read routine
7	`filemap_fault`	read in file data for page fault handling (if page is not mapped)
8	`ecryptfs_readpage`	relay call to the lower file system
9	`ecryptfs_decrypt_page`	Performs the actual decryption

Table 2.13: eCryptfs write method

Level	Name	Description
1	`write` system call	write to a file (descriptor)
2	`vfs_write`	does some sanity checks and invokes the file system's `write` method
3	`do_sync_write`	synchronous write
4	`generic_file_aio_write`	write data to a file
	[...]	
11	`generic_writepages`	walk the list of dirty pages and `writepage()` all of them
12	`__writepage`	Wrapper; calls the file system's writepage
13	`ecryptfs_writepage`	Encrypts a single page
14	`ecryptfs_encrypt_page`	Performs the actual encryption ...
15	`ecryptfs_write_lower`	... and passes it on to the lower file system

contains the actual Data Encryption Key, the initialization vector and the used cipher along other bookkeeping information to keep track of the lower inode. A common technique also employed by eCryptfs is C inheritance to extend the kernel's inode structure. Generally, the kernel does not (need to) know any specifics about a file system. Instead, it merely requires file system functions to return a data structure it can understand. In case of inode creation, it expects a `struct inode`. Since eCryptfs or any other file system might want to store additional data within an inode, a method is needed to satisfy both the kernel's API and the file system's information requirements. In C++, one would use class inheritance and polymorphism to accomplish this setup. In C however, these techniques do not exist, therefore another approach must be used.

Snippet 2.7: Data inheritance in C

```
1  /* A VFS inode as understood by the kernel */
2  struct inode {
3      dev_t    i_rdev;
4      loff_t   i_size
5      umode_t  i_mode;
6      [...];
7  };
8
9  /* An eCryptfs inode with additional data */
10 struct ecryptfs_inode {
11     unsigned char ei_key[];
12     unsigned char ei_iv[];
13     struct inode  ei_ino;
```

```
14 };
15
16 /* Called by the VFS layer - expects a VFS inode */
17 struct inode *ecryptfs_new_inode(void)
18 {
19     struct ecryptfs_inode *ino = ...;
20     return &ino->ei_ino;
21 }
22
23 /* Used by eCryptfs to get the cipher key from an VFS inode */
24 unsigned char *ecryptfs_key_ptr(struct inode *i_ptr)
25 {
26     struct ecryptfs_inode *ei_ptr;
27
28     ei_ptr = container_of(i_ptr, struct ecryptfs_inode, ei_inode);
29     return ei_ptr->ei_key;
30 }
```

The `container_of` macro is a very common macro used throughout the kernel that is defined as follows:

Snippet 2.8: The container_of macro

```
1 #define container_of(ptr, type, member)                        \
2 ({                                                             \
3         const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
4         (type *)( (char *)__mptr - offsetof(type,member) );     \
5 })
```

As can be seen in Listing 2.8, the macro takes a pointer to a struct `inode`, the name of the parent structure and its own name within the parent. It then uses the `offsetof` macro on the memory layout as depicted in Figure 2.10 to calculate the offset of the requested member and subtracts this value from the supplied pointer. It should be noted however that this technique only works if the “superclass” `inode` is within the memory range of the “subclass” `ecryptfs_inode` and is therefore not a pointer.

Before data may be read or written, the corresponding file needs to be created or opened. [Hal05, Fig. 2] outlines the process of creating new files. The actual content of a file is further preceded by the file header. Table 2.14 describes the full file header as created by `ecryptfs_write_headers_virt`. The information is taken from a comment block preceding the function. The packet set starting at offset 26 is described by [CDFT98] and in the current eCryptfs version is either a Tag 3 (Symmetric-Key Encrypted Session-Key) or a Tag 11 (Public-Key Encrypted Session Key) packet and contains the Data Encryption Key in encrypted form. When opening an eCryptfs encrypted file, the header is parsed to re-gain the encrypted Data Encryption Keys. eCryptfs then uses the kernel keyring API to locate a Key Encryption Key that can be used to decrypt the Data Encryption Key (DEK). At this point, the Kernel’s `request_-`

Figure 2.10: container_of macro – memory layout

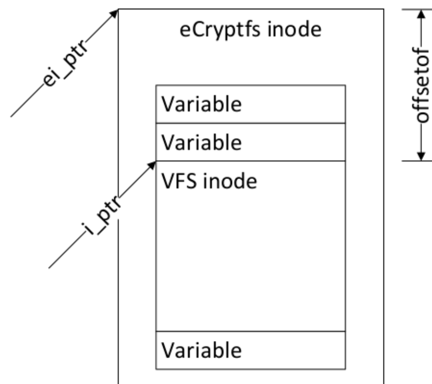


Table 2.14: eCryptfs file header

Offset	Size	Description
0	8	Unencrypted file size
8	8	Special marker (magic value)
16	1	Header version
17	2	Reserved
19	1	Additional flags
20	4	Extent size
24	2	Number of header extents at front of file
26	x	Begin RFC 2440 authentication token packet set

`key` function as introduced in subsection 2.8.2 "Key Management" can be used to obtain a key from userspace applications.

File I/O as handled by eCryptfs is based on unit sizes of *extent*. The size of an extent is determined by the page size of the host creating the file (commonly 4096 bytes on Linux). Cryptographic operations are performed using vectored I/O with scatterlists on units of full pages, thus an eCryptfs encrypted file is always a multiple of page size in size with the exact length being stored and refreshed in the file header. In the current version, multiple ciphers are supported[11] with AES-128 being the default.

2.9 User–space Encryption

User–space encryption covers those parts of encryption that are handled entirely by applications under control of a user of the system. A basic example of file encryption

[11] the full list is available in crypto.c or when mounting a file system using `mount.ecryptfs`

is a program that takes three parameters: a secret key and input and output file. It reads the file, encrypts it with the secret key and writes the result to the specified output. Consequently, those applications are characterized by their simplicity, but at the same time they provide the least flexibility if the situation changes.

The goal of using encryption is to ensure confidentiality. In enterprise environments a policy might be in place to enforce encryption of valuable data. If aforementioned applications are used, there exists no possibility to ensure that written data has been encrypted. It is up to the user to utilize appropriate applications to encrypt data. Furthermore, even if file encryption is performed, data has already been written to disk before encryption took place. Additionally, even if the source file is deleted, sensitive content may remain on disk. To securely erase affected sectors on disk, special procedures described by standards like [KRSS14] and [NSA14] have to be used.

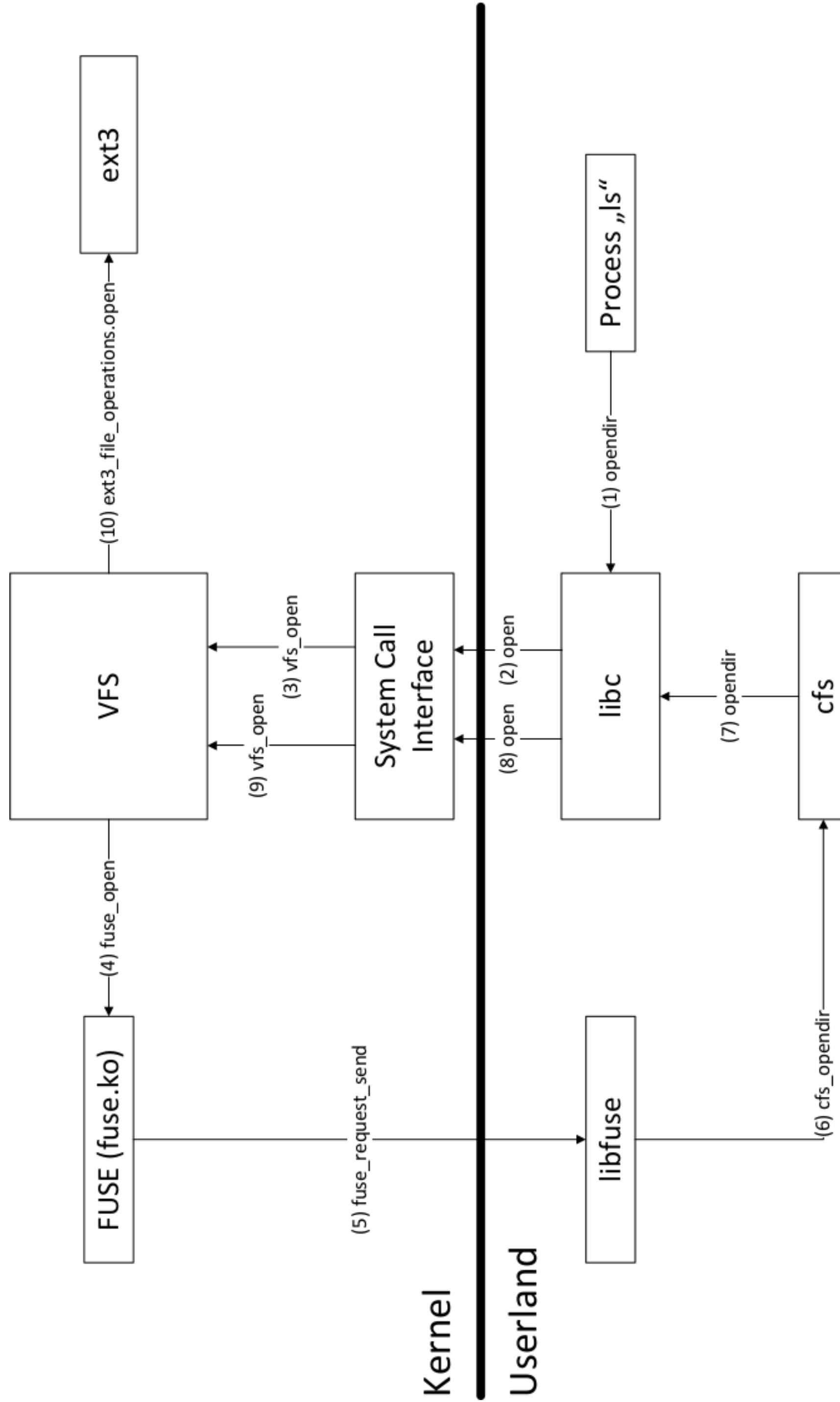
Another aspect of file encryption is the achieved performance. When using simple file encryption tools, data has to be written, then encrypted using a special tool that needs to read the very same information once more, process it and write the results back to disk. Afterwards it may then has to use a multiple rounds / multiple overwriting pattern to securely erase the source data. The performance loss is proportional with the data size to be encrypted and if secure overwrite is performed, increases even more per round. In addition, while the operation is in progress, the file occupies twice its original size on the hard drive. Memory consumption also has a direct influence on the overall performance. Clearly, those tools are therefore not suitable for enterprise file encryption as disadvantages outweigh advantages.

Aforementioned tools operate on the highest possible layer of file handling. They take actions after files have been stored. Due to their functioning they have to redo most of the work. With respect to the Linux Filesystem architecture as presented in section section 2.7 “The Virtual File System Layer”, performant and secure encryption is only possible for software located within the VFS layer. Tools in userspace simply do not and cannot offer the granularity for file handling that is needed in order to perform the required tasks efficiently. With reference to figure 2.6, an alternative would only be possible to operate in kernel space. As a consequence, ordinary system users would be unable to benefit from performant file system as mounting of such file system requires system administrator privileges.

2.9.1 FUSE – Filesystem in User Space

To remedy restrictions implied by having to use the mount system call, FUSE provides a more powerful way of data handling as opposed to plain userspace tools. From a high–level point of view, FUSE is a two–part application that operates in both kernel space and user space. As stated in [FUS11], this approach makes it possible to “implement a fully functional file system in a userspace program.” A general overview is illustrated in figure 2.11. In contrast to simple file encryption tools, the encryption layer can be implemented as a standalone file system in user space. FUSE

Figure 2.11: FUSE Architecture



allows the registration of a set of function pointer callbacks that will be used to hook into well–known system calls like `open()` or `read()`. Consequently, data streams to the underlying file system pass through the file system and can be manipulated on a per–byte basis.

libfuse

Custom file system implementations do not communicate with the FUSE kernel module directly. Instead, *libfuse* is used in user–space to set up a bi–directional communication channel with the kernel module. File systems are then registered via libfuse by a call to `fuse_main()`, which expects a set of callbacks of type `struct fuse_operations`. This structure, defined in `fuse.h` can be understood as a set of instructions to libfuse which can then invoke the correct function if a operation is performed on one of the files located in the FUSE file system. An excerpt of `struct fuse_operations` is provided in listing 2.9.

Snippet 2.9: FUSE operations - sample

```
1  struct fuse_operations {
2      int (*getattr) (const char *, struct stat *);
3      int (*open) (const char *, struct fuse_file_info *);
4      int (*read) (const char *, char *, size_t, off_t,
5                   struct fuse_file_info *);
6      int (*write) (const char *, const char *, size_t, off_t,
7                    struct fuse_file_info *);
8      [...]
9      int (*release) (const char *, struct fuse_file_info *);
10 };
```

The `fuse_main()` function sets up a communication channel by either performing a `mount()` system call if supported or by calling *fusermount*. The result is an open file descriptor for the FUSE character device `/dev/fuse` as exposed by the kernel module. Eventually, the library enters `fuse_session_receive_buf()` which receives events from the kernel module and passes them on to `fuse_session_process_buf()`. The function parses the event and invokes one of the users supplied callback functions. The result is then written back to the file descriptor which in turn allows the kernel to report a status back to the application that originally performed the call.

fuse.ko

fuse.ko denotes the kernel modules that needs to be present in order to use custom file systems through libfuse. Since FUSE has been merged into the mainline kernel version `2.6.14`, the module is present on all recent versions. This is important as the use of proof–of–concept implementation does not require modifying kernel source code, although the user mounting the file system has to a member of the group `fuse`

as the character device exposed by the module has its permissions set to 0660, thus it does not allow arbitrary users to read from the device.

The FUSE file system residing in kernel space follows the same implementation rules with regards to the VFS layer as any other file system. It has to register itself using register_filesystem during its initialization and unregister_filesystem during unloading. Once it has been initialized, users can use the fusermount command to mount file systems of type FUSE as long as they are in the fuse group[12].

FUSE at runtime

The simplest yet fully functional file system implementation using FUSE would just relay all calls to the underlying file system. For demonstration purposes, the *relfs* (relay file system) shall serve as an example file system throughout this chapter, accompanying explanations.

A basic file system shall at least offer the functionality to obtain directory listing through ls and to work with ordinary files. Special files like pipes or sockets are not supported.

When creating a file system, the primary ingredient is the fuse_operations structure as mentioned before.

Snippet 2.10: RelFS main

```
1  int main(int argc, char **argv)
2  {
3      struct fuse_operations relfs_ops;
4      return fuse_main(argc, argv, &relfs_ops, NULL);
5  }
```

Because no operations are specified in relfs_ops, the file system is not yet functional.

Snippet 2.11: RelFS getattr function

```
1  static int relfs_getattr(const char *path, struct stat *sb)
2  {
3      return (stat(path, sb) ? -errno : 0);     /* Issue stat() system call */
4  }
5
6  int main(int argc, char **argv)
7  {
8      struct fuse_operations relfs_ops = {
9          .getattr = relfs_getattr
10     };
11     return fuse_main(argc, argv, &relfs_ops, NULL);
12 }
```

[12] one can use the id command to find out if this is the case. Otherwise fusermount will give the error Operation not permitted due to the permissions of the character device /dev/fuse

The getattr function is used extensively by FUSE due to in–kernel caching of meta data. Therefore this function is called quite frequently to obtain up to date information about files. This is important as encrypted files may require additional computation after having performed stat on them. An example would be eCryptfs where the actual file size is stored in the header, whereas the file size is always a multiple of page size. The file system should report the exact file size as there might are applications that perform stat on the file and then try to read exactly the amount of bytes returned by stat and treat any other result as error.

To enable directory listing, one has to implement the opendir, readdir and closedir functions.

Snippet 2.12: RelFS directory function

```
1  static int relfs_opendir(const char *path, struct fuse_file_info *fi)
2  {
3      DIR *dp = opendir(path);
4      if (!dp)
5          return -errno;
6      fi->fh = (uintptr_t)(void *) dp;
7      return 0;
8  }
9
10 static int relfs_closedir(const char *path, struct fuse_file_info *fi)
11 {
12     return (closedir((DIR *) fi->fh) ? -errno : 0);
13 }
14
15 static int relfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
16              off_t offset, struct fuse_file_info *fi)
17 {
18     DIR *dp = (DIR *) fi->fh;
19     struct dirent *de;
20
21     de = readdir(dp);
22     if (!de)
23         return -errno;
24
25     do {
26         if (filler(buf, de->d_name, NULL, 0) != 0)
27             return -ENOMEM;
28     } while ((de = readdir(dp)) != NULL);
29
30     return 0;
31 }
32
33 int main(int argc, char **argv)
34 {
35     struct fuse_operations relfs_ops = {
36         .getattr    = relfs_getattr,
```

```
37          .opendir    = relfs_opendir,
38          .releasedir = relfs_closedir,
39          .readdir    = relfs_readdir
40      };
41      return fuse_main(argc, argv, &relfs_ops, NULL);
42  }
```

Although the file system can still not handle any file operations, the functionality implemented so far already outlines all functionalities also needed for file I/O. As can be seen in the `*dir` functions, the FUSE library passes a `fuse_file_info` structure parameter. That parameter will be used to maintain state information across multiple calls to different function callbacks. The `fh` (file handle) field is large enough to hold a pointer to arbitrary data. *cfs* will use this field to keep a pointer to associated cryptographic metadata like keys or initialization vectors.

If the file system is now compiled into a binary, it has to be linked to the FUSE shared library that is already installed on the system[13].

```
cc main.c -lfuse -o relfs
```

The resulting binary can then be started as follows:

```
./relfs <mountpoint>
```

Given an arbitrary, empty directory as mountpoint, a new entry should be visible when issuing the `mount` command in a terminal. `cd`'ing into that directory and executing `ls` should also already output directory entries. In the case of relfs, it outputs the content of the root directory / because the paths passed to the callback functions are relative to the mountpoint. To create a mapping between a mountpoint and its source directory, one has to maintain his own path to the root directory and prepend the path to each `path` parameter passed one of the callbacks. Since maintaining such a mapping is a common task, FUSE offers built-in support through the *subdir* module:

```
./relfs -omodules=subdir,subdir=/tmp /mnt
```

Again, `cd`'ing into the mount directory `/mnt` and issuing `ls` will now list the content of the `/tmp` directory as the `path` parameter to the `opendir` callback will have the string as specified for the `subdir` command line parameter prefixed. Additional command line parameters available for the FUSE mount call are listed on the man page `man 8 mount.fuse`. Important parameters used in the prototype are

fsname
Specifies the file system name

[13]One can use the distributions package manager to install `libfuse2` – e.g. `aptitude install libfuse2` on Debian

use_ino
: Honor the `st_ino` field in kernel functions `getattr` and `fill_dir`

auto_umount
: Automatically unmount the file system on process exit

large_reads and max_read=N
: raises the maximum buffer size passed to the `read` function to N bytes

big_writes and max_write=N
: raises the maximum buffer size passed to the `write` function to N bytes

modules=subdir
: automatically prepend a given string to all functions that take a path parameter

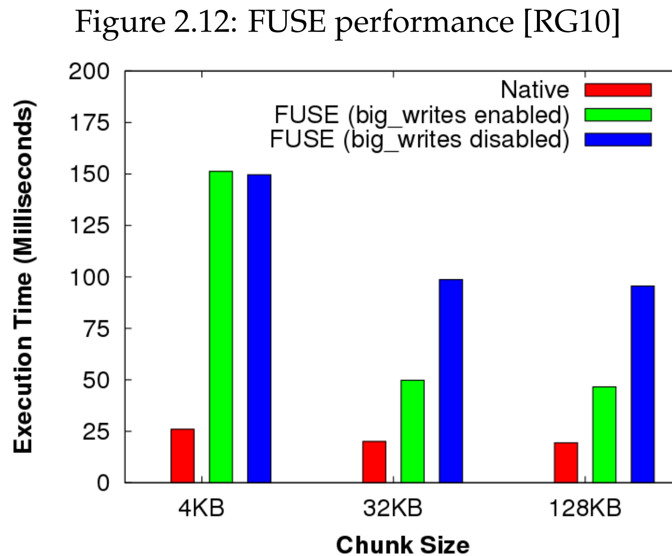
allow_other
: allow access to other users

default_permissions
: enable permission checking by kernel

The parameters `allow_other` in conjunction with `default_permissions` are important if the file system should grant access to users different from the user who mounted the file system. By default, only the mount user has access to files through the mountpoint, not even the root user can bypass the access checks. Specifying `allow_other` lifts that check and causes FUSE to forward permission checks to the file systems `access` function. Due to the complexity of such a security sensitive operation, one can further specify `default_permissions` to enforce permission checks within the kernel. Therefore, the combination of both parameters determine if a file system needs to implement the `access` function.

The parameters `large_reads` and `big_write` directly influence how much data will be passed to the file system's `read` and `write` functions. Their values should range between 4 KB (one page on most Linux systems) and 128 KB (the maximum supported size in the kernel). Not providing these parameters results in read and write calls defaulting to requests being issued with sizes of 4 KB, which significantly degrades performance as shown by [RG10]. These numbers reflect the execution time needed when writing a 16 MB file for varying values for the max_write parameter. As can be seen, `big_write` plays a major role when it comes to performance. The reason behind the improvement is that less write calls need to be issued for a given buffer to be written. When only a native file system like ext4 is used, the operation triggers only two mode switches – to and from kernel mode. When writing to a FUSE mounted file system, the kernel needs to perform a mode switch to kernel mode, copies the data to the kernel's page cache, but it also needs to forward the data to *libfuse* which in turn copies the data and sends it to the FUSE file system.

Figure 2.12: FUSE performance [RG10]



2.10 Interim results

This chapter has outlined the foundation for implementing the prototype file system *cfs*. Introduced cipher algorithms have been evaluated in terms of speed and their area of application, which is important when it comes to modes of operation that turn a block cipher into a stream cipher as there are properties that must be followed in order to protect encrypted information in a proper way.

The evaluation of persistent storage facilities was limited to two of the most popular applications encountered on the typical end–user desktop machine. A circumscribed source code audit has revealed the weaknesses of KDE's default password store KDE, nevertheless it should be supported as it is widely used and a transition concerning the on–disk format towards a compliant GPG based format is being developed by the time of writing this thesis.

The required basis for supporting alternative / stand–alone persistent storage facilities has been introduced in form of D–Bus as a form of extended inter-process communication system.

The rest of this chapter was devoted to elaborating on the various layers of the Linux kernel where one can employ techniques to perform data encryption. Although not implemented, the respective sections should serve as a source of information if the resulting software as introduced in the next chapters should be implemented in–kernel or if full–disk encryption support should be added. To further aid in the transition, the naming of functions and interfaces and the architecture of the prototype itself will try to resemble as much as possible from Linux kernel, so the user–space implementation serves as a fully functional basis for further development.

CHAPTER 3

Development of shared component

While the next chapters describe the implementations of specific applications, the following sections aim to address functionality shared among them. The goal during their implementation was to provide a single service while keeping dependencies to other tools and libraries as low as possible to allow their inclusion in other projects as they offer useful functionality.

3.1 Configuration file reader

libcfg provides an API to read values from a given configuration file. Currently, the underlying library is *libconfig*[1] as it is easy to use and runs with low memory consumption. Applications can utilize the header file located at `cfg/cfg.h` to make use of the interface. libconfig uses a simple language that makes up a configuration.

Values
: Scalar values of type integer, long long, double, or string.

Settings
: A simple `name = value;` or `name : value;` pair.

Groups
: One or more settings enclosed in curly brackets: `{` setting . . . `}`.

Arrays
: Similar to groups, but each member setting must be of scalar values of the same type. Array members are enclosed in square brackets, but an array is not required to have any members.

[1] libconfig is a very compact and easy to use library to process configuration files. More information and documentation is available at http://www.hyperrealm.com/libconfig/

Lists
: Group of items enclosed in round brackets. Lists can contain ordinary settings, groups, arrays, or other lists.

Comments
: Lines starting with # are ignored entirely. `/* ... */` excludes a comment from normal text while // excludes till the end of the line.

A sample configuration file conforming to the above mentioned grammar can be seen in listing 3.1.

Snippet 3.1: libconfig grammar

```
# A comment line
server = {
    hostname = "192.168.1.1";
    port = 443;
    use_ssl = true;
};

/* A list of user credentials */
http_users = (
    { user = "foo"; pass = "abc"; },
    { user = "bar"; pass = "def"; }
);

custom_msg = (           // Below is an array - comment until EOL
    { lang = "en"; msg = [ "Error",  "Warning", "Notice"  ]; },
    { lang = "de"; msg = [ "Fehler", "Warnung", "Hinweis" ]; }
);
```

Currently, libcfg only provides access to scalar values and groups. The interface exposed to read configuration files consists of the following functions:

`cfg_init()`
: Initializes a new configuration object and reads in the configuration file specified.

`cfg_free()`
: Releases any resources associated with the previously obtained configuration object, which also includes any values obtained through earlier calls to `cfg_get_*` as the storage for those variables is internally managed by libconfig.

`cfg_reload()`
: Reloads a previously initialized configuration object. This function can be called if the used configuration file has been changed and new values should be read in by the application.

cfg_get_type()
Looks up the setting identified by key and setting path ident and stores the value in (*value). The path is a dot–concatenated sequence of names describing the location of a value. If the value is part of an array or a list, square brackets can be used to specify the exact position. Currently implemented values for type are int, int64 and char *.

cfg_last_err()
If cfg_init() or cfg_reload() return an error, this function can be used to obtain a textual description of the last error that has occurred in libconfig.

3.2 Event logging

As all of the applications introduced in the following chapters typically run as daemons in the background, hence without a terminal attached, they need a way to report errors to the user to allow appropriate actions to be taken. *liblogger* provides an easy to use interface that allows applications to log messages to multiple logging backends.

Before messages can be sent to a logging sink, an application has to call msg_sink_add() to add a specific logger to the logging chain. The parameters passed to this function are used to customize the logging sink.

type
Sink identifier. Supported types include LOGGER_FILE to log to an open file descriptor, LOGGER_SYSLOG for logging to the syslog daemon and LOGGER_RSYSLOG for logging to a remote syslog daemon over a network connection.

verbosity
Defines the verbosity level that is used in the msg_debug() to decide whether to discard the message or pass it on to the underlying logging backend.

mask
A bitmask specifying the events the sink is responsible for. With the mask parameter, an application can define multiple logging sinks, with each handling a different type of message. This feature comes in handy during software rollout. While warnings and debug output are just sent to the local syslog facility, errors can be forwarded to a centralized rsyslog server to accumulate and identify potential problems users may encounter. Sinks that should receive all log messages can specify the special value MSG_LVL_ALL.

...
The last set of parameters to the msg_sink_add() function is a list of sink–specific options, terminated by a special NULL sentinel value. The number of

arguments and their type are defined in the respective `init()` function of the sink.

The different logging sinks are organized in a double linked list protected by a read–write lock to allow concurrent threads accessing it. Each logging backend defines an identifier `enum logger` and a `struct logger_ops` as defined in `logger-backend.h` and shown in listing 3.2.

Snippet 3.2: Logger backend functions

```
1  struct logger_ops {
2      int (*open) (va_list ap);
3      void (*close) (void);
4      void (*log_info)(const char *fmt, va_list args);
5      void (*log_warn)(const char *fmt, va_list args);
6      void (*log_error)(const char *fmt, va_list args);
7      void (*log_fatal)(const char *fmt, va_list args);
8      void (*log_debug)(const char *fmt, va_list args);
9  };
```

The `open()` function is called by `msg_sink_add()` before adding it to the sink chain. If it returns a non–zero value, it will not be added to the chain and an error is returned to the caller instead. `close()` is invoked when `msg_sink_clear()` is called to clear all logging sinks or `msg_sink_del()` to remove a specific one. The rest of the functions contained in the structure are used to log messages with a particular log level. Upon invocation of the proper `msg_*` function, the message is passed to each logging backend that has previously set the appropriate log level in its `mask` parameter.

Note:

`log_fatal` is declared with `__attribute__((noreturn))`. It can be used in cases where an application suffers from a critical condition and needs to be terminated. It should be noted that when an application is dealing with sensitive information, it should not use `log_fatal` directly. Instead, a wrapper function should be provided that erases sensitive information before calling `msg_fatal()` to terminate itself.

file logging

When called with `type` set to `LOGGER_FILE`, the logging sink will expect the following parameter to be specified:

`char *ident`
: Identifier that makes up the first part of a log message. Can be set to the application's name for example.

`int is_fd`
: Species whether the third parameter is referring to an open file descriptor or a file pointer of type `FILE *`.

`(FILE * | int) out`
: File descriptor or file pointer that refers to the stream the log messages should be sent to.

The logging functions will then output a message with the following format:

```
[ident] level:  message
```

syslog logging

`LOGGER_SYSLOG` expects the following input for the `open()` function:

`char *ident`
: Identifier that makes up the first part of a log message. Can be set to the application's name for example.

`int options`
: Specifies a bitmask of zero or more options from `msg.h` combined by a logical OR operation.

`int facility`
: Selects the syslog facility. In addition, `/etc/rsyslog.conf` can be used to assign a certain facility to a specific log file to prevent multiple *liblogger* users from writing to the same log file.

It should be noted that the defined values in `msg.h` reflect the values defined by `syslog.h` so they can be passed as–is to the respective syslog functions.

rsyslog logging

The rsyslog logger can be used to send log messages generated by applications to be sent to a remote syslog server. Parameters for the `open()` function are as follows:

`char *ident`
: Identifier that makes up the first part of a log message. Can be set to the application's name for example.

`int options`
: Specifies a bitmask of zero or more options from `msg.h` combined by a logical OR operation.

int facility
: Selects the syslog facility. In addition, `/etc/rsyslog.conf` can be used to assign a certain facility to a specific log file to prevent multiple *liblogger* users from writing to the same log file.

char *hostname
: Specifies the hostname of the remote server.

char *port
: Specifies the port where the remote server is listening.

type
: Selects whether to use TCP or UDP when establishing a connection to the remote server.

Upon initialization, the `open()` function resolves the hostname and tries to establish a connection to `hostname:port`. When invoked through one of the `log_*` functions, the sink composes messages according to RFC 3164.

`PRI`
: A numeric value that combines the facility identifier and the message priority level into a single value. It is calculated using as PRI = 8 * facility + priority.

`HEADER`
: The first part of the header contains the current timestamp in the format of Mmm dd hh:mm:ss. The timestamp is followed by a whitespace and the hostname of the current machine.

`MSG`
: The rest of the message is composed of a string identifying the creator of the message and the user supplied message as passed during one of the `msg_*` functions. If the flag `MSG_FLAG_USERNAME` was set during `open()`, the name of the current user will be embedded in the log message.

A sample log message as generated by `logger-rsyslog` and sent to remote hosts is shown below. The priority level of 165 reflects the facility `LOCAL4` with a severity level of *notice* as defined in `syslog.h`.

```
<165>Jan 3 17:41:13  logger-rsyslog Could not fork
                     (from user foobar)
```

3.3 Generic object caches

Generic object caches can be used to efficiently store and retrieve items from a cache. As will be shown in later chapters, object caches play a central role for file and password management. Their use is rather simple and mostly limited to two functions:

an `insert()` function that inserts a given (key, value) pair into the cache and a `find()` function that finds a value given its key. As the cache should be usable by different applications that typically use different types of data, the goal was to allow an application to define its own data types and only use the cache as storage to keep track of pointers to the supplied data.

3.3.1 Key format

Any application using the cache should not be concerned about how the cache manages its data. Internally, the cache has various options to keep track of data supplied by clients which will be outlined briefly:

- Arrays
- Lists
- Search trees
- Hash tables

The object cache is implemented in `lib/ocache.c` and currently uses hash tables to manage keys. To be more precise, the library in use is *uthash*[2].As mentioned above, the cache should be usable by different applications with varying key format requirements, so it should be unaware of any specific data. For this purpose, an item as managed by the item cache consists only of data shown in listing 3.3.

Snippet 3.3: Object cache – Base structure

```
1 struct ocache_handle {
2     struct UT_hash_handle hh;
3 };
```

With a given (key, value) pair, the cache further needs a way to copy and compare keys, but also provide a way to the application to access the data it once stored into the cache. Therefore, each cache user has to define a structure according to listing 3.4 that provides necessary information for the cache in order to handle its items.

Snippet 3.4: Callback operations for item caches

```
1 typedef void (*oc_foreach_cb)(struct ocache_handle *handle, void *u);
2
3 struct ocache_operations {
4     void *(*get_key_desc)(struct ocache_handle *handle, unsigned int *k_len);
5 };
```

[2]A very simple to use hash table implementation for C. It only consists of one header file and is implemented using only preprocessor macros. More information, documentation and examples are available at https://troydhanson.github.io/uthash/

If the application has implemented the `get_key_desc` callback, it can make use of the public API located in `include/cfs/ocache.h`.

The typical workflow starts with creating a new cache using `ocache_create()` which returns a handle to an object cache. The application then has access to a number of functions:

`ocache_insert(struct ocache *cache, struct ocache_handle *handle)`
: Adds an item to the cache. Uses the `get_key_desc` to get a pointer to the key, but also the length of the key. If an item with the same key was already contained in the cache, it will be removed. The given key is inserted and the previously contained object is returned to the caller. Based on the return value, the caller can determine if that has happened.

`ocache_tryinsert(struct ocache *cache, struct ocache_handle, *handle)`
: Behaves exactly like `ocache_insert`, except that if an item with the same key was already contained in the cache, no action is performed and an error is returned to the caller with errno being set to `EEXIST`.

`ocache_remove(struct ocache *cache, struct ocache_handle *handle)`
: Removes a given handle from the cache.

`ocache_for_each(struct ocache *cache, oc_foreach_cb cb, void *u)`
: Calls the given callback function `cb` along with arbitrary information supplied in `u` for each cached object. This function is deletion safe, therefore the callback function `cb` is allowed to remove an item from the cache. The key value must not be modified. This function for instance can be used to remove all *expired* values from the cache with one function call.

CHAPTER 4

Development of a generic keystore interface

Based on the findings as presented in chapter 2.5 ”Secure storage“, including key management directly in the file system is not expedient. From a philosophical standpoint, this is justifiable as it violates the fundamental principle of UNIX: *Keep it short and simple*. [Ray03] goes one step further and introduces a series of design rules that are vital especially for new software as they allow easier code maintenance and ease the implementation of enhancements:

Rule of Modularity
Simple components should represent the application baseline. Well–defined interfaces act as connectors between the components. This allows changes to remain isolated which results in easier components upgrades / replacements.

Rule of Composition
Applications should not be built as a complicated monolithic block, but instead split among multiple applications with a simple, textual, stream-oriented data format.

Rule of Separation
This rule is today’s equivalent of the Model–View–Controller pattern. It suggests that a piece of software is easier to maintain, enhance, and test if it is decoupled from other parts of the software. interfaces from engines.”

Rule of Simplicity
It is preferable to use a simple approach to solve complex problems instead of over–complicating the implementation. This makes it easier for other people to read, maintain and extend the code and lowers the chance of introducing logical errors and software bugs.

When looking at the typical Linux desktop it becomes quite obvious why these rules have a right to exist. The kernel itself is divided into multiple subsystems with distinct interfaces. Changes in how processes are handled (for example replacing the scheduler) do not affect I/O operations. Likewise, loadable kernel modules can extend the kernel's functionality without interfering with any subsystems. The "Rule of Simplicity" becomes obvious when looking at Linux' source code. Even without knowledge of the "big picture" one can tell what a function does almost immediately as they are short and simple.

Due to the success of Linux, following the aforementioned rules should result in a simple yet easy to understand, maintain and enhance prototype. The key management was separated from the file system introduced in the next chapter and implemented in an abstract base class. The interface to specific keystores like gnome–keyring or KWallet will be provided by connectors which can be built on a per–host basis. The base class `PasswordStore` acts as an abstract interface, providing only three functions related to key material:

`find`
: Find a key in the keystore given its identifier

`add` Add a (identifier, key) pair to the keystore

`batchAdd`
: Add a collection of (identifier, key) pairs to the keystore

In the given implementation, connectors for gnome–keyring and KWallet are present in classes `PasswordStoreGnome` and `PasswordStoreKde`. By default, these stores operate on a collection (gnome–keyring) / wallet (KWallet) named according to the value of `PASSWORD_STORE_NAME`, defined in `PasswordStore.hpp`, with a default value of "`keyd`".

4.1 Keystore Interface

`PasswordStoreGnome` uses the interface provided by gnome–keyring as described in `gnome-keyring.h`. On instantiation, the constructor `PasswordStoreGnome()` verifies that a communication with gnome–keyring is possible using `gnome_keyring_is_available()`. This is important as it would not make sense to keep retrying querying the daemon if it is not present on the system or suffers from a critical condition itself. Subclasses of `PasswordStore` therefore provide an `init()` function with return type `InitResult` to reflect this circumstance. Possible values include `INIT_TEMP_FAIL` and `INIT_PERM_FAIL` to indicate temporary or permanent failure. In either case, the calling application can then take appropriate actions – permanent failure would typically result in application termination, whereas temporary errors would just cause the application to retry the request after a certain amount of time. In addition, the `Init()`

function also verifies the presence of the default store that is used for querying and adding keys. If no such store exists, it will try to create it.

As the name suggests, the `find()` function uses a given Globally Unique Identifier (GUID) to find the corresponding key entry. A *key entry* however does not necessarily imply a single password string in the keystore. Instead, gnome–keyring uses attribute lists to make up the entries. Supported attributes are strings and unsigned integer types. Although not currently in use, these attributes could be used to store additional information about a key datum. For security reasons, this might include an age timestamp to enforce periodical refreshment of certain keys which otherwise has to be stored and synchronized in a different location.

Before keys can be looked up in the keystore, one needs to add key material to the store. Ignoring the fact of how this material is obtained for now, the `Add()` function adds a single key entry to the keystore using `gnome_keyring_item_create()`. If multiple keys have to be added at once, one can use `BatchAdd()` which expects a key bundle as input parameter and adds each key of this bundle to the store using `Add()`.

Find and add functions use the return type `QueryResult` to indicate whether the operation was successful (`OK`), failed (`FAIL`), or did not return any result (`NOENT`) in case of `Find()`.

`PasswordStoreKde` provides exactly the same functionality, but the KDE environment requires the application to provide additional information about itself in an `KApplication` object. This information will be displayed when the user receives a prompt to set a password for a newly created wallet or an existing one needs to be unlocked. The information is also helpful to the user as it helps to categorize the request, so the user knows, what is going on and where the request is coming from and why.

Note:

Subclasses of `PasswordStore` use a special class `SecureString` to handle sensitive information. This class is similar to the standard string class and internally uses a combination of such a `std::string` object and the Linux system call `mlock()` to protect the information stored within. Although one could argue that `mlock()` is not secure as is does not guarantee that the data will not be swapped out to disk in this case, it is neglectable here as the focus is on the secure removal of the data after it is of no use anymore. The functionally provided by `mlock()` addresses swapping data out to disk, but since the keystore application is of a short lifetime, the Linux page replacement algorithm will not swap the page to disk anyway.

Once the password has been retrieved from the store, the questions of where it came from and how it is returned to the application still remains. As the goal of this paper was to investigate multiple methods of encryption, the source of the request could either originate from user–space or kernel–space and even there, multiple file

systems or applications could be implemented and thus be the origin of the request. For example, when FUSE volumes are mounted during start–up using `/etc/fstab` and the mount option `allow_other`[1] was listed in the mount options, the permission handling of FUSE will act in a slightly different way: user root and other users are given access according to the permissions associated with the files in that directory, therefore honoring UNIX file permissions. As a result, the keystore–interface should be as flexible as possible to handle such situations.

As a first step, the keystore–interface was implemented as a daemon process. This allows arbitrary request sources to be handled properly by providing a common entry point to the requester's keystore. As mentioned above, the source of the request is arbitrary and therefore assumed unknown. To handle different mount options, the daemon needs to be able to react to different requests appropriately. Internally, a request is a C structure as shown in listing 4.1.

Snippet 4.1: Key daemon Lookup request

```
1 struct lookup_request {
2     char     type;     /* 'P' or 'G' */
3     uint32_t len;      /* Length of the data field */
4     uid_t    uid;      /* UID of the requesting user */
5     gid_t    gid;      /* GID of the requesting user */
6     char     data[];   /* Data: Path ('G') or GUID ('G') */
7 }
```

The `type` field governs the nature of the request and indicates the content of `data`. Its value is one of

'P' `data` field contains a path or

'G' `data` field contains a GUID

Different type values are the result of different file operations. A `creat` system call typically causes a `type` value of `'P'` as the GUID for a given path has to be resolved before a password can be obtained for it, whereas opening a file results in a type `'G'` look–up as an existing GUID is read from the file header and a key needs to be found for it. These settings are typically configured in the SafeGuard Enterprise Management Console. The client received those mappings ("policies") and applies it when a file needs to be created.

The `uid` and `gid` fields are used by the requesting application to describe the sender. In the case where a normal user mounts and uses the file system, the values are identical to the `uid` and `gid` of that user, but if the volume was mounted with e.g. FUSE and `allow_other`, the values reflect the real identity of the requester, not the

[1] `man 8 mount.fuse` states that "this option overrides the security measure restricting file access to the user mounting the file system. So all users (including root) can access files. This option is by default only allowed to root"

application running the file system. Access to the file system however is always conducted using the effective user identity of the mount process. As a result, when using system calls that create files and folders (`creat`, `symlink`, and `mkdir`), their owner will always reflect the effective user and group identities of the mount process. The user under which the file system is mounted therefore needs access to all files that should be accessible to other users through the mountpoint. However running it as non-root user makes it impossible for the mount process to change owners and permissions on newly created files. New files will therefore always belong to the user running the file system.

As a consequence, the keystore–interface needs to be able to communicate across different session as the mount process may be running under user A whereas access is performed by user B. In such cases the key lookup becomes quite complex. Initially, the daemon has to be started by either a user or at start–up. When an application starts, Linux assigns the effective user identifier (UID) which will then determine access permissions on file systems.

Linux users and groups

Each process under Linux has 3 pairs of (user, group) IDs assigned to it. When the process starts, the *real* user and group are set to the user who executed it. If the program uses `setuid`, the *effective* user and group are set according to the file's owner, otherwise they are equal to the real user and group IDs. *Saved* UIDs are used for set–uid programs to temporarily drop privileges to do some work and then later regain elevated privileges by re–using the values from the saved UID and GID fields.
Since `setuid` applications run with privileges of their owner, not the user who started them. This could potentially be harmful, so their use should be avoided or restricted to applications that can easily be security–audited. Lastly, file system UID and GID are used for file system access checks. They are usually equal to the effective UID and GID. Changing these values requires superuser privileges though.

As a result, the keystore–interface faces the problem of the user accessing files is not necessarily the user who started the key–store application. This raises the question of how the keystore–interface is able to access the right store for a given user. As it runs as a background process with limited permissions and capabilities, it does not have the ability to impersonate the accessing user's home directory in order to open the wallet and retrieve the requested keys. A first solution to this problem would be to increase the permissions of the daemon process, but this violates the principle of least privilege and causes the daemon to run with root privileges at all times. As mentioned earlier, running the daemon `setuid` root would also solve the problem, but could potentially cause security vulnerabilities, especially when the complexity of the application increases. On the other hand, `setuid` programs typically run with

low privileges to do most of the work and increase / drop their privileges as needed. This approach is exactly what is needed here, but will be solved in a different, more flexible way. The keystore–interface is simply split into two parts: a daemon, running with lower privileges to resolve and map policies to the requests and a second daemon that received these requests and performs actual look–ups. From a security point of view, this setup is optimal as all components run with least privileges, hence lowering the possibilities of exploitable vulnerabilities.

4.2 Architecture

Figure 4.1 shows the resulting architecture for the key–store interface. What seems to be a rather complex solution for the aforementioned problems, simultaneously solves all problems in a very clean and efficient way.

The component *keyd* is the main entry point for any given application requesting key material. If necessary, it first performs a path–to–guid mapping for a given "`P`" request. It then passes the request on to *keyd-runas* which performs the actual lookup in either KWallet or gnome–keyring. As the name suggests, *keyd-runas* will impersonate the user which is trying to access his keystore. To be able to do that, *keyd-runas* requires root privileges. As mentioned before, this would be a violation of the principle of least privilege, but *keyd-runas* only performs very little work until it sets its privileges to the user and group ID as requested.

When a request is received, *keyd* offers two possibilities of how to resolve the user sending the request:

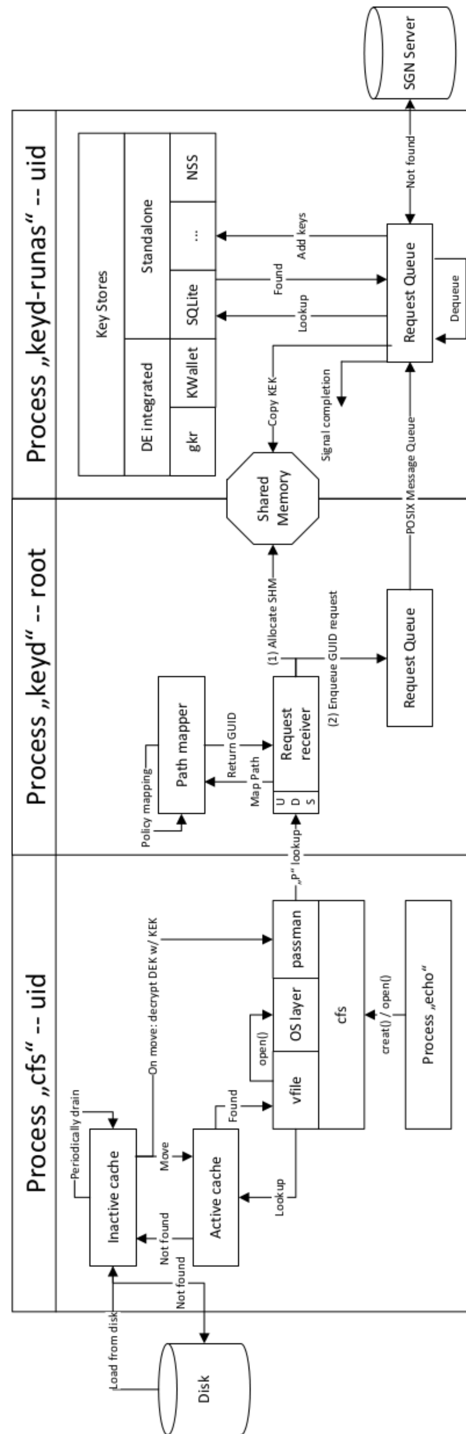
As resolved

keyd uses the information provided by the kernel to determine the connecting users. This approach is recommended due to the trustworthiness of the kernel. To resolve the user ID of the socket connection, *keyd* uses `getsockopt()` with the option `SO_PEERCRED` on socket level `SOL_SOCKET`. When successful, this function call fills a structure of type `struct ucred` as shown in listing 4.2. *keyd* then checks if the credentials match the IDs specified in the request. Non–matching values are an indicator of malicious requests and therefore discarded.

As sent

keyd uses the values the client has sent. This is of course not very trustworthy as the client could send arbitrary values, but sometimes this behavior is required as the sender could not be determined correctly by the kernel. This is the case when FUSE is mounted with `allow_other`. Even though user Y is accessing the data, the FUSE process is running under UID X. Using aforementioned function `getsockopt()`, the returned structure would therefore return the ID of user X. The discrepancy of the values in `struct ucred` and the IDs in the request would therefore always cause the data to be discarded. If FUSE is operated in that way,

Figure 4.1: Keystore–interface architecture



the configuration directive `use_ucred` in section `socket` in the configuration file of *keyd* should be set to 0.

Snippet 4.2: struct ucred

```
1 struct ucred {
2     pid_t pid;  /* PID of sending process */
3     uid_t uid;  /* UID of sending process */
4     gid_t gid;  /* GID of sending process */
5 };
```

When the request has been fully received, *keyd* performs some initial sanity checks and invokes a handler for the given type of request. A request handler points to a function of the following type:

```
int (*req_handler)(lookup_request *, lookup_result **)
```

Currently, `secret_for_guid()` and `secret_for_type()` are implemented to handle "`G`" and "`P`" request. Since these functions have access to the request data, the `secret_for_type()` handler could easily be extended to perform source–specific operations and support further types of sources. Eventually, all handlers call the `do_key_lookup()` function to initiate the key lookup through *keyd-runas*.

4.3 Keystore–Interface message passing

To limit the impact of software bugs and exploitability, *keyd* is running with limited privileges. When clients connect to the socket, the only information *keyd* can obtain about the client are its credentials. Normally, daemons make use of system calls like `setuid()` and `seteuid()` to impersonate a particular user, perform a specific task, report the result, and then exit. In this case however, this is not possible due to the restricted privileges of the *keyd* user. Therefore, the software has been split into two parts: *keyd* to perform the policy mapping and *keyd–runas* to query the desktop environment's native key store to obtain key material.

In this scenario, messages need to be exchanged between the components at four different locations:

client → keyd interface
The main entry point for the client. *keyd* expects messages in a particular format as described by listing 4.1.

keyd → keyd–runas interface
When messages have been received, parsed and found valid, *keyd* will submit the query to *keyd–runas* using a System V message queue.

keyd–runas → keyd interface

Due to the fact that *keyd* and *keyd–runas* are two different processes, results need to be communicated back to *keyd*. The IPC mechanisms used to perform the data exchange are System V shared message segments for the data and semaphore sets to maintain synchronization between the processes.

keyd → client interface

When *keyd–runas* signaled completion, *keyd* needs to send the result to the client. This happens over the same socket connection as in the client → keyd interface.

When *keyd* is started, it performs some initialization tasks that also include the creation of the message queue. It then submits a simple helo packet as shown in listing 4.3.

Snippet 4.3: keyd helo message

```
1  struct keystore_helo {
2      long  mtype;
3      pid_t pid;
4  };
```

The reason for sending this message is to force clients to not directly "speak" to *keyd–runas* as it is running with root privileges which could be considered a security risk if the interface is accessible to other applications. The `mtype` field of the helo message has to be set to `KEYSTORE_MTYPE_HELO`. The field `pid` is simply set to the return value of `getpid()`. As long as *keyd–runas* has not received any helo message, the submission of any messages other than of type helo will result in application termination due to security considerations. Otherwise, for each received message, the sender's PID will be checked against the saved helo PID. A mismatch also results in application termination.

Note:

The `pid` field of `struct keystore_helo` has to be set to the PID of *keyd* as *keyd–runas* will check this value against the `msg_lspid` field of `struct msqid_ds` as obtained through `msgctl()` for option `IPC_STAT`

After the initial helo has been received, lookup requests as described in listing 4.4 can be submitted to *keyd–runas*.

Snippet 4.4: keyd lookup message

```
1  struct keystore_request {
2      long   mtype;        /* Message type */
3      uid_t  uid;          /* Requesting user's UID */
4      gid_t  gid;          /* Requesting user's GID */
5      key_t  sem_key;      /* Key used to attach the semaphore for signaling */
6      key_t  shm_key;      /* Key used to attach the shared memory segment */
7      size_t shm_size;     /* Size of the allocated shared memory segment */
```

```
8     size_t guid_len;    /* Length of the guid field */
9     char   guid[];      /* GUID to lookup in the keystore */
10 };
```

Before sending such a keystore_request to *keyd–runas*, *keyd* has already performed some preliminary work. This is done to concentrate complex operations in code executed with lower privileges and to detect and inform the client as soon as possible about errors.

When generating a keystore_request, *keyd* sets the value of mtype to KEYSTORE_-MTYPE_REQ to indicate that this is a lookup request. The uid and gid fields reflect the identity of the user that performed a file system operation that requires key material. sem_key and shm_key are the return values of the ftok() function. These values will be used by *keyd–runas* to obtain the same semaphore and shared memory segment as *keyd* uses. After data has been placed in the shared memory of size shm_size, the semaphore returned by semget(sem_key) will be used by *keyd–runas* to signal completion to *keyd*, which was put in blocked state by the operating system when it invoked the semop() call.

In terms of pseudo–code, the interaction between *keyd* and *keyd–runas* is handled as shown in table 4.1. It should be noted that *keyd* uses the timed variant of semop(). This is useful, because if errors in *keyd–runas* occur, the point where the semaphore is signaled is never reached, so *keyd* would not be able to drop the connection by itself, as it is blocked by the operating system. The semtimedop() makes sure that *keyd* will be awakened by the kernel after a configurable period of time. One can use the configuration setting runas_timeout in section keyd of the keyd configuration file to control the timeout.

Table 4.1: Keyd / Keyd–runas interaction – pseudo code

keyd	keyd–runas
msgsnd(req)	
	msgrcv(req)
semtimedop() (wait)	
	keystore lookup shmat(shm_key) copy result to shared memory shmdt(shm_key) semop() (post) exit
shmat(shm_key) send result to client IPC cleanup	

4.4 External keyring sources

A question left unanswered so far is: where do key rings initially come from? In case the application has just been newly installed on a host, `keyd` is able to interact with keystores and create collections and wallets, but the actual key material is not yet stored on the host. Moreover, the origin of a lookup request could also be a user opening a file that was encrypted on a different host with a key the user does not have on the machine he is currently logged in. Least likely the reason for a not found key could also be a damaged KWallet or GNOME keyring.

In any case, the `Find()` function of a keystore connector returns the status code `NOENT`. If this status code is encountered, *keyd* uses the `keystore_fetch_external()` function to connect to a key server and receive a key bundle owned by the requesting user. An external server is described by a structure as shown in listing 4.5.

Snippet 4.5: Description of external key servers

```
1 struct trusted_host {
2     const char *hostname;                /* DNS name or IP address */
3     const int   port;                    /* TCP port to connect to */
4     const char *keyring_path;            /* Remote location of the keyring */
5     const char *cacert_loc;              /* Location of a SSL certificate chain
6                                             for certificate validation */
7     const char fingerprint[(2*FP_LEN) + 1]; /* SSL fingerprint */
8     const char *name;                    /* Pretty name for logging purposes */
9 };
```

The first two members are self–explanatory, and simply indicate the DNS hostname or IP address and port where the key ring should be retrieved from. The `name` field is used in log messages and denotes the "pretty name" of the respective key server. The `keyring_path` field specifies a path on the server where the key ring can be found. During development, this string was set to a PHP script[2] that will redirect the user to an Extensible Markup Language (XML) document containing a list of keys. The XML document uses a rather simple structure as it is only used for testing purposes. Its structure is denoted in listing 4.6. In production environments, this data is typically sent by the SafeGuard Enterprise server in a different format.

Snippet 4.6: XML structure for key rings

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <sg:keyring xmlns:sg="http://www.olorenz.org/KeyRing">
3     <sg:user-record>
4         <sg:name>John Doe</sg:name>
5     </sg:user-record>
6     <sg:key-record>
7         <sg:guid>9b69b8d3c1dfaeba2d53164c361b72be</sg:guid>
```

[2]PHP or *PHP: Hypertext Preprocessor* is a server–side scripting language mostly used for web development.

```
8           <sg:data>yjx2XvNwqFsPVQLBekNDTy3f</sg:data>
9       </sg:key-record>
10      <sg:key-record>
11          ...
12  </sg:keyring>
```

The owner of the key ring is currently used only for logging purposes, but it would also be possible to verify the returned string against the value of the `pw_gecos` field which is returned by `getpwnam()` and `getpwuid()`. This would most likely require the use of external user databases like LDAP to keep all information synchronized.

Ultimately, *keyd* uses *libcurl* to retrieve the keyring from the server. *libcurl* is a widely used library supporting vast amounts of protocols and features[3] as it would otherwise be required to implement HTTPS capabilities. Upon connection establishment, *keyd* uses curl's `curl_easy_getinfo()` function with the option `CURLINFO_-CERTINFO` to check if the server certificate matches the expected value specified in the `struct trusted_host`. Mismatches are a fatal error and lead to *keyd* giving up on the key ring retrieval, even though further key server are specified in the `trusted_hosts` array.

When the data has been received from the server, *keyd* uses *libxml* to parse the data into a valid `struct key_bundle`. The underlying C structures are presented in listing 4.7.

Snippet 4.7: Keyd key structures

```
1   struct key {
2       char *guid;              /* Global Unique Identifier */
3       char *secret;            /* The actual password */
4       struct list_head next;   /* Next password in list */
5   };
6
7   struct key_bundle {
8       char *owner_name;        /* Name of the owner */
9       struct list_head keys;   /* List of struct key */
10  };
```

A bundle is simply a linked–list of keys with an additional owner field. The key bundle can then be passed to the `BatchAdd()` function to add all contained keys to the keystore. The previous `Find()` call will be restarted to check if the requested key was contained in the received key bundle.

[3] A comprehensive list of features is available under http://curl.haxx.se/docs/features.html

CHAPTER 5

Development of user–space encryption using FUSE

This chapter will elaborate on the implementation details of the FUSE file system *cfs*. The result will be a user–mountable file system fully compatible with the already existing SafeGuard Enterprise file format as described in tables A.1, A.2, and A.3.

5.1 Layout

FUSE requires file systems to register a structure containing pointers to functions that will be called whenever events occur in the exposed file system. Therefore they act as entry points to the file system. Additionally, these callback functions are similar to the traditional system calls and also named accordingly. During the file system's initialization, the structure containing the callbacks is passed to FUSE in its `fuse_main()` function.

Due to the complexity of the file system, it has been split into the following subsystems. Each of them is explained in greater details in this chapter, concluding with an overview of how the interfaces between them are designed.

File manager
The file manager provides functionality to access the underlying file system and also maintains a cache of frequently used files to speed up meta data deserialization.

Password manager
This subsystem is primarily used to provide access to key materials previously received from a key store provider or an external key server.

Memory management
The memory management functionality allows efficient memory management

and object caching for frequently used objects. This chapter will elaborate why it is needed and how the file system benefits from its caching capabilities.

Crypto core
: As the name suggest, this facility is a high–level interface to the OpenSSL crypto functions used to encrypt and decrypt data

Anti–mountpoint bypassing
: Whenever the *original* folder has to stay intact, it is possible that the user saves his files to the wrong directory. The anti–mountpoint bypass daemon ensures that files are stored to the correct location and therefore prevent information exposure and policy violations.

The various sub–systems are initialized when FUSE starts up. The file system operations that are passed to the `fuse_main()` function contain a pointer to an `init()` function that will take care of this task. Even though the initialization task could be done in a different location, FUSE advises to do it in the `init()` function as threads that would be started beforehand would be killed when the FUSE file system forks and daemonizes itself[1]. When the file system has started up, it waits for incoming tasks from the FUSE kernel module in the `fuse_session_loop()` function.

A FUSE file system can be operated with only a few functions inside the `struct fuse_operations` being present, which in turn limits the capabilities of the file system. However, this approach can be used to iteratively extend the implementation. Starting from the implementation of e.g. `readdir()` and `open()` calls, one can successively add needed functionality until all desired features are fully functional. It should be noted that not all of the callbacks are actually needed to provide all needed functionality. Read-only file systems for instance do not need a `write` method. However, the file system resulting from this research will be used in many different situations so it is mandatory to implement all 47 callbacks provided by the FUSE library.

FUSE path names

Functions provided in the `struct fuse_operations` usually have one or more `char *path` parameters. These parameters are filled by FUSE using the relative path of a file or directory. It is therefore required to translate this path to an absolute path on the underlying file system by prepending the root directory the FUSE mountpoint is responsible for. To accomplish the translation, one has to implement a function that simply concatenates the two strings, and has to call this function each time one of the FUSE callbacks is executed. A cleaner solution can be achieved by loading the *subdir* module before calling `fuse_main()` which results in all FUSE callbacks already receiving the absolute file system paths. The prototype will use latter approach.

[1] This would not be a problem if the file system is launched with the foreground option (-f) set

One of the most crucial functions to FUSE file systems is the `getattr()` function. FUSE uses it to obtain meta–information about files. Therefore, this function is called quite frequently and should therefore be short and efficient.

5.2 File management sub–system

The purpose of a file system is to operate on files and give users access to stored information. Cryptographic file systems enhance the access part by introducing an additional, mostly transparent layer of encryption between the user and the storage backend. In case of the FUSE file system implementation, the *file manager* is used to provide all necessary functions to allow a user to access his data and perform day to day tasks on files. The API offers functions similar to the traditional system calls like `creat`, `rename`, or `unlink`. The reason why this functionality is outsourced to a sub–module becomes evident when one reflects about how files are usually used on a computer. Typically, users make use of graphical file managers to maintain their files. These programs provide lots of information to users, e.g. file names, file size, date of last access and so on. When a directory is opened, the file manager iterates over all files in the folder and typically uses the `stat()` system call to obtain this information from the operating system, more specifically the underlying file system. In kernel–space, files are represented by a structure defined in `fs.h`: `struct inode`. This structure holds all information the kernel needs in order to perform requested operations on the file itself. Internally, the kernel uses a hash table to provide caching functionality, thus increasing the performance of the system. In case of the graphical file manager, the kernel makes use of that cache when the user navigates through the folder structure as files and folders entries must be displayed repeatedly. The kernel does not request the inode information from the underlying file system, instead, the cached data is returned to the user. A file manager can then use the information obtained through the `stat()` system call to populate its user interface.

When it comes to encrypted files however, the information as cached by the kernel might not be of much use to the user. File–name encryption for example requires the file system implementation to perform a transformation (e.g. decryption) each time the folder is opened and the file manager needs to display the file's name. In addition, opening a file multiple times would increase memory consumption of the file system itself and also the amount of delay the user experiences when the file manager has to display information about the file to the user as necessary transformations have to be performed each time.

As a consequence, the file management sub–system is built around two caches. These are used to store meta–information about opened files and allow the file manager to quickly return information when requested by FUSE. As will be shown later, the process of deserializing a file header is a complex and tedious process, so both caches are used to minimize access time by keeping deserialized data in memory

while the files are in use. The *active cache* contains items that are currently in use by users. That is, when a process has already called `open()` and is now in the process of reading or writing data, but has not yet called `close()`. After the file has been closed, it will be moved to the *inactive cache* and remains there for a configurable amount of time. During this step, all sensitive information is removed from memory, but data structures are left intact to allow a fast reconstruction of necessary crypto data when the file is used again. This could happen after the user used save and close on a text document in an editor, but keeps navigating through his files in a file manager.

Both the active and the inactive cache make use of the generic item cache as introduced in section 3.3 "Generic object caches". Commonly, the path of a given file would be sufficient to use for a cache key, as it is unique and already provided by FUSE when actions on files need to be performed. *cfs* however uses cache keys that are comprised of the device identifier that contains the file and the inode number of the file itself as obtained by the `stat()` system call as shown in listing 5.1.

Snippet 5.1: File manager cache key

```
1 struct fileman_cache_key {
2     dev_t st_dev;       /* File device number */
3     ino_t st_ino;       /* File inode number */
4 };
```

While it would seem more natural to use the path of a file as key for the cache item, using device and inode numbers offers two important advantages:

Lower memory consumption
: Regardless of the path of a given file, cache keys use a constant amount of memory of 12 bytes on 32–bit platforms and 16 bytes on 64–bit platforms.

Simpler file handling
: When a file is renamed, its device and inode numbers do not change. As a result, the cache key does not change and no operation has to be performed in the cache itself. However, auxiliary information of the file needs to be updated. The same holds true if directories are renamed, although refreshing cached items is a more complex task in such a case as the change needs to be propagated throughout all cached data. This is due to the fact that the file has path information associated required for password lookups.

The data structure stored along with a key and obtainable by `item_cache_lookup()` is shown in listing 5.2. It contains housekeeping information about the underlying file in a first layer in `struct vfile` and cryptographic information in `struct ocbfile`.

Snippet 5.2: File manager cache item

```
1 struct vfile {
2     struct ocbfile file;
3     atomic_t refcnt;
```

```
4       time_t last_used;
5       atomic_t pending_unlink;
6       int no_recache;
7       pthread_rwlock_t rwlock;
8       int valid;
9   };
```

The file manager manages these vfile objects in one of the two caches. While they are in use, the vfile object is contained in the active cache, with the `refcnt` field being > 1 to prevent deletion. If the file is closed, the `refcnt` reaches zero and the vfile object is moved from the active to the inactive cache. The object resides there until a configurable amount of time has elapsed. This expiration timer is used to shorten the amount of time needed to re–open a file. An `open()` call on a file goes through the following steps:

1. Generate a cache key given the file's path.
2. Query the active cache. If a file could be found, return a handle to it.
3. Query the inactive cache. If a file could be found, move it to the active cache and return a handle to it.
4. Load the file header from disk and attempt to deserialize it. If successful, insert it into the active cache and return a handle to it.

Particularly cache hits in step 2 and 3 result in a significant decrease of the latency of the `open()` operation, as lookups are typically fast operations on caches. In case the file has to be loaded from disk, prompting the user to unlock his keystore (if it is not cached anymore) might add further delay.

Moreover, two distinct caches are useful if the file manager caches contain a large number of files, because the scan for expired objects would take additional time. During the scan, the cache has to be protected by a mutex to avoid inconsistent data structures. The locking adds additional delay to the lookup operation as the `mutex_lock()` operation blocks until the mutex is unlocked by the scanning thread.

5.2.1 Serialization and Deserialization

The file manager operates on a layer close to the system calls `creat()`, `open()`, and `close()`. When the file system is mounted, both caches are empty and operations requested by users result in one of the three aforementioned system calls to satisfy the request.

Creating encrypted files

When a file needs to be created, the file manager uses `vfile_get_new()` to create a new vfile object. By the time of creation, the only associated data with the object is

the path of the underlying file. The file manager then uses this path information to generate a cache key comprised of the device ID and inode number of the file and inserts it into the active cache. This immediately allows concurrent access to the file as it is visible to other users by the time the insertion completes and the cache mutex is released. Otherwise, the cache mutex needs to be held until the vfile object has been fully constructed; an approach which is highly discouraged as it could take a long time to retrieve the key material from the user's keystore, perform necessary key expansion (e.g. PBKDF), and write the file encryption header to disk.

The file header used by the file system is identical to the header used by *SafeGuard Enterprise*, hence providing full compatibility between the software products. An interface to read and write headers is located in `ocbfeheader.h`. It strictly follows the file header specification as outlined in appendix A. For file creation, `ocbfeheader_write()` can be used to write the header content to disk.

Endianness

When dealing with data types spanning more than one byte in memory, their representation can be of two different kinds: little endian and big endian. Internally, memory is managed in cells, with each being accessible by an index called *address*. If an application wants to store a 32–bit integer value, four consecutive cells will be occupied. On little endian platforms like Intel processors, the most significant byte is stored on the lowest address, whereas on big endian platforms like Motorola, the least signification byte will be assigned to the lowest address. In terms of C programming, the problem becomes evident when considering the following example:

```
uint32_t i = 123;              /* equal to 0x7B */
uint32_t i_le = htole32(i);    /* convert to LE */
uint32_t i_be = htobe32(i);    /* convert to BE */
```

When printing the values, `i` will be printed as *123*. On the Intel platform, `i_le` is equal to `i`, but `i_be` evaluates to a large number for the simple reason that `0x7B` is interpreted as the most significant byte out of the four, thus leading to a 32–bit integer of value `0x7B000000`.

For each multi–byte value, the function uses the `TO_HD_VAL` macro along with the bit–size of the number[2]. Depending on the architecture the software is built on, the compiler might remove the calls due to optimization steps. Nevertheless, the output of this function is important because it ensures a consistent file format across different hardware platforms. Furthermore, after passing through this layer, the software does not have to deal with endian issues as it is ensured that data is in *host byte order* before the function is called and in *little endian* afterward.

[2]When reading the header, one can use `FROM_HD_VAL`, which performs the inverse operation

At this point, the header is ready to be written to disk. Two steps are required to do so:

1. `header_write_prep()` ensures that there is enough space available on the disk by using `posix_fallocate()`. This function may expands the underlying file and guarantees that subsequent writes do not fail due to the lack of disk space.

2. `header_write_data()` writes the header that consists of the following main information to disk. For a detailed header description, refer to appendix A.

    a) Copyright information

    b) Generic meta–information. Data includes header version, magic value, block size for cipher operations.

    c) Key Storage Area that contains initialization vector and key count.

    d) Cryptographic information about the file. This includes the encrypted data encryption key, its identifier and human readable name as well as a description about the used cipher algorithm to allow decryption of the used Data Encryption Key when opening the file.

After the header has been written to disk, the application returns to the file manager. At this point the active cache contains a new entry. All necessary information is available to perform any operation the user might requests. If an error has been encountered during this process, the underlying file is deleted and an error is returned to the kernel. Otherwise, the application returns a unique handle to the FUSE kernel module which in turn associates this information with the in–kernel inode structure of the file. The calling application will receive an ordinary file descriptor as usually.

Opening an encrypted file

First of all it should be noted that the following description only applies to files that were not found in the active or inactive cache. This happens if a file has never been opened before or its cached meta–information has expired due to inactivity.

Opening an already existing file requires deserialization of the file's header. First, the raw file header needs to be read into a buffer, which is done by calling `ocbfeheader_read()`. This function tries to read up to `FE_HEADER_SIZE` bytes (4096) starting at offset `FE_HEADER_OFFSET` (0) within the file. The term *up to* is important here, because pre–existing files could be smaller than the header size and would be impossible to open if a return value less than `FE_HEADER_SIZE` of the `read()` system call would be treated as an error. For files larger than `FE_HEADER_SIZE`, `ocbfeheader_read()` performs endian correction and verifies magic values contained in the header to make sure that the data read into the buffer belongs to an encrypted file. In any case other than the successful verification of the buffer, the error number is set to `ENOMSG` to indicate that data of the desired type could not be parsed.

Once returned to the file manager it inspects the return value of the read call which governs the next steps. If the error number is ENOMSG, the file is treated as plaintext. All operations on such a file are simply relayed to the underlying file system without any modification. This behavior can be observed when opening unencrypted files that have not been created by this software, but copied into the root directory directly. Otherwise, the file manager evaluates the `headerVersion` field in the base structure of the header and tries to find a parser for this header version using `ocbfile_get_parser()`. If a parser was available, its `new()` function will be invoked to deserialize the header. If an error was encountered during this process, it is returned to the kernel which in turn forwards the error number to the calling application. Otherwise, the application returns to the file manager which inserts it into the cache and returns a unique handle to the kernel.

5.3 Password management

A central part of file handling with encrypted files is the management of sensitive information for cryptographic operations like symmetric keys. When new files are created, necessary encryption keys and initialization vectors need to be generated. Opening files results in the deserialization and decryption of data. In any case, the secure storage of sensitive information must be ensured. This is important as file systems are long–running processes, which might results in the operating system swapping unneeded pages out to disk, which in turn may lead to information exposure.

As a first step, one can use memory locking functions like `mlock()` to lock pages in memory and prevent the operating system from swapping out affected pages to disk. `mlock()` has two important properties that must be taken into account when dealing with sensitive data:

Unit size
: Memory affected by `mlock()` and `mlockall()` is divided in units of size `PAGE_SIZE`, typically 4 kB on modern processors when not operating in huge or large page size mode.

Stackability
: Calls to `mlock()` and `mlockall()` do not stack, which causes a page that has been locked several times to be unlocked by a single call to `munlock()`.

Both properties interfere with secure storage when considering the life time of the file system. Suppose memory for an AES key has to be allocated. In C, one would use `malloc(32)` to obtain the memory for a 256–bit key from the operating system. If additional keys need to be stored, the procedure is repeated. The return value of `malloc()` is a pointer to a memory region of the requested size. In hexadecimal representation, this value can be represented as e.g. `0x1980010`. Further calls to `malloc(32)` could yield pointers of equidistant values.

Recalling the fact that `mlock()` operates on whole pages, the results of a series of `malloc()` calls could look like the following table:

Pointer address	Page address
0x1980010	0x1980000
0x1980040	0x1980000
0x1980070	0x1980000

with the page address being calculated as

```
page_addr = pointer & (~(PAGE_SIZE - 1))
```

As can be seen, multiple memory allocations result in multiple memory segments located on the same memory page if the requested memory is small enough. Calling `mlock()` after each allocation would work perfectly, but as soon as the first pointer is unlocked by using `munlock()`, *all* other pointers on the same page are unlocked as well. This circumstance reduces the benefits gained by using `mlock()` to nothing.

A first solution would be the use of `posix_memalign()` instead of `malloc()` while keeping track of used page addresses for later reuse to reduce the memory footprint of the software. On Linux however, the default limit of locked memory is 64 kB. With a page size of 4 kB, the process would only be able to store up to 16 encryption keys in memory until running out of memory. Furthermore, the approach would not be of much use as it also requires additional memory management to use the remaining space on a page or without memory management, wastes a lot of memory due to an encrypted key being much smaller than a page (typically 32 byte for a key and 4096 byte for a page).

A better and also implemented approach would be to divide each page into slots and use some sort of housekeeping to keep track of the status of each slot. This allows easy management of memory as the issue of stackable locking is of no concern any more. Further, the allocated memory is utilized in a much more efficient way as a page of 4 kB can be used to store up to 128 keys. With a default limit of 64 kB of locked memory, the total amount of keys lockable in memory results in 2048. Of course, the real number of data that can be stored is below that limit as necessary housekeeping information also occupies memory. As a result, the password management subsystem maintains a series of caches as described in subsection 5.4.1 "Slab allocator" that are used to securely store sensitive information and to prevent the operating system from swapping the memory out to disk which could be a potential information leak if the hard drive is not encrypted.

The second area of use for the password management is the retrieval of key material from the key daemon *keyd* as described in chapter section 4.2 "Architecture". Password lookups are initiated through one of two functions: `ocbsecret_for_path()` and `ocbsecret_for_guid()`. The former function is typically used when a file is to be

created and the file's path is the only available information. It is sent to *keyd* which uses its mapping policy to translate the path into a key identifier and tries to find a match for it in the user's keystore. The latter is used to open a file where a valid key identifier is already contained in the file header. It is also sent to *keyd*, which in this case does not have to perform a mapping, but forwards the key directly to *keyd–runas* instead to get a key from the keystore.

Both functions eventually call `ocbsecret_get_from_store()`. This function undertakes a series of steps to obtain passwords for either forward–mappings of file system paths to key identifiers, or the reverse mapping of key identifier to password. In any case, the result is a password as obtained from a keystore by *keyd* and *keyd–runas*. The interface between the password manager and the keyd communicator is based on the opaque structure `keyd_comm_handle`. When a password has to be looked up, `keyd_get_handle()` is the first function to start the procedure by obtaining such a handle. Because a connection needs to be established to keyd's unix domain socket, `keyd_set_opt()` with option `KEYD_COMM_PATH` can be used to adjust the path the communicator is trying to connect to. Ultimately, the function calls `keyd_get_secret()` to establish a connection with *keyd*, send the request and then wait for a response. To avoid indefinite waiting and thus a not responding process, the time the function waits for a response can be configured separately by using `keyd_set_opt()` with option `KEYD_COMM_RECV_TIMEOUT` on the handle before invoking `keyd_get_secret()`. By default, a value of 60 seconds will be used. The result of a lookup operation is a `struct ocbsecret` as shown in listing 5.3

Snippet 5.3: ocbsecret data structure

```
1  struct ocbsecret {
2      unsigned char *secret;      /* Password content */
3      size_t len;                 /* Length of secret field */
4      time_t last_used;           /* Timestamp of last usage */
5      char *guid;                 /* Identifier for the secret */
6      size_t guid_len;            /* Length of the identifier */
7      pthread_rwlock_t rwlock;    /* Read/write lock for concurrent access */
8      atomic_t refcnt;            /* Reference counter */
9      mem_cache_t *cache;         /* Cache the secret is stored in */
10 };
```

If the password manager receives a secret, it will insert it in the appropriate password cache. Each secret has two keys linking to it: the path a secret is valid for and a key identifier.

Key mappings

SafeGuard Enterprise uses path based policy mappings. When the password manager receives a request for a given path, it strips off the part behind the last "/" as two files in the same directory would map to the same policy anyway. Doing so prevents some lookups as the password for a given path might be already contained in the cache in consequence of previous lookups.

The number of caches the password manager maintains depends on `KEYD_PASSWD_-MIN_LENGTH` and `KEYD_PASSWD_MAX_LENGTH` with default values of 8 and 64 respectively, but can be overridden by the configuration values `pass_min_len` and `pass_-max_len` respectively in the configuration file in section `passman`. During application startup, `create_secure_storage()` will take these values and set up an array of caches for the specified range. Each cache supports a specific object size that is in the range `pass_min_len` and `pass_max_len` and that is a power of two. After a password has been received from *keyd*, its length will be converted to an index in the `secure_storage` array. The password manager then allocates an object from the cache at `secure_store[index]` and stores the secret.

Note:

The caches created by `create_secure_storage()` have the special self–defined flag `SLAB_PROTECTED` set. As will be shown in section subsection 5.4.1 "Slab allocator", this flags is used to automatically lock allocated pages in memory and prevent the operating system from swapping them out to disk.

To allow applications access to the retrieved passwords, the password manager provides accessor functions to its users. Defined in `passman.h`, their functionality includes:

`ocbsecret_lock()`
: Because applications might request access to data concurrently, the `lock()` applies a read lock on the secret to protect it from being freed while the information is in use. The data must *not* be modified.

`ocbsecret_unlock()`
: Signals completion by unlocking the read lock.

`ocbsecret_get_guid_locked()`
: As the name suggest, this function returns a constant pointer to the key identifier. The data must *not* be modified.

`ocbsecret_get_secret_locked()`
: Returns the length of the secret along with a constant pointer to its storage. The data must *not* be modified.

`ocbsecret_unref()`
: Releases the secret. This entails updating the `last_used` timestamp and decreasing the reference counter. It is not safe to keep a reference to the secret as it might gets released due to the expiration timer. Further access will therefore might cause a segmentation fault and application termination.

5.4 Memory management

Memory management as used in the FUSE file system serves two purposes: increasing performance by keeping frequently used structures allocated in memory and to allow sub–systems that deal with sensitive information like encryption keys to securely store them in memory.

5.4.1 Slab allocator

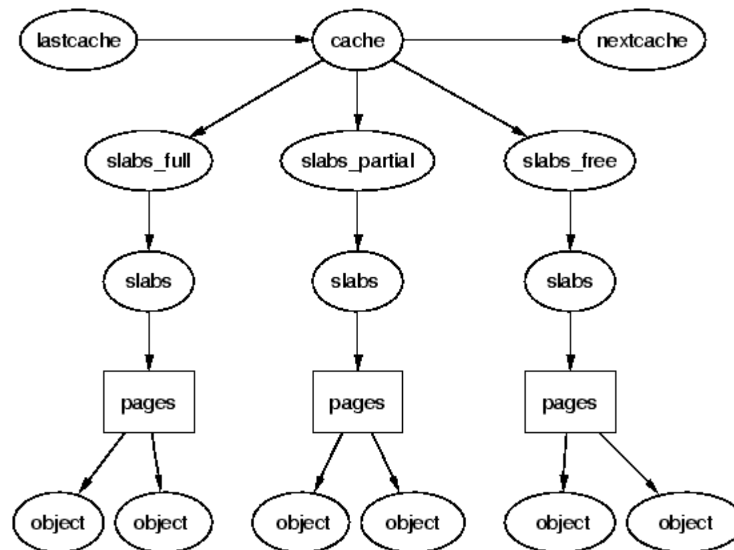
One of the main tasks a kernel has to perform is memory management. Due to its high importance, the kernel tries to conduct it as efficiently as possible to achieve low internal memory fragmentation and to speed up the process of memory allocation. For userspace applications, memory allocation is a simple three step operation:

1. Allocate memory using one of the `malloc()` functions
2. Manipulate the content of the returned memory portion
3. Return the memory to the operating system by calling `free()`

For the kernel however, memory allocation is a much more tedious task. It has to keep track of a virtual addresses, their affiliation to certain processes and it might also has to perform swapping of underlying pages to and from disk when physical memory is tight. But the kernel does not only satisfy request from user–space. The kernel occupies memory for itself through file system operations, incoming network packets and various other events that happen dynamically. To efficiently provide memory for those operations, the Linux kernel employs an algorithm known as *slab allocator* as described in [Bon94]. When performing a specific task, the size argument to `malloc()` is of constant value in many cases, as the same data structures have to be allocated repeatedly (e.g. a `struct inode` in file system code). When following the aforementioned three step approach, necessary constructors and destructors have to be executed each time an object is requested, which needlessly slows down the allocation process.

The slab allocator renders these steps superfluous as it does not really free objects when the user returns memory to the allocator. Instead, it keeps objects cached in an initialized state to satisfy subsequent requests for the same object type in a shorter amount of time. For the user, the three step allocation scheme remains identical,

Figure 5.1: Slab allocator overview [Bon94]



except the fact that a cache needs to be created and deleted during application startup and shutdown respectively. The memory an initialized object is stored in is called a *slab*. A slab is a continuous region of memory of size $n * PAGE_SIZE$, where n denotes the *order* of the cache. As shown in figure 5.1, a slab can be in one of three states: empty, partially full, or full. Allocating and freeing objects from the cache therefore does not necessarily involve the operating system at all. In cases where slabs are available on the free or partially free list, the request can be satisfied immediately. Only when both lists are empty, the cache needs to allocate pages from the operating system. The slab allocator as implemented in `slab.c` provides the following functions to facilitate memory management:

`mem_cache_create()`
: Creates a new cache for object of size according to the `size` parameter and adds it to the global cache list. It calculates the amount of object that fit on a slab and sets cache–global variables on the cache itself.

`mem_cache_destroy()`
: Attempts to destroy a cache. If there are still objects allocated on a slab contained in the cache, it will return the error `EBUSY`. Otherwise it unlinks the cache from the global cache list and frees all associated slabs.

`mem_cache_alloc()`
: Allocates a object from the given cache, but also allocate memory from the operating system in case all slots are occupied.

`mem_cache_free()`
: Returns an object to the cache

Initially, storing meta–information is the sole memory consumption of an empty cache. When applications allocate memory in this state, the request cannot be satisfied immediately. In this case, the cache has to grow. It does so by allocating `PAGE_SIZE << order` of memory using `posix_memalign()`. Figure 5.2 shows the result of the cache grow process. Each slab contains a slab descriptor structure as seen in listing 5.4 at the beginning of the memory region to ensure efficient management of newly created slabs.

Snippet 5.4: slab descriptor structure

```
typedef struct slab_s {
    struct list_head list;      /* List this slab belongs to */
    void *s_mem;                /* Pointer to the 1st object */
    unsigned int inuse;         /* Number of allocated objects */
    mem_bufctl_t free;          /* Location of free objects */
} slab_t;
```

The `list` field specifies which list (empty, partially full, or full) the slab belongs to. `inuse` simply stores the number of active items in the slab and will be used to quickly determine if the slab has become empty or full as a result of `mem_cache_alloc()` or `mem_cache_free()`. If so, the slab will be moved from one list to another and the `list` field is updated accordingly. The two other members are used to keep track of object slots. `s_mem` is a pointer to the first available slot. As seen in figure 5.2, it points right after the last `mem_bufctl_t`. The area between the `slab_t` structure and the actual object storage contains an array of `mem_bufctl_t` entries that are used to keep track of free items. When the slab is created, the array of bufctl's will be consecutively numbered as shown in listing 5.5 as part of `mem_cache_init_objs()`.

Snippet 5.5: slab allocator slab initialization

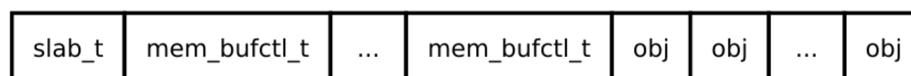
```
void mem_cache_init_objs(mem_cache_t *cachep, slab_t *slabp)
{
    int i;

    for (i = 0; i < cachep->num; i++)
        slab_bufctl(slabp)[i] = i + 1;

    slab_bufctl(slabp)[i - 1] = BUFCTL_END;
    slabp->free = 0;
}
```

This code sets up a chain that links the bufctl's, with the `free` field pointing to the first available item. When an allocation has to be performed,

Figure 5.2: Slab allocator – slab layout



```
slabp->s_mem + slabp->free * cachep->objsize
```

yields the address of the next available slot. Since the slot pointed to by `free` is now in use, the field also needs to be updated according to

```
slabp->free = slab_bufctl(slabp)[slabp->free]
```

When an object is returned to the cache, the bufctl corresponding to the slot number has to be updated, otherwise the slot would be lost as no other bufctl is referring to it, thus making it impossible to allocate it. Given a pointer to an object in the slab,

```
(objp - slabp->s_mem) / cachep->objsize
```

calculates the slot number within the slab. With that information, the corresponding bufctl can be updated accordingly using

```
slab_bufctl(slabp)[objnr] = slabp->free
slabp->free = objnr
```

hence turning memory allocation into a last in – first out allocator.

As applications keep allocating and freeing from the allocator, it might runs out of memory and needs to allocate more slabs from the operating system. Over time, this could lead to numerous slabs on the free list being unused and memory would be wasted. To remedy that problem, the slab allocator contains a component that examines the caches and tries to free slabs that are not used any more. The *cache reaper* is initialized in the `mem_cache_init()` function and periodically calls `mem_cache_reap()`. This function performs the following checks:

- If the cache is marked `SLAB_NO_REAP` or is currently growing, skip it
- Count how many pages could be freed
- If `REAP_PERFECT` or more pages could be freed, free half of the slabs from the free list
- Otherwise keep looking for a candidate that yields the most memory when freeing half of the slabs from the free list

The reaper runs every 10 seconds as memory consumption is not an issue for the FUSE file system. Such situations can only arise when e.g. a large number of files is copied to a directory and the file manager needs to allocate corresponding `vfile` structures. However, if the reaper has freed more than `REAP_THRESH` pages, its sleep time will be reduced by 50% until a minimum of two seconds is reached. After the caches have been drained sufficiently, its sleep time will grow back to `REAP_SLEEP_TIME`.

A log entry as reported on request by the FUSE file system may contain an entry like the following:

Name	objsize	nr_objs	active	allocs	grown	reaped
vfile_cache	184	21	8174	16386	781	304

The line gives information about the file manager's vfile cache. As these objects are used quite frequently, the file manager uses a slab cache to speed up their allocation. Furthermore, during cache creation, `mem_cache_estimate()` has calculated that for a given size of 184 byte each, 21 `vfile` objects will fit on a slab of size 4096. 232 bytes or 5.6% are used for bookkeeping. On a 64–bit platform, 32 byte of that will be occupied by the slab descriptor as seen in listing 5.4. The rest of 200 bytes is used for the bufctl array as depicted in figure 5.2. 21 objects require 21 bufctls, therefore occupying additional 84 bytes. The remaining 116 bytes are not used. The next number indicates the currently active (allocated) objects for this cache. 16386 objects were allocated consecutively which caused the slab allocator to grow this cache 781 times. Since half of the objects have been freed already, causing the allocator to move the slabs to the `slab_free` list, the cache reaper gave back 304 pages to the operating system. It should be noted that the reaper does not merge slabs. If 21000 objects had been allocated and every other object is freed, the cache would still occupy 1000 pages as the slabs would remain on the `slab_partial` list.

When creating a cache using `mem_cache_create()`, one can specify a bitmask of flags for cache creation. One of those flags is `SLAB_PROTECTED`. Facilities like the password manager or the crypto system can use this flag to instruct the cache to automatically perform a call to `mlock()` after additional pages have been allocated. With the slab allocator in place, its users can utilize the amount of locked memory compared to the initial approach. Furthermore, it can be used as an ordinary memory allocator by not specifying the `SLAB_PROTECTED` flag during creation.

5.5 Crypto core

So far, the user was able to mount the file system and perform open and close operations on files through the file manager, which also took care of caching of meta–information and file opening or creation on the underlying file system. Clearly, a file system is of no use if it does not support any data manipulation operation. Therefore, this chapter will explain how the `read()` and `write()` callbacks are implemented, but also the internal interfaces to the file and password manager.

The crypto system takes over right after the file manager has finished its work. The starting point is a valid vfile object in the `open()` functions which was returned by the file manager. Until now, all introduced data structures did not carry any information that maps them to a particular user, because they are valid throughout the system and not bound to any particular user. They are maintained in a global cache with access for all users on the system. The cryptographic context as introduced shortly however is assigned to a specific transaction. It is the file system's equivalent

of a file descriptor. Similar to the well–known system calls which use the file descriptor as their first argument, the cryptographic context is then automatically passed to functions by FUSE. The context is obtained after the file manager has returned a valid vfile object by using `ocbfile_state_init()`. As the name suggest, this function initialized the state and assigns it to the user. The information a state carries are shown in listing 5.6.

Snippet 5.6: Context structure

```
struct vstate {
    int fd;                     /* fd to underlying file */
    uid_t uid;                  /* user ID */
    gid_t gid;                  /* group ID */
    dev_t dev;                  /* device ID of the underlying file */
    ino_t ino;                  /* inode number of the underlying file */
    struct ocbfile *file;       /* pointer to the cached meta-data */
    void *crypto_state;         /* cryptographic information */
    crypto_cleaner_t cleaner;   /* cleanup function */
};
```

`fd` is the file descriptor that refers to the underlying file and that has been obtained by the file manager through `open()` or `creat()`.

`uid`, `gid`, `dev`, `ino`
can be used to ensure that the state belongs to a transaction in later calls to `read()` or `write()`. Mainly used for debugging purposes to verify that the state information is intact.

`file`
link to the file that holds all information that are file–global and used for cryptographic operations: initialization vector, encryption key, and block size.

`crypto_state`
since multiple file versions could require different state information, this field contains version–dependent information.

`cleaner`
function pointer that cleans up `crypto_state` when the context is not needed any more.

Upon creation, the `file` member is set using `ocbfile_ref()` and during deletion freed using `ocbfile_unref()`. This approach increments `refcnt` in the vfile object and ensures that the base vfile structure is not released and deleted while there is state information attached to it.

If the file header parser invoked by the file manager was successful, it has assigned a set of functions to the vfile object before it returned. These functions will now be

used by the crypto system to set up a version depended state that will handle all crypto operations performed on this file.

Snippet 5.7: Crypto operations

```
1  struct crypto_ops {
2      int (*get_version)(void);
3      void (*state_free)(void *state);
4      int (*state_init)(void **state, const void *cipher,
5                        void *key, const struct ocbkey_ops *key_ops,
6                        size_t block_size, int fd);
7      ssize_t (*read)(void *state, struct io_request *req);
8      int (*write)(void *state, struct io_request *req);
9      int (*truncate)(void *state, off_t size);
10 };
```

Listing 5.7 shows the operations provided by each crypto handler. `init()` and `free()` are used to create and delete states, whereas the last three functions are equal to the identically named system calls. After `ocbfile_state_init()` has created the state with common information about the file, it will invoke the `state_init()` function to initialize necessary cryptographic context on a per transaction basis and assigns it to the `crypto_state` member in the vstate structure.

5.5.1 Crypto states

If a file has been opened and the user issues a write system call, a buffer has to be supplied that holds the data that needs to be written to the file. Depending on the size of that buffer, the kernel might split the write into several calls and invokes the FUSE write callback only with a chunk of that buffer until all data has been written to disk. It is therefore the file system's responsibility to handle each junk independently and guarantee the correct handling of data split across several write requests. In addition, required data structures remain the same on a per–file basis, so leaving them initialized from one call to another saves time due to their already initialized state.

5.5.2 File meta–information

Irregardless whether a file has been opened or newly created, the crypto subsystem maintains a structure `ocbkey` for each file. The structure holds information necessary to initialize a crypto state if a new transaction on the file is issued.

Snippet 5.8: Per–file cryptographic data

```
1  struct ocbkey {
2      struct ocbfile *file;          /* Associated file */
3      byte_t *dek;                   /* Data encryption key */
4      mem_cache_t *dek_cache;        /* Secure cache for the DEK */
```

```
5      size_t dek_len;                  /* Length of the encryption key */
6      byte_t iv[FE_IV_MAX_LENGTH];     /* Initialization vector */
7      byte_t mac[FE_MAC_NULLW_SIZE];   /* MAC for key decryption */
8      byte_t blob[FE_DEK_BLOB_SIZE];   /* Encrypted encryption key */
9      const void *cipher;              /* Associated cipher */
10     char *symbolic_name;             /* Symbolic name of this key */
11     char *kek_id;                    /* ID of this key */
12     int locked;                      /* Blob needs to decrypted to get DEK */
13     struct fe_padding *pad_hdr;      /* Padding header for the last block */
14 };
```

On file creation, the crypto system generates a new DEK and initialization vector for the file. `crypto_rand()` can be called to obtain a certain amount of random data that is suitable for initialization vectors and data encryption keys, as long as the `strong` parameter is set to a non–zero value. Internally, the function uses the OpenSSL library to produce random bytes by calling `RAND_bytes()`.

OpenSSL:

OpenSSL is a widely used library with support for all major operating systems. According to [Pro], it describes itself as *"a collaborative effort to develop a robust, commercial–grade, full–featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library."* Due to its availability is was the first choice during development as it eases porting the software to other platforms.
The library implements all cipher suits currently approved by NIST along with their modes of operation. Currently, the cipher used throughout the application is the AES. The mode of operation depends on the intended goal in a particular situation but uses one of the modes as described in chapter section 2.3 "Block Cipher – Modes of operation".

The data encryption key is stored in non–swappable memory to prevent the raw key to be written to disk. It does so by maintaining SLAB caches for common key sizes as described in section 5.4.1. Since AES–256 has a 32 byte key size, the most important one is the `ocbkey32_cache`. While the file is open and referenced, the content of the key remains in memory. When the last reference to the file is closed however, the file manager transfers it to the inactive cache, where it remains until it is either reused or deleted as a consequence of its expiration.

Because crypto information could remain in memory for a longer period of time, keeping the DEK in memory can be considered a risk and should therefore be avoided (which is also the reason why there is no *inactive* cache for passwords like there is for vfile objects). Furthermore, the DEK has to be recoverable when the file is opened again, therefore it is written to disk along with other information when the file header

is exported. The procedure of initializing file meta–data involves several stages, split into the following series of events:

1. An empty `ocbkey` is created. Random data is obtained by calling `crypto_rand()` to generate a DEK and IV.

2. The known file path is sent to the password manager which in turn forwards the path as a type `P` request to *keyd*.

3. If *keyd* reports an error, file generation stops and control is returned to the file manager which reports the error to the user.

4. Otherwise the result of the password lookup includes the user's password along with an identifier. The password will undergo a key expansion process that yields the Key Encryption Key (KEK).

5. The random DEK is encrypted using the KEK and stored in the `blob` field. The cipher used is AES in CBC mode.

The resulting meta–information is kept in memory, but also written to disk by the time the header is exported using `ocbfeheader_write()` as all necessary information is already generated and can be returned as is. A more complex scenario arises if a header is to be deserialized as a result of a `ocbfeheader_read()` invocation. The function reads the header from disk and passes the header to the crypto subsystem where the above-mentioned process has to be reverted to obtain a decrypted version of the DEK.

1. An empty `ocbkey` is created. The IV is copied to the key object. The encrypted DEK is stored in the blob field.

2. The key identifier is read from the header and passed to the password manager which then issues a type `G` request to obtain the user's password.

3. If *keyd* reports an error, file deserialization is aborted and an error is returned to the user.

4. Otherwise the user password undergoes the very same key expansion that gives the same KEK as used for encryption.

5. The DEK is recovered by decrypting the blob using the KEK and stored in the `dek` field.

The `locked` field in an `ocbkey` object indicates the state of the DEK. If `locked` is true, DEK memory is allocated, but does not contain valid data. Attempts to obtain the key will result in an additional request to the password manager in order to be able to decrypt it. Because the file manager keeps passwords cached, it is possible that it

can return a password immediately. Otherwise, a request to *key* is issued to obtain the user's password. The `locked` field changes during a vfile transition from the active to the inactive cache or vice versa.

5.5.3 IO operations

The main task for the crypto subsystem is the handling of read and write operations on the underlying file. These functions are invoked when an application issues `read()` and `write()` system calls respectively. In the FUSE file system, the user issued calls are transformed to `io_requests` as seen in listing 5.9.

Snippet 5.9: IO request

```
1 struct io_request {
2     off_t offset;   /* Offset to write to */
3     byte_t *data;   /* Data to be written */
4     size_t len;     /* Size of the data */
5 };
```

Before IO requests are written to disk, the data has to be processed by an encryption algorithm. Currently, CBC mode as described in chapter 2.3.2 is supported to provide compatibility with *SafeGuard Enterprise*. Before elaborating on how read and write operations are implemented, it should be noted that generally there is no "correct" way for applications to read and write data. An application may

1. Allocate a large buffer
2. Read up to size(buffer) from the file

or

1. `stat()` the file to be read
2. Allocate a buffer to hold the entire file
3. Read the file
4. (optional) verify that the read call returned `st_size` bytes of data to ensure success.

or

1. Open the file for reading
2. `mmap` the file into the processes' virtual memory. See `man 2 mmap` for additional details.
3. Access the file on a per–page basis as the operating system loads and unloads content from the file.

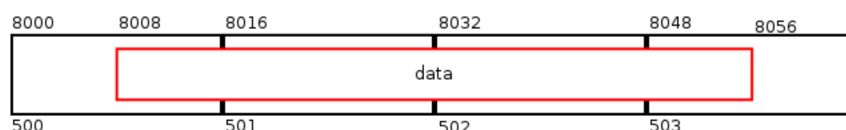
Because the file system should not constrain applications, there should be no assumptions about how an application performs its IO. Each request has to be treated independently. As far as the second example is concerned, if an application performs the optional verification of the read system call, it matches the return value of `read()` to the file size it has obtained through `stat()`. The `stat()` call is relayed to the file system which in turn stats the underlying file and because each file contains an encryption header, it simply subtracts the header size from the `st_size` field. This behavior is implemented in the vfile layer. The vfile layer is however not aware of the underlying encryption scheme used. Depending on the used block cipher mode, additional data in the form of padding or additional authenticated data (AAD) might be present. The vfile layer therefore passes a `stat()` call on to the crypto layer which is then able to calculate the exact file size.

Despite the file size issues, applications might also issue IO requests to arbitrary offsets within the file. From an application's point of view, a request to a certain offset within the file is perfectly valid, but the following example should illustrate the implications of random access for a cryptographic file system:

file size	**operation**	**offset**	**length**
1 MB	read	8008	48

With regards to CBC mode, the request cannot be satisfied directly. As can be seen in figure 5.3, the length of the request is a multiple of the AES block size of 16 byte, the access happens on a – with respect to the block size – unaligned offset. Therefore, the offset has to be rounded down to the nearest block boundary (8000 in this example) and the length of the read request has to be extended to 64. After decryption, spare data needs to be discarded securely before the final outcome is returned to the user. The situation gets even worse if the previous request changes to a write operation. As for the read operation, data needs to be aligned, read into a buffer, replaced, and finally written to the underlying file. Due to the usage of CBC mode, the replacement causes all blocks from number 500 onwards to become invalid as each block n acts as initialization vector for block $n+1$. Given a 1 MB file with a total of 6656 blocks, this circumstance requires the re–encryption of 6156 full blocks. In the worst-case scenario, write access happens at offset 0, the consequence being that the whole file needs to be rewritten in order to propagate the change throughout the file. Clearly, this approach suffers from major drawbacks as unnecessary crypto operations slow

Figure 5.3: IO request – read access



down the process and additional disk IO might increases degradation of hard drives, especially if solid–state drives are used.

As a counter–measure, the file is divided into blocks of constant size. A block itself is an isolated unit within the file. Changes in one block do not affect other blocks. In addition, each block possesses an individual initialization vector, derived from the random per–file IV. Applied to the previous example with a block size of 512, the following steps have to be taken to read data according to the request:

1. The request offset is rounded down to the nearest 512 byte block boundary. Dividing the aligned offset by the file block size yields the block number.

2. Based on the per–file initialization vector, a block IV is calculated for this particular block.

3. The block is decrypted.

4. The result is truncated to meet the request and returned.

For read operations, the application of file segmentation does not bear immediate benefits. For write operations however, the decryption process only affects blocks depending on the length parameter of the request.

1. The request offset is rounded down to the nearest 512 byte block boundary and divided by the file block size to get the block number.

2. Based on the per–file initialization vector, a block IV is calculated.

3. If the offset does not match a file block boundary, the whole block is decrypted.

4. The affected portion of the decrypted buffer is overwritten with the new data.

5. The block is re–encrypted and written to disk.

6. If the request spans additional blocks, the procedure is repeated until all new data has been processed.

Using internal file segmentation therefore has less impact on the overall throughput of the write operation itself and the underlying hard drive. Compared to the initial write request where the affected position was at offset zero and required rewriting of the entire file, segmentation only affects one block. In addition, IO throughout might also increase if the number of affected blocks is relatively small as disk fragmentation is of no concern.

Read requests

A read request is the result of a user invoking the `read()` system call. The FUSE library will relay the call to the specified FUSE callback along with the crypto state as explained in the previous section. The file system will verify that the state is valid and eventually call `ocbfile_read()`.

Since it could be possible for a multi–threaded file system that multiple users are accessing the same file concurrently, each file is protected by a read–write lock which in this case is locked for reading.

> **read–write locks**
>
> A read–write lock is a synchronization primitive used in multi–threaded programming. It is used to protect common data from concurrent modifications. On Linux, read–write locks can be used if the application is linked with the POSIX library `pthread`.
> A read–write lock is particularly useful if data is read more often than it is written. In such cases, the application can allow users to perform reads concurrently. Using a mutex would be possible, but it limits the number of users with access to the data to one. A read–write lock grants multiple users concurrent access as long as they happen read–only leading to higher throughput. Write access suffers from the same penalties mutexes do.
> If a lock is locked for writing, each lock–to–read request is simply delayed and the thread is put to sleep. The same applies to the lock being locked for reading an a lock–to–write request occurs. Sleeping threads will be woken when a respective unlock occurs (refer to `man 3 pthread_rwlock_unlock` for additional information about Linux' implementation and especially how waiting threads are prioritized).

Write requests

The complexity of the write operation mainly depends on the parameters passed to the `write()` function. Especially the offset parameter that specifies where in the file the access has been requested directly influences which work has to be done by this function. In the simplest case the offset as well as the size parameters are a multiple of the block size. Then data is encrypted, written to disk and the operation returns.

In all other cases, the final write operation is preceded by a read operation that reads in partial blocks that are then overwritten with the new data.

Truncate requests

Depending on the desired size of the file, truncate requests are satisfied by either writing additional zeros to the file using the crypto `write()` function as introduced

before or by trimming the file to the next lower block size boundary and appending a partial block – also reusing the crypto `write()` method. This way, the truncate operation does not add any complexity to the software.

5.6 Mountpoint bypassing

File systems using FUSE usually take a single path as their last argument that tells the kernel module where in the file system hierarchy the to–be–created file system should be mounted. The generic nature of FUSE does not impose any restrictions or requirements on the file system's features. Instead, it just forwards all requests to the respective callback functions and reports the results back to the calling application.

Since running the file system presented in this paper to enforce certain company policies, circumventing encryption is a major problem. However, the data the file system exposes is not artificial, thus it must eventually persist all data passed to it on a physic medium which leads to two folders being present on the system:

- A folder that has been specified when the file system was started. This folder performs transparent file en– and decryption (e.g. ~/Documents_SECURE).
- A folder all data is read from and written to when it is processed by `cfs` (e.g. ~/Documents).

Especially for untrained users this could pose as a risk as it is likely they do not understand how to handle this situation correctly. Things like "Save as", "Open", "Recent files" and so on initially still refer to the *wrong* files (under ~/Documents). The problem is point–blank the existence of two folders with the same files in it. Therefore, `cfs` uses another technique to get along with only one folder in the file system: using the `at` extended system calls where appropriate (e.g. `openat()` instead of `open()`. These system calls require a file descriptor and a relative path to function. FUSE already provides the relative path. The file descriptor is obtained by opening the path specified when `cfs` is invoked. The file system is then mounted to the same location. All operations use the file descriptor that still refers *to the original folder*. Tools like `ls` or graphical file managers only display one folder.

If for some reason this technique does not work, one has to manually track changes to the original directory and encrypt files after they have been closed.

Since version `2.6.13`, the Linux kernel offers a powerful and efficient facility to monitor changes to directories and files through *inotify*. To make use of that notification mechanism, an application registers a set of events it is interested in for a specific directory. Due to the fact that the obtained *watch descriptor* is not watching a directory recursively, each directory along a given file system tree has to be watched separately.

inotify will be used to monitor changes to the root folder of the FUSE file system. Recalling file meta–information caching in the the file manager subsystem, it becomes

obvious that changes to files in the root directory are only of interest if the affected files are not cached yet. When a file is to be manipulated, the user has to perform at least a three step approach to do so:

1. `open()` a file
2. `write()` data to the obtained file descriptor
3. `close()` the file

When performed on a file through the exposed FUSE mountpoint, step 1 will already trigger caching of meta–information. By the time a file is closed as seen in step 3, two possible situations can occur:

- the meta–information has been cached already. It is assured that the requested operation was performed on the FUSE mountpoint and file encryption has been performed.
- the cache does not contain any information about the file. This is an indicator that the `write()` operation was performed on the root directory rather than on the FUSE mountpoint because the file system did not see a corresponding `open()` operation.

In the latter case, the mountpoint watchdog needs to detect this policy violation and move the file to the same location with the write occurring on the FUSE mountpoint to enforce file encryption.

Snippet 5.10: notifyd info structure

```
1  typedef struct notifyd_handle notifyd_handle_t;
2
3  struct notifyd_callbacks {
4      void (*on_start)(void *u);
5      void (*on_stop)(void *u);
6      void (*on_close)(const char *path, void *u);
7  };
8
9  struct notifyd_info {
10     char *root;                          /* root directory to watch */
11     void *priv_data;                     /* user data */
12     struct notifyd_callbacks callbacks;  /* struct of callbacks */
13     notifyd_handle_t *handle;            /* opaque handle */
14 };
```

When the file system is started, it passes its root folder to `notifyd_init()` using a `struct notifyd_info` as described in listing 5.10. The `priv_data` member is a pointer to arbitrary data that will be available in all callbacks. The callback structure itself allows an user (the file manager in this case) to register a set of functions that will be

invoked when certain events are picked up by the watchdog. `on_close()` is the most important callback as it will be used to ensure that a file has really been encrypted.

Initializing the watchdog is done by calling `notifyd_init()` with an info structure set up appropriately. If successful, the watchdog will return an opaque `notifyd_handle_t` in `info->handle` that can be passed to `notifyd_stop()` to shut down the watchdog.

Interacting with inotify requires an application to first obtain a unique inotify instance through `inotify_init()`. The returned file descriptor can then be used in subsequent calls to `inotify_add_watch()` and `inotify_rm_watch()` to add and remove watch descriptors to and from files and folders respectively. A watch descriptor is a unique number (per inotify instance) that allows the application to keep track of where in the file system the event was triggered. Because the file system most likely already contains existing files, the watchdog has to walk through the file system structure starting at the given root directory and install a watch for each encountered directory. The *notifyd* subsystem performs those steps in the `notifyd_init()` function. First, *notifyd* installs a watch descriptor for the root directory and adds the the value to the item cache. It then uses `notifyd_scan_recursive()` to recursively walk through the file tree and install watch descriptors for the preexisting folders.

5.6.1 Watch descriptor management

When inotify events for files are received, the watch descriptor refers to the parent directory of the file, as *notifyd* does not watch single files for performance reasons. Therefore, it has to keep track of where the watch descriptor of an event refers to in the file system. A simple solution would be to store a mapping between watch descriptors and file system paths and perform lookups if a file event occurs. As memory consumption of an application should always be kept as low as possible, this approach seems unrewarding as the stored path information would carry a lot of redundancy when performing such a mapping. When a new subdirectory `s` is created in an already monitored directory `d` with path `p`, the application has to store the new watch descriptor `wd(d)` along with the path of that folder which consists of the path `p` of the parent folder and the name of the subdirectory `s`. In case of the `notifyd_init()` function, this process has to be repeated for each encountered directory along the path. Given a root directory with a large amount of subdirectories, keeping track of full paths therefore requires a lot of memory, especially if directories have long names (e.g. when a certain naming scheme for folders is required) and a lots of subdirectories. A different approach has to be chosen to reduce the memory footprint of *notifyd* in order to allow versatile usage.

Until now, *notifyd* keeps track of full paths so that for a given `wd` the corresponding path can be easily recovered. Appending the event name to that path yields the full path of an event in very few steps. As mentioned before, the path of a directory carries a lot of redundancy, so optimization should address primarily the path handling.

When walking the directory tree specified by the `root` parameter in `notifyd_scan_recursive()`, the information available in each function call consists of the parent directory and the current directory name. Passing the concatenation of those two paths to `inotify_add_watch()` yields a new watch descriptor for the current directory which is then added to the cache.

Note:

There is a certain upper limit for open watch descriptors. The limit applies to real UIDs which could cause less than *limit* watch descriptors being available if other applications running with the same UID already make use of inotify. The default limit for inotify watch descriptors can be obtained by running the following command on a terminal:

```
$ cat /proc/sys/fs/inotify/max_user_watches
```

If notifyd is running out of watch descriptors for a large root directory to monitor, the limit can be temporarily raised by `echo`ing an appropriate value to `max_user_watches` or permanently set by adjusting `fs.inotify.max_user_watches` in `/etc/sysctl.conf`.

When traversing the directory, the obtained watch descriptor is not preserved as it is not needed because all paths are available at any time. Therefore, as a first optimization, the watch descriptor for a directory is passed along the path. Table 5.1 shows the resulting recursive function calls up to a level of three below the root. As can be seen in table 5.1, the cache data contains the full path of an entry. Another approach to recover the full path is the following algorithm:

Snippet 5.11: notifyd recursive path builder

```
1  function build_path(wd, path)
2  {
3      if wd == -1
```

Table 5.1: notifyd recursive cache set up

Level	pd	parent path	entry	cache key (wd)	cache data
0	-1	NULL	root	1	root
1	1	root	s1	2	root/s1
2	2	root/s1	s2	3	root/s1/s2
3	3	root/s1/s2	s3	4	root/s1/s2/s3
2	2	root/s1	s4	5	root/s1/s4
3	3	root/s1/s4	s5	6	root/s1/s4/s5
1	1	root	s6	7	root/s6

```
4           return
5
6       path = prepend(entry(wd), path)
7       return build_path(pd(wd), path)
8   }
9
10  function get_path(pd, entry, path)
11  {
12      path = entry
13      return build_path(pd, path)
14  }
```

Using the pseudo code of listing 5.11 makes tracking of full paths superfluous, as they can be rebuilt on demand. The downside of applying the path recovering algorithm emerges when deleting directories. As only the watch descriptor *wd* for the parent directory and the name of the deleted directory *name* are available, first the full path of the parent directory *parent_path* has to be rebuilt, which simultaneously yields the parent watch descriptor *pd*. With that information, the whole cache has to be scanned for an item with name equal to *name* and parent watch descriptor equal to *pd*. When recursively deleting folders, the deletion process starts with the leaf nodes and from there back up to the top directory, therefore causing the application of the rebuild algorithm and cache scanning for each directory, as no direct mapping between paths and watch descriptors is kept. However, recursive folder deletion can be considered a rare event, so the additional work might be neglectable for shallow directory structures. In other cases, when memory consumption is not an issue, an additional cache could be kept to maintain forward mappings from paths to watch descriptors. If both options are not applicable, the FUSE file system has to forbid directory removal entirely by simply returning an error from the `rmdir()` function callback. File removal is then only possible through the root directory where the encrypted data resides.

5.6.2 Event handling

After an initial set of watch descriptors has been installed by walking the directory structure through `notifyd_scan_recursive()`, *notifyd* has to wait for incoming events dispatched by the kernel. As this call blocks until events are available, *notifyd* has to outsource this task to a dedicated thread. When events are queued by inotify, the `read()` call on the inotify file descriptor will return and the events will be available in the supplied buffer. The format of events as emitted by inotify is is described in listing 5.12.

Snippet 5.12: inotify event structure

```
1   struct inotify_event {
2       int      wd;       /* Watch descriptor */
3       uint32_t mask;     /* Mask of events */
4       uint32_t cookie;   /* Unique cookie */
```

```
5     uint32_t len;      /* Size of name field */
6     char     name[];   /* Optional null-terminated name */
7 };
```

The `wd` field will contain the watch descriptor the event occurred in. `mask` specifies the event that has happened in `wd`. The `cookie` field is used to link two related events. This is needed when watching for rename events for example. Both the `MOVED_FROM` and `MOVED_TO` events will have the same cookie value in order for an application to ascertain their affiliation. The `name` member carries the name of the affected file when the `wd` refers to a directory. Otherwise its content is empty and `len` is 0. Since only directories are watched, the `name` field will always be set.

When reading from the inotify file descriptor, multiple events may have been returned. `notify_process()` walks through the buffer and for each event performs a certain action:

event	type	action
create	file	discard
create	directory	`notifyd_dir_create()`
delete	file	discard
delete	directory	`notifyd_dir_delete()`
close	file	`notifyd_file_closed()`

- If a directory has been created (mask = `IN_CREATE`), it is added to the list of watched directories and to the inotify instance using `inotify_add_watch()`.
- If a directory has been removed (mask = `IN_DELETE`), it is removed from the list of watched directories and from the inotify instance using `inotify_rm_watch()`.
- If a file has been closed (mask = `IN_CLOSE_WRITE`), the callback function `on_close()` will be called. *Note:* The watchdog will react on the close flag, because waiting on the modify flag for instance would maybe interrupt an ongoing write operation when the file is moved.

The `on_close()` callback is implemented with the following prototype:

```
void (*on_close)(const char *path, int isdir)
```

By default, the callback is set to `fileman_notifyd_cb()`. This function first translates the path to a cache key and performs a lookup in the cache. If the lookup is successful, the write operation went through the FUSE mountpoint and no action needs to be taken. Otherwise, `fileman_move_dirty()` is called to move the unencrypted file through the mountpoint to the same location to trigger file encryption. It should be noted that the file cannot be simply moved through the mountpoint as the mountpoint itself reflects the file and folder structure of the root directory the file is already

contained in. Doing so would therefore overwrite the original file and lead to information loss. To illustrate the situation, the following steps were performed by the user or occurred as a result of them:

1. File system mounted with root directory `~/root` to FUSE mountpoint `~/fuse`.
2. Storage of a file `foo.txt` to `~/root/dira/dirb/foo.txt`, hence bypassing file encryption and possibly violating policies.
3. `notifyd` picks up the event, cannot find a corresponding file in the file manager cache, and therefore attempts to move the file.
4. `fileman_move_dirty()` is called with the path `~/root/dira/dirb/foo.txt` (*dirty path*).
5. Since both the root directory and the FUSE mountpoint are known, the *clean path* is simply a replacement option that yields `~/fuse/dira/dirb/foo.txt`.
6. As both paths refer to the same underlying file, the file cannot be moved at this time. `fileman_move_dirty()` uses the *dirty path* as input to a hash function (e.g. SHA–1) and prepends the *dirty path* excluding the file name which produces the *dirty random path* of the following form: `~/root/dira/dirb/{sha-1 hash}`.
7. As all needed information is available at this time, the file at *dirty path* is renamed to *dirty random path* using `rename()` to allow the creation of the file `foo.txt`.
8. The file at *dirty random path* is copied to *clean path* using the `sendfile()` syscall as a simple `rename()` would cause an `EXDEV` error because source and destination folder are not located on the same file system which is a requirement of `rename()`.
9. The file at *dirty random path* is deleted using the `unlink()` syscall.
10. The file at *clean path* is now encrypted.

5.6.3 Known issues

Although inotify provides an easy way to monitor for certain events, various tests have shown that especially folder creation is prone to drop events if they occur in a timely manner. A simple example will illustrate:

```
shell 1: $ inotifywait -e create -r /tmp
shell 2: $ cd /tmp && mkdir d d/d d/d/d d/d/d/d
```

which results in the following output in shell 1: `/tmp/ CREATE,ISDIR d`. The created directories under `d/` are not captured by inotify and therefore lost. The inotify man page[3] addresses this issue by advising newly created directories to be scanned recursively immediately after a watch descriptor for the top directory has been obtained. Further tests have shown that this improves the situation, but does not completely solve the problem. Especially when an application performs exhaustive folder and file manipulation[4], the immediate recursive scan of a newly created directory *may* takes place before sub-directories have been created, which will always end in a race condition. It should be noted that manual folder creation and deletion through a file manager is not affected by this circumstance as the time between two `mkdir()` calls is sufficient to install required watch descriptors.

Since the goal of the mountpoint watchdog is to monitor files and enforce encryption policies, unreliable file operation detection is unsatisfactory as events happening in unmonitored folders would simply be unnoticed. With inotify, the problem cannot be solved completely and no guarantee can be made that all events in a monitored directory are actually captured by *notifyd*.

5.6.4 fanotify

An alternative to inotify would be the use of *fanotify*, which provides the same interface as inotify, that is an `fanotify_init()` function that obtains a new file descriptor to a new fanotify instance and `fanotify_mark()` that adds a fanotify mark on a file system object. As for inotify, fanotify transfers events as shown in listing 5.13 to userspace by sending them on the file descriptor obtained by `fanotify_init()`. The application can then use `read()` on the descriptor to obtain the events.

Snippet 5.13: fanotify event structure

```
1  struct fanotify_event_metadata {
2      __u32 event_len;    /* length of the data for this event and the offset
3                             to the next event in the buffer */
4      __u8 vers;          /* fanotify API version number */
5      __u8 reserved;      /* reserved */
6      __u16 metadata_len; /* length of the structure */
7      __aligned_u64 mask; /* bitmask describing the event */
8      __s32 fd;           /* file descriptor for the object being accessed */
9      __s32 pid;          /* PID of the process that caused the event */
10 };
```

When using inotify, events contain a watch descriptor and a name which requires the application to keep track of mappings between path names and watch descriptors to resolve a given watch descriptor to a file system path. With fanotify however, the application receives an already opened file descriptor to the object itself. Using the

[3] obtained by running `man 7 inotify` in a shell.

[4] for example when unzipping a file with lots of files and subfolders

Linux system call `readlink()`, it is possible to obtain the file system path, hence rendering the required mapping redundant. Besides the file flags known from inotify, fanotify supports the additional flags `FAN_OPEN_PERM` and `FAN_ACCESS_PERM`. In addition, fanotify enables the application to mark directories with the `FAN_ACCESS_PERM` flags set. This allows an application to receive events when a file is to be opened by a process. The monitoring application can then send an appropriate reply as shown in listing 5.14 to fanotify using `write()` on the file descriptor obtained by `fanotify_-init()` *before* control is returned to the requesting application, therefore facilitating a basic access control mechanism.

Snippet 5.14: fanotify permission reply

```
1 struct fanotify_response {
2     __s32 fd;           /* file descriptor */
3     __u32 response;     /* access permission,
4                            either FAN_ALLOW or FAN_DENY */
5 };
```

As the goal of the watchdog is to prevent direct access to the root directory, a different approach is possible with *fanotify*. As can be seen in listing 5.13, the event data incorporates the PID of the application that has caused the event. A policy that enforces file encryption on the root directory would therefore be reducible to

```
if pid(sender) == pid(file system)
    return FAN_ALLOW;
else
    return FAN_DENY;
```

because file operations on the underlying root directory are always caused by the process running the FUSE file system. If the reported PID differs from the known PID of the FUSE file system, it is assured that the write operation did not pass through the mountpoint.

CHAPTER 6

Runtime performance

The following chapter will evaluate the FUSE file system in terms of performance. For this purpose, the command line utility `generate_random_files`, located in the `extra/` directory of `cfs` [Lor15] has been used to generate large quantities of random files. The command line tool has been run three times to generate enough files to simulate an ordinary office environment with a large amount of relatively small files and only a minor portion exceeding the 16 Megabytes mark. In terms of file encryption policies, the runtime tests simulate an initial encryption of the user's Documents folder. Moreover, a large amount of small files is an ideal basis for benchmarks as they cause a magnitude of copy operations to and from userspace (over the FUSE socket), but also mode switches from kernel to user mode. In addition, these operations also cause context switches from the processes that issued the system call to the FUSE file system and another one in the opposite direction.

Table 6.1 gives an overview of the randomly generated files that will serve as basis for further benchmarks. Benchmark times for *write* operations are generated using `cp -r source destination`. Additionally, prior to each test run, the commands

```
sync; echo 3 > /proc/sys/vm/drop_caches
```

were issued to free the pagecache and cached dentries and inodes to avoid getting tainted results due to caching.

6.0.1 Test setup

The following test results were generated on an ordinary office workstation. The system was composed of the following components:

CPU
: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 4 cores, 8 threads

Table 6.1: Sample files generated for benchmarking

			File size		
Dirs[1]	Files[2]	Levels	Minimum	Maximum	Total
16	48	3	4 KB	4 MB	40 GB
16	48	3	4 MB	16 MB	16 GB
16	48	3	16 MB	64 MB	16 GB
348	26931	3			80 GB

[1] Directories per level
[2] Files per (sub–) directory

Important CPU Instructions
as reported by `grep flags /proc/cpuinfo`: [...] ht pclmulqdq aes [...]

Memory
8 GB PC3–12800, 1600 MHz

HDD
Western Digital Blue WD10EZEX 1TB 7200 RPM 64MB Cache SATA 6.0Gb/s

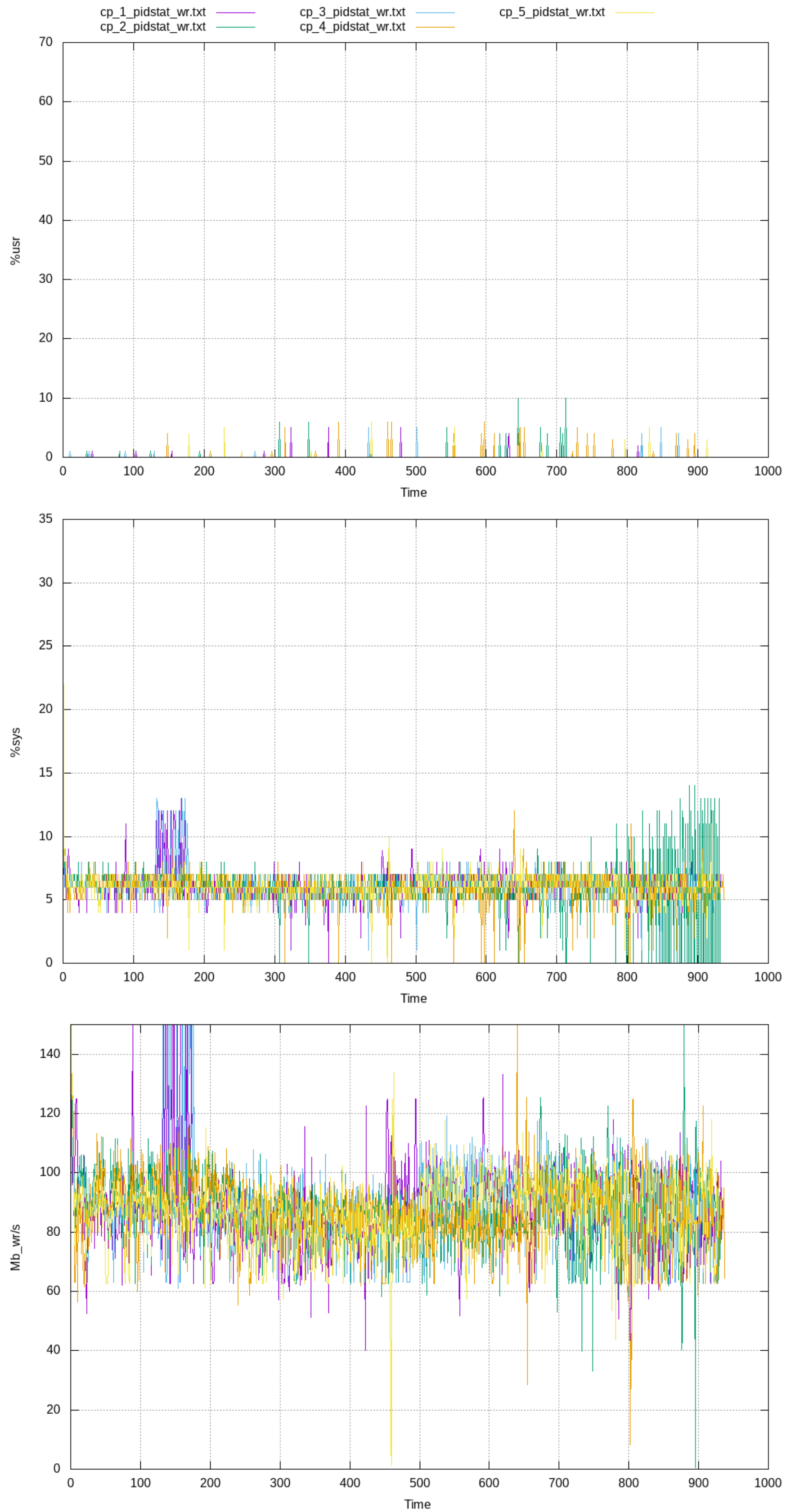
Operating System
Debian Linux, Kernel version 3.16

IO scheduler
Completely Fair Queuing (CFQ)

The 1 TB drive has been split into three partitions: 10GB for the base system and two 100GB ext4 formated partitions for the actual tests. The rest of the drive is unused.

The file system tests start with a plain copy operation and add one layer of complexity per test case. Additionally, each test case is run five times in order to gather consistent results. Data was recorded using the `pidstat` command. The keys in the graphs are comprised of the operation itself (`cp`), the application gathering the results (`pidstat`) and the round number. The mean values in Figure 6.5 were calculated by computing the average value of rounds one to five for each log entry of the `pidstat` command. This approach has been chosen because from Figures 6.2 to 6.4 the duration was almost identical and differences neglectable. The complete test case suit including the script that produced the output in the following graphs can be found in the `tests` directory of [Lor15].

Figure 6.1: Performance: raw



Performance result: raw

In this test case, `cfs` is not used at all. Instead a plain copy operation is measured in order to set the baseline for further tests.

As can be seen in figure 6.1, the copy operation brings no surprises. The load in user space (%usr) is mostly zero as all of the processing is done entirely in kernel space (%sys). The CPU is fully utilized on all cores and the data transfer rate oscillates at around 90 MB/s.

Performance result: FUSE passthrough

The first step is to use FUSE and simply pass all operations through to the underlying file system. Figure 6.2 shows the associated graphs. The first thing that should be noticed is that the speed does not drop at all. Instead, both the time spent in userspace as well as kernel space increases. This is a logical consequence of how FUSE works. When `cp` needs to write data to the destination, it is copied to kernel space, then sent over the FUSE device to `cfs` which in turns issues another system call that performs the same operation on the underlying file system and data needs to be copied once again. Since the CPU and memory are fast enough, the performance penalty is effectively non-existent.

Performance result: FUSE memcpy–encryption

The test case with the results shown in figure 6.3 introduces the first layer of encryption. Although it is just a simple `memcpy` operation, it introduces additional logic in `cfs`: write operations are split into chunks of 512 byte. For each block a initialization vector is generated and furthermore, non–aligned access to files is aligned to block boundaries. This could potentially cause a read operation on the underlying file to read a partial block, overwrite them with new data and write the result back to disk. Therefore, the userspace load rises insignificantly due to `memcpy`, system load increases due to additional file reads and copying from underlying files.

Performance result: FUSE AES256–CBC–encryption

The last test case performs AES–CBC encryption of all data passing through the file system. While the load in kernel space does not increase significantly compared to the previous test case (because the same logic applies with regards to block read–back from the underlying file), the encryption operation dramatically increases the load in user space. This is not a problem on a workstation, but on a battery–driven device like a laptop it has a significant impact on battery life. Moreover, as can be seen in figure 6.4, the write speed only drops marginally. This is an indicator that the CPU is fast enough to perform encryption.

Figure 6.2: Performance: FUSE passthrough

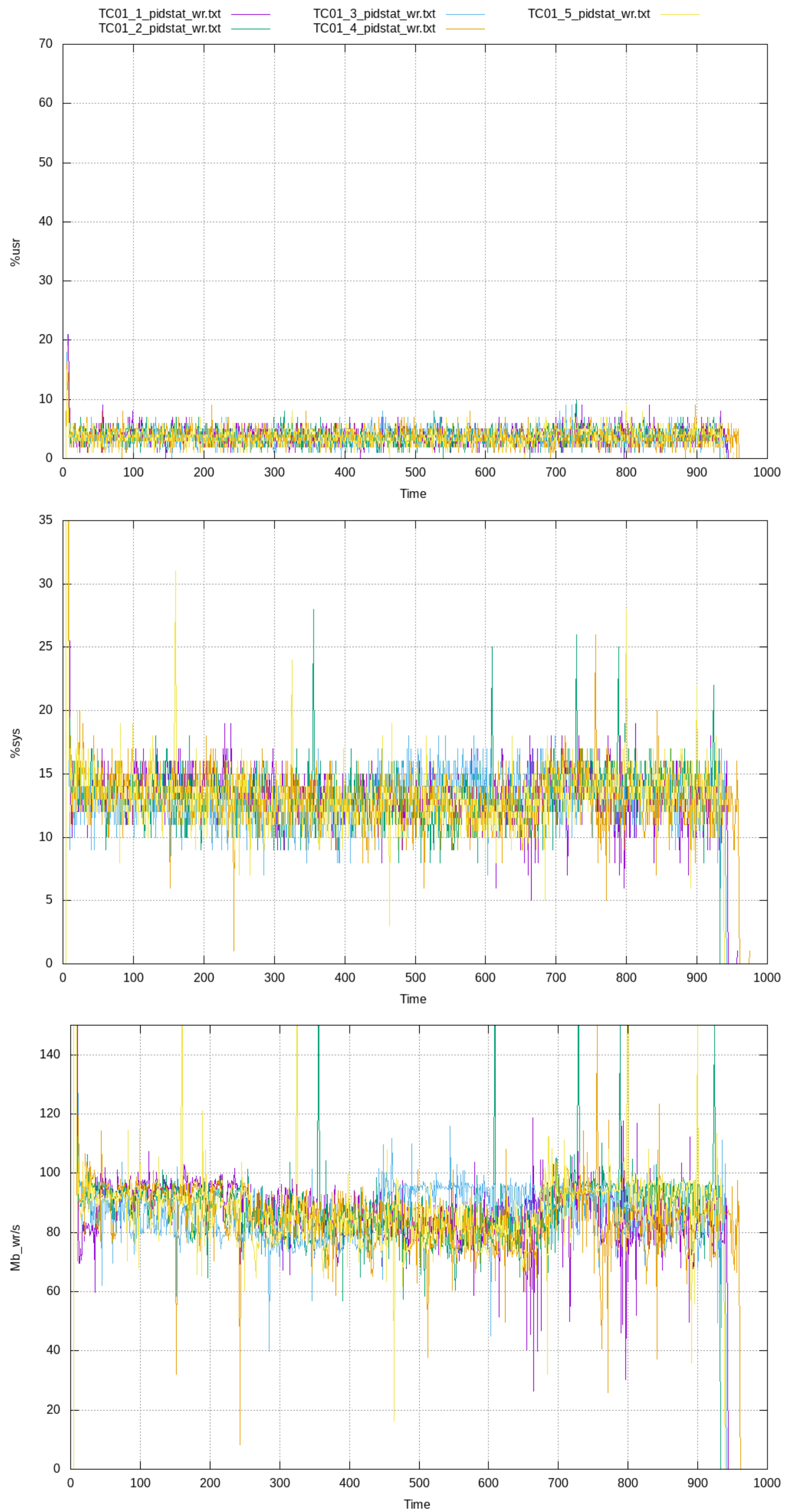


Figure 6.3: Performance: FUSE memcpy–encryption

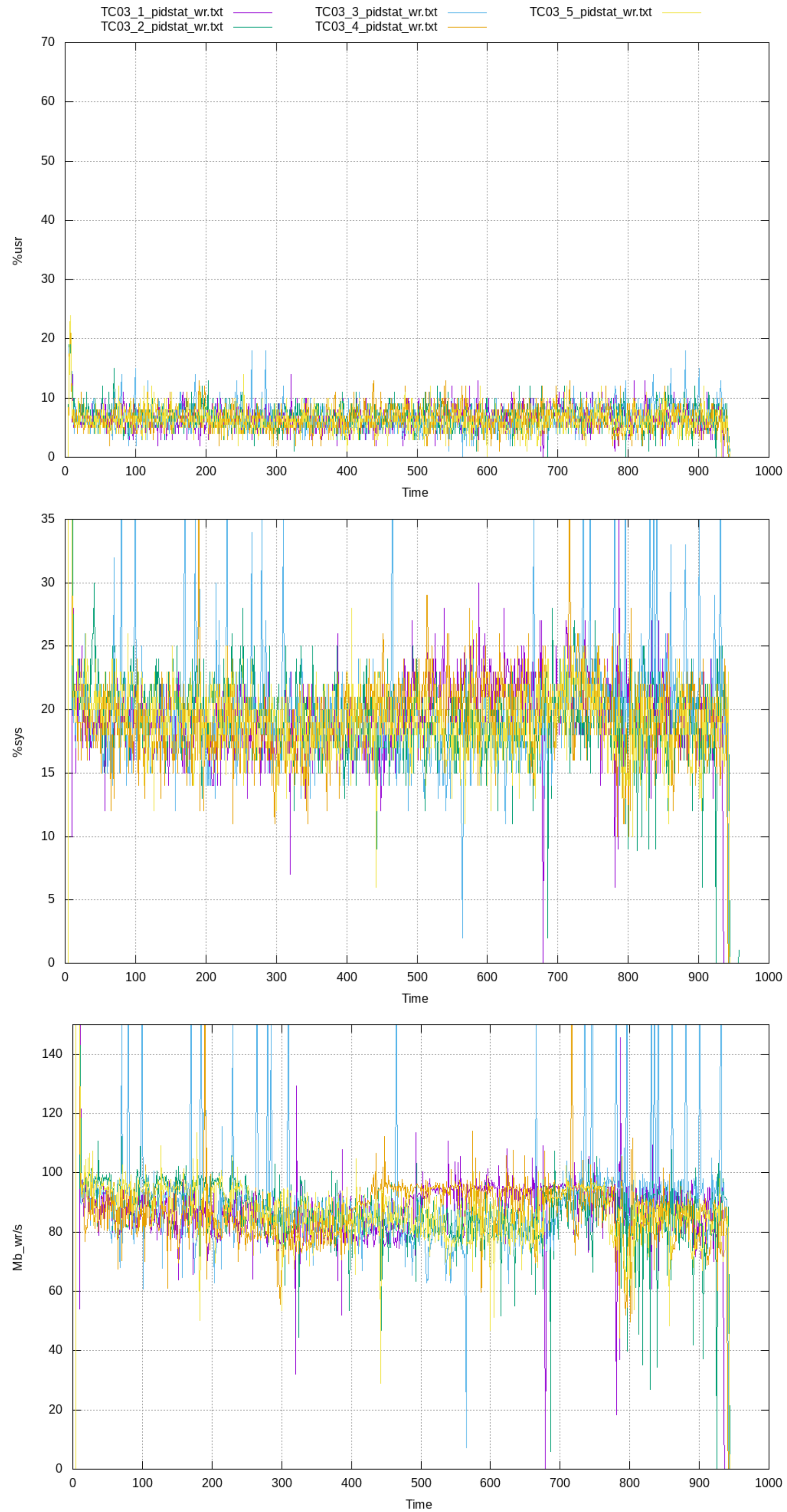
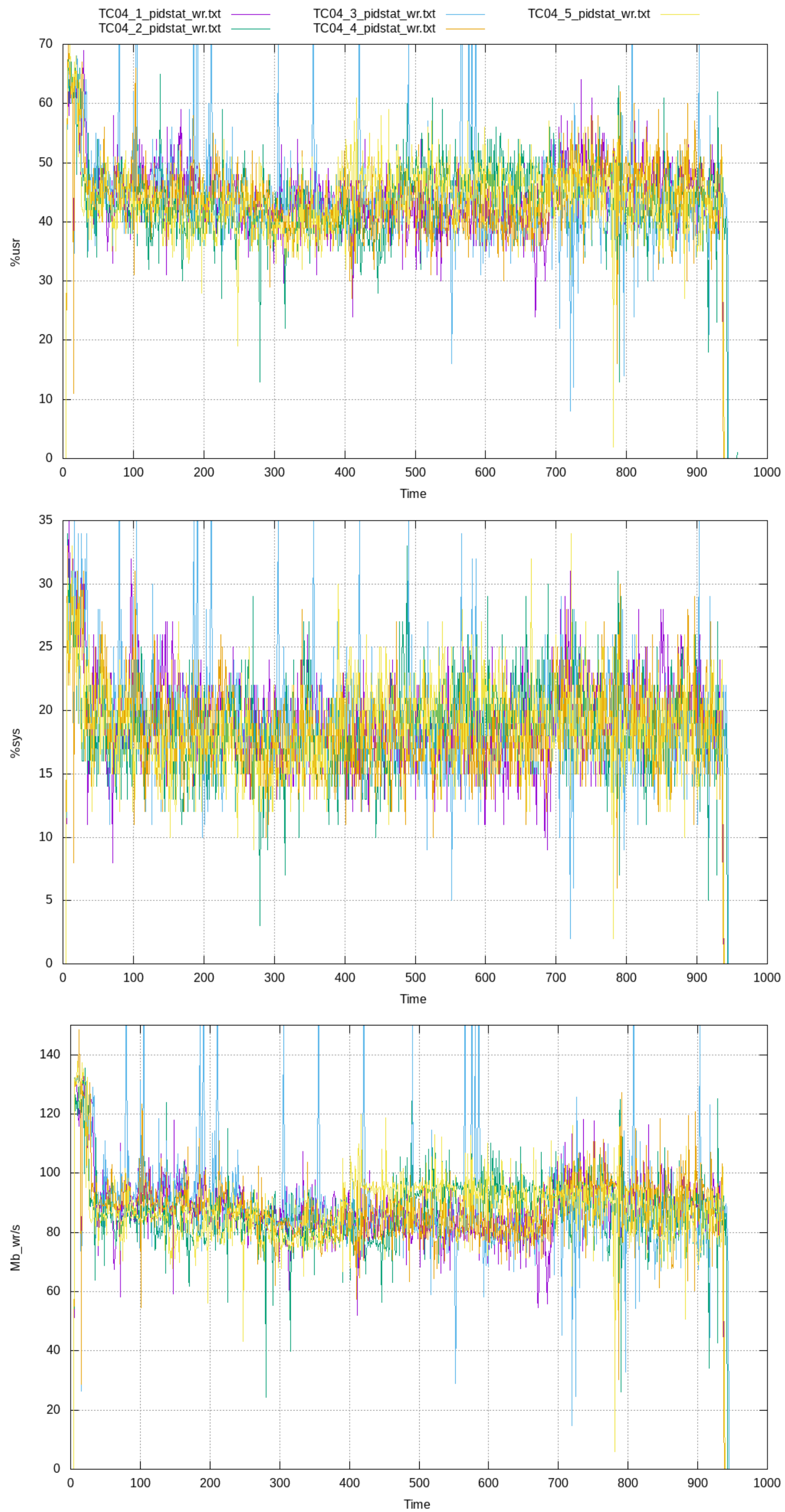


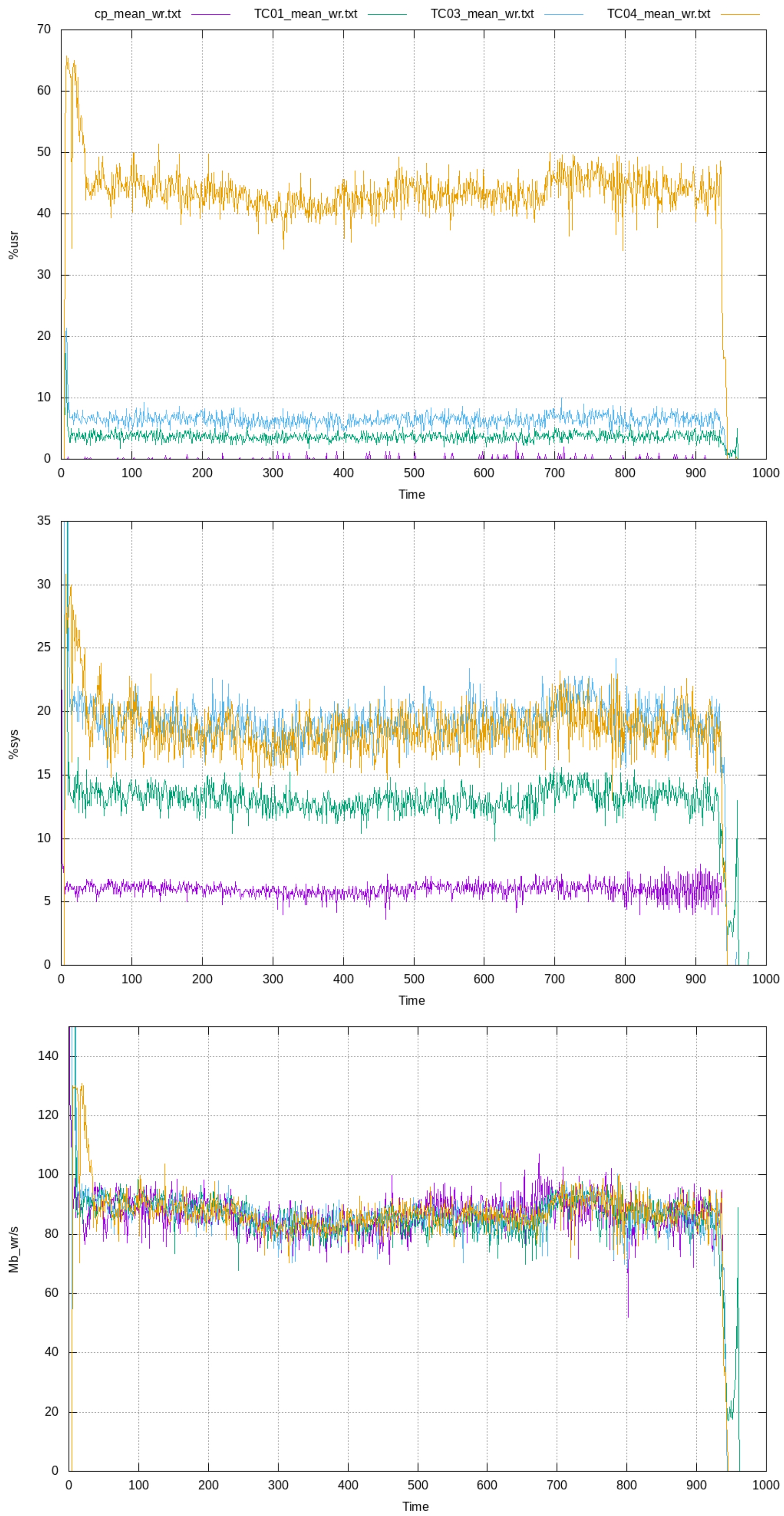
Figure 6.4: Performance: FUSE AES256–CBC encryption



Performance result: Comparison

Figure 6.5 shows the mean values per test case in an all–in–one plot for easier comparison. As stated in the beginning of this chapter, the mean values are simply the sums over all y axis values divided by the number of runs (five in this case). Due to similar duration of the test cases, this should not be a problem but evens out any spikes that are caused by other processes, caching or hardware peculiarities.

Figure 6.5: Performance: Comparison



CHAPTER 7

Conclusion

The last chapters have shown that it is definitely possible to provide all the software that is needed in order to achieve file format compatibility across the major operating systems in use today. Starting from a preliminary overview of available and approved cryptographic algorithms and modes, it was suggested that newer releases of the file system should take advantage of Intel's new instruction and switch to modes like CTR or XTS to speed up the encryption process. Especially modes that do no require padding should be preferred as they do not require the file system to rewrite the padding information on the hard drive – an operation that definitely has a negative impact on performance as it requires the drive to mechanically reposition its heads. Although the benchmarking results as presented in the previous chapter were expected to be better, they provide a reference implementation for further improvements nevertheless.

Regarding persistent storage on Linux using one of the default key stores, one should mind the specifics and restrictions of the software. As has been pointed out, it might pay off to even analyze their source code and identify weaknesses. This circumstance however simultaneously shows the significant advantage Linux has over proprietary software. Source code for most of the components is commonly available and can be used to identify problems.

Introducing various inter-process communication facilities not only available on Linux, but all POSIX systems, a high level architecture has been developed that serves as a common entry point for access to users' key material. Having this facility in place, it does not really matter any more where requests for key material come from. For this paper, it is a file system running entirely in userspace, but instead or even simultaneously it could also be an in–kernel file system.

The FUSE–based file system has been developed with full compatibility to already existing software for the Windows and Mac OSX operating system in mind. Although this aspect was not explicitly in scope of this research, Sophos expressed their interest

in this feature for the proof–of–concept implementation – primarily for demo purposes.

Because the compatibility layer is basically limited to the file format of the header and therefore established quite easily, focus had been shifted to other crucial areas such as secure handling of sensitive information while it is being kept in memory. The code therefor is mostly based on the Linux kernel's implementation of the SLAB allocator.

A different area investigated was proper caching of metadata and file size correction. Both are equally important as operations on a file system occur quite often and parsing a rather complex header structure needlessly slows down the user's experience.

A problem identified but not fully resolved it the fact that users can bypass a FUSE mountpoint and therefore file encryption in general. As the file system implementation merely is a prototype implementation, it is suggested that the file system should be moved to the kernel's virtual file system layer for production use. The benefit of doing so would be increased performance while still being able to perform encryption on a per file basis. Additionally, it brings the advantage of the cryptographic file system being stacked on top of an ordinary file system, hence making it impossible for userspace to bypass encryption.

Bibliography

[BC05] D.P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, Boston, Massachusetts, 1994. USENIX Association.

[CDFT98] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP Message Format. RFC 2440 (Proposed Standard), November 1998. Obsoleted by RFC 4880.

[Dwo01] M. Dworkin. Recommendation for Block Cipher Modes of Operation – Methods and Techniques. Special Publication 800-38A, National Institute of Standards and Technology, Dec 2001.

[Dwo07] M. Dworkin. Recommendation for Block Cipher Modes of Operation – Galois/Counter Mode (GCM) and GMAC. Special Publication 800-38D, National Institute of Standards and Technology, Nov 2007.

[Dwo10] M. Dworkin. Recommendation for Block Cipher Modes of Operation: The XTS–AES Mode for Confidentiality on Storage Devices. Special Publication 800-38E, National Institute of Standards and Technology, Jan 2010.

[Fru11] Clemens Fruhwirth. LUKS On–Disk Format Specification, version 1.2.1. http://wiki.cryptsetup.googlecode.com/git/LUKS-standard/on-disk-format.pdf, Oct 2011. Accessed: 2013-01-05.

[FUS11] Filesystem in Userspace. http://fuse.sourceforge.net, Jul 2011. Accessed: 2013-12-30.

[GK12] Shay Gueron and Michael E. Kounavis. Intel® Carry–Less Multiplication Instruction and its Usage for Computing the GCM Mode. http://download-software.intel.com/sites/default/files/article/165685/clmul-wp-rev-2.01-2012-09-21.pdf, Sep 2012. Accessed: 2013-12-29.

[Gue12] Shay Gueron. Intel® Advanced Encryption Standard (AES) New Instructions Set. http://download-software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf, Sep 2012. Accessed: 2013-12-29.

[Hal05] Michael. A. Halcrow. ecryptfs: An Enterprise–class Cryptographic Filesystem for Linux. In *Proceedings of the Linux Symposium*, pages 201–218, 2005.

[Hal07] Michael. A. Halcrow. eCryptfs: a Stacked Cryptographic Filesystem. http://www.linuxjournal.com/article/9400, Apr 2007. Accessed: 2013-01-06.

[Hou09] R. Housley. Cryptographic Message Syntax (CMS). RFC 5652 (INTERNET STANDARD), September 2009.

[IEE08] IEEE. IEEE Standard for Cryptographic Protection of Data on Block–Oriented Storage Devices. *IEEE Std 1619-2007*, pages c1–32, 2008.

[Jona] M. Jones. Anatomy of the Linux kernel. http://www.ibm.com/developerworks/library/l-linux-kernel. Accessed: 2013-12-04.

[Jonb] M. Jones. Anatomy of the Linux virtual file system switch. https://www.ibm.com/developerworks/linux/library/l-virtual-filesystem-switch. Accessed: 2013-12-04.

[Kal00] B. Kaliski. PKCS #5: Password–Based Cryptography Specification Version 2.0. RFC 2898 (Informational), Sep 2000.

[KRSS14] R. Kissel, A. Regenscheid, M. Scholl, and K. Stine. Guidelines for Media Sanitization. Special Publication 800-88 Rev. 1, National Institute of Standards and Technology, Dec 2014.

[LM08] Moses Liskov and Kazuhiko Minematsu. Comments on XTS-AES, Sep 2008.

[Lor15] O. Lorenz. cfs: A Sophos Safeguard Enterprise compatible file system for Linux – accompanying material. [CDROM], Oct 2015.

[LRW02] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 31–46. Springer Berlin Heidelberg, 2002.

[Mor03] James Morris. The Linux Kernel Cryptographic API. http://www.linuxjournal.com/article/6451, Apr 2003. Accessed: 2014-01-04.

[MV05] David A. McGrew and John Viega. The Galois/Counter Mode of Operation (GCM). http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf, May 2005. Accessed: 2013-12-28.

[NSA14] National Security Agency. NSA/CSS Policy Manual 9–12. Policy Manual 9–12, Central Security Service, Dec 2014.

[oST01] National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. Accessed: 2013-12-29.

[Pac08] V.K. Pachghare. *Cryptography and Information Security*. PHI Learning, 2008.

[Pal05] J. Palmieri. Get on D–BUS. http://www.redhat.com/magazine/003jan05/features/dbus, Jan 2005. Accessed: 2013-12-12.

[PCL+03] H. Pennington, A. Carlsson, A. Larsson, S. Herzberg, S. McVittie, and D. Zeuthen. D–Bus specification. http://dbus.freedesktop.org/doc/dbus-specification.html, Sep 2003. Accessed: 2013-12-12.

[Pro] The OpenSSL Project. Welcome to the OpenSSL project. https://www.openssl.org. Accessed: 2013-12-16.

[Ray03] E.S. Raymond. *The Art of UNIX Programming*. Addison-Wesley professional computing series. Pearson Education, 2003.

[RG10] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM.

[Sch84] Bob W. Scheifler. window system X. http://www.talisman.org/x-debut.shtml, Jun 1984. Accessed: 2014-11-25.

[Sch96] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1996.

[Smi11] R.E. Smith. *Elementary Information Security*. Jones & Bartlett Learning, 2011.

[TBBC10] M. S. Turan, E. Barker, W. Burr, and L. Chen. Recommendation for Password–Based Key Derivation. Special Publication 800-132, National Institute of Standards and Technology, Dec 2010.

[Vel08] Sainath S. Vellal. A Device Mapper based Encryption Layer for TransCrypt. Master's thesis, Indian Institute of Technology Kanpur, Jun 2008.

APPENDIX A

SafeGuard Enterprise file format

The following tables describe the on–disk header format used by SafeGuard Enterprise file encryption. The header is creating during file creation and written to disk. The only information altered during the lifetime of a file is the padding header at the end of the header as it always reflects that last, possibly partial block of the file.

The header consists of three distinct sections. Table A.1 describes a key that is used to initialize a block cipher will all relevant information.

As mentioned before, the last block of the file most likely does not end on a block boundary. Therefore, padding information has to be kept to reconstruct the block upon alteration. Table A.2 shows the padding header used to maintain that kind of information. It should be noted that the header size is fixed. It is therefore not possible to use a different block size than the predefined 512 byte blocks.

The largest part of the header is located right at the beginning of a file. The main header is used to identify SafeGuard encrypted files by using various magic values, but also to describe the cipher that produced the file along with a series of cipher keys as described in detail in Table A.1.

Table A.1: SGN key format

Offset	Size (b)	Description
0	258	GUID of the key encryption key
258	2	Unused
260	4	Size of wrapped data encryption key
264	100	Wrapped data encryption key
364	8	Checksum
372	8	Unused

Table A.2: SGN padding header format

Offset	Size (b)	Description
3560	4	Padding identifier
3564	4	Unused
3568	8	Offset
3576	4	Padding size
3580	4	Flags
3584	512	Data

Table A.3: SGN file format

Offset	Size (b)	Description
0	128	Copyright string
Base structure		
128	4	Base structure identifier
132	4	Size of base structure
136	4	Header version number
140	4	Owner identifier
144	4	Encryption mode
148	4	Block size
152	8	Unused
Key storage area		
160	4	Key storage area identifier
164	4	Size of key storage area
168	32	Initial vector
200	8	Unused
208	2	Number of key entries
210	2	Filler bytes
212	*	Array of keys (A.1)
3560	536	Unused or padding header (A.2)

Glossary

A

AES

A block cipher designed by Vincent Rijmen and Joan Daemen and winner of the AES selection process held by NIST in 2001. Today it is the most encountered cipher.

API

An Application Programming Interface (API) is usually a collection of functions and data types exported by libraries that other applications and libraries can use to access certain functionality of the library that implements the interface..

Asymmetric cryptography

Asymmetric cryptography or public–key cryptography denotes a process where each party is in possession of a key pair. The public part should be distributed among the recipients, whereas the private part remains secret and is used to decrypt messages previously encrypted with the associated public key.

C

CBC

A cipher mode for block ciphers. It uses the output of the previous block and XORs it with the current block as input for the cipher during encryption.

CLI

A type of User–Computer–Interface that is solely based on textual input and output.

CTR

A mode of operation for block ciphers. The ciphertext stream is produced by XORing input with an encrypted keystream. This mode basically turns a block cipher into a stream cipher.

D

DEK

The key used for symmetric cryptography.

E

ECB

A mode of operation for block ciphers. The ciphertext stream is produced by simply concatenating subsequent output blocks of the cipher.

F

FHS

The Filesystem Hierarchy Standard, specified and maintained by the Linux Foundation provides information used by Unix distribution developers, package developers and system implementers about directory structures and contents.

G

GCM

A mode of operation for block ciphers that provides confidentiality and integrity checking by maintaining an internal hash value that can be used upon decryption to verify the integrity of the regained plaintext..

GNOME

A desktop environment written in C, C++, Python, Vala, Genie, JavaScript.

GPG

GPG is an open–source implementation of the OpenPGP standard as defined by RFC4880 and provides encryption and digital signatures for data and communication. It also includes key management and is widely used, especially among the Linux community.

I

IV

Used in cipher chains as input for the first block $\oplus$ operation, as no previous block exists.

K

K Desktop Environment

Precisely KDE Software Compilation denotes a desktop environment written in C++ (Qt) and C.

KEK

Key used to encrypt the DEK.

Kernel

The core of the operating system. It has complete control over the system, controls the hardware and interacts indirectly with the user for various tasks (e.g process creation, file I/O, . . .).

L

Linux distribution

Collection of packages combined with a Linux kernel release.

M

MMU

Hardware unit that performs translation from virtual addresses to physical addresses as they pass through.

P

Package Manager

A collection of tools to install, remove or upgrade packages on a Linux distribution.

Package repository

Collection of packages.

PBKDF

Applies a pseudo–random function to the user input along with a salt multiple times – typically >= 4096 as this number provides a good balance between security benefits and speed penalties.

PID

A positive number used to uniquely identify a process.

POSIX

The Portable Operating System Interface is a family of standards specified by the IEEE to maintain compatibility between operating systems.

S

SHA-1

A cryptographic hash function designed by the NSA. It produces a 160 bit output after 80 rounds of Merkle–Damgård construction.

Symmetric cipher

Symmetric ciphers use the same key for encryption and decryption. Unlike Asymmetric cryptography, it is of uttermost importance that the secret key remains secret as each individual in possession of the key is able to decrypt the ciphertext.

V

VFS

An API layer in the Linux kernel that acts as an abstraction layer and dispatcher for file system operations..

W

W Window System

Precursor of the X, originally developed at Stanford University.

X

X

A client–server specification for device-independent windowing operations on bitmap display devices.

XTS

A cipher mode for block ciphers originally designed for hard drive encryption where the sector size does not necessarily have to be dividable by the block size of the cipher. Today it is widely supported including dm–crypt, geli or Apple's FileVault.

List of Abbreviations

A

AES

Advanced Encryption Standard. See *Glossary:* AES.

API

Application Programming Interface. See *Glossary:* API.

C

CBC

Cipher Block Chaining mode. See *Glossary:* CBC.

CLI

Command Line Interface. See *Glossary:* CLI.

CTR

Counter Mode. See *Glossary:* CTR.

D

DEK

Data Encryption Key. See *Glossary:* DEK.

DES

Data Encryption Standard.

E

ECB

Electronic codebook mode. See *Glossary:* ECB.

ext3

Third Extended Filesystem.

F

FAT

File Allocation Table.

FHS

Filesystem Hierarchy Standard. See *Glossary:* FHS.

FUSE

Filesystem in User–Space.

G

GCM

Galois Counter Mode. See *Glossary:* GCM.

GPG

GNU Privacy Guard. See *Glossary:* GPG.

GUID

Globally Unique Identifier.

H

HAL

Hardware Abstraction Layer.

I

IPC

Interprocess Communication.

IV

Initialization Vector. See *Glossary:* IV.

K

KEK

Key Encryption Key. See *Glossary:* KEK.

L

LKM

Loadable Kernel Module.

LUKS

Linux Unified Key Setup.

M

MMU

Memory Management Unit. See *Glossary:* MMU.

N

NIST

National Institute of Standards and Technology.

P

PBKDF

Password Based Key Derivation Function. See *Glossary:* PBKDF.

PID

Process Identifier. See *Glossary:* PID.

POSIX

Portable Operating System Interface. See *Glossary:* POSIX.

R

RAID

Redundant Array of Independent Disks.

S

SHA-1

Secure Hash Algorithm. See *Glossary:* SHA-1.

U

UID

user identifier.

V

VFS

Virtual File System. See *Glossary:* VFS.

X

X

X Window System. See *Glossary:* X.

XML

Extensible Markup Language.

XTS

XEX-based tweaked-codebook mode with ciphertext stealing. See *Glossary:* XTS.

Oliver Lorenz

PERSONAL INFORMATION

Date of Birth	**October 1987**.
Place of Birth	**Rohrbach, Upper Austria, Austria**.
Citizenship	**Austrian**.
Marital status	**Single**.

EDUCATION

1994 – 1998	**Primary school**, Julbach.
1998 – 2002	**Grammar school**, Rohrbach.
2002 – 2007	**Federal Higher Technical Institute**, Neufelden. A–levels in automation technology

CIVILIAN SERVICE

2007 – 2008	**Civilian service**, State hospital Rohrbach.

HIGHER EDUCATION

2008 – 2011	**Carinthia University of Applied Sciences**, Campus Klagenfurt, Bachelor of Science in Network engineering (with distinction).
since 2011	**Johannes Kepler University**, Linz, Master of Science in Networks & Security.

MASTERS THESIS

Title	*cfs: A Sophos SafeGuard Enterprise compatible file system for Linux*
Supervisors	Doz. Dr. Gerhard Eschelbeck & Dipl.–Ing. Dr. Rudolf Hörmanseder
Description	This thesis explored the possibilities of file encryption on Linux resulting in a proof–of–concept implementation for Sophos.

INTERNSHIPS

08/2004 – 09/2004	**Woess Blockhäuser**, Julbach. Production
08/2008 – 09/2008	**Voest Alpine Stahl AG**, Linz. Production

03/2010 – 06/2010	**Fa. Alexander Hoffmann**, Klagenfurt. Network support
02/2011 – 06/2011	**Fa. neo-IT**, Klagenfurt. Network engineer
08/2011 – 09/2011	**Vöest Alpine Stahl AG**, Linz. Production
08/2012 – 09/2012	**Vöest Alpine Stahl AG**, Linz. Production
08/2013 – 09/2013	**Vöest Alpine Stahl AG**, Linz. Production

PROFESSIONAL EXPERIENCE

since 2014	**Sophos GmbH**, *Enduser Security Group*, Linz, Development Engineer.

LANGUAGE SKILLS

German	Mother tongue
English	fluent

IT SKILLS

Operating systems	Windows Servers, Linux, Mac OS X, OpenBSD, Cisco IOS, HP Procurve (basics), Juniper JUNOS (basics)
Programming	C, C++, Objective-C, Swift, Java
Scripting	Bash, PHP
Typesetting	LaTeX, HTML, CSS (basics)
Databases	MySQL, SQLite, Oracle
Networking	Advanced Routing, switching, network design, service administration (Linux)
IT security	Penetration testing, vulnerability identification and response, encryption technologies, fundamentals of computer forensics, malware analysis (basics)

INTERNATIONAL EXPERIENCE

07/2010 – 12/2010	**Exchange semester**, Universiti Kuala Lumpur. Malaysian Institute of Information Technology

Affidavit

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die an gegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtliche und sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

I hereby declare under oath that the submitted Master's degree thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented.

Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

14 October 2016

(Oliver Lorenz)