# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

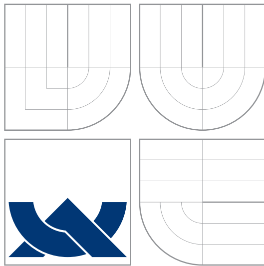

# REDIRFS - APPLICATION ON DATA ENCRYPTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

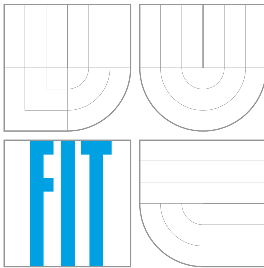

AUTOR PRÁCE — PAVEL ZŮNA
AUTHOR



BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

# REDIRFS - APLIKACE NA ŠIFROVÁNÍ DAT

REDIRFS - APPLICATION ON DATA ENCRYPTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PAVEL ZŮNA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ KAŠPÁREK

BRNO 2009

## Abstrakt

Tato práce se v prvé řadě zaměřuje na návrh a implementaci šifrovacího filtru pro Redirecting Filesystem Framework (dále jen RedirFS) v podobě modulu linuxového jádra. Zpočátku se věnuje problematice kryptografie a šifrování. Dále rozebírá hlavní části RedirFS a možnosti jejich rozšíření. Potom co je čtenář seznámen s důležitými pojmy, následuje hlavní část práce. Nejdříve je nastíněn samotný návrh filtru, zvolené postupy - jejich výhody a omezení. Poslední kapitola je plně věnována cipherflt: vzorové implementaci šifrovacího filtru.



## Abstract

This thesis focuses on design and implementation of a cryptographic filter for the Redirecting Filesystem Framework (RedirFS for short) in the form of a Linux kernel module. It starts by giving the reader a general overview of modern cryptography and encryption. Before diving into the filter specifics, it takes a tour of the most important features of RedirFS. When done with introductory chapters, it describes the design choices made considering their benefits and limitations. The final chapter is completely devoted to cipherflt: a proof of concept implementation of the cryptographic filter.



## Klíčová slova

Linux, jádro, modul linuxového jádra, VFS, souborový systém, RedirFS, kryptografie, šifrování, CryptoAPI

## Keywords

Linux, kernel, linux kernel module, VFS, file systém, RedirFS, cryptography, encryption, CryptoAPI

## Citace

Pavel Zůna: RedirFS - application on data encryption, bakalářská práce, Brno, FIT BUT, 2009

# RedirFS - application on data encryption

## Statement

I hereby state that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged. This thesis was elaborated under the supervision of Ing. Tomáš Kašpárek.

. . . . . . . . . . . . . . . . . . . . . . .
Pavel Zůna
May 20, 2009


## Acknowledgements

I would like to thank the following people: my supervisor Ing. Tomáš Kašpárek for fast feedback, František Hrbata for advice regarding RedirFS, my dad for proofreading, Martin Nagy for providing me with a document template, the rest of my family for support and my friends for listening to my constant ranting about how I'll never make it in time. Thank you.



© Pavel Zůna, 2009.

*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Contents


**1 Prologue** **3**

**2 Cryptography** **4**
- 2.1 Encryption . . . 4
  - 2.1.1 Algorithms . . . 4
  - 2.1.2 Prominent block ciphers . . . 5
  - 2.1.3 Modes of operation . . . 6

**3 Redirecting Filesystem Framework** **7**
- 3.1 Filter developement . . . 7
  - 3.1.1 Registration . . . 7
  - 3.1.2 Path lists . . . 8
  - 3.1.3 Filter operations . . . 9
  - 3.1.4 VFS operations . . . 10
  - 3.1.5 RedirFS data . . . 11
  - 3.1.6 sysfs interface . . . 12
- 3.2 Extending RedirFS . . . 12
  - 3.2.1 New supported operations . . . 12
  - 3.2.2 Future extension ideas . . . 13

**4 Cryptographic filter design** **14**
- 4.1 Goals . . . 14
- 4.2 Encryption mechanism . . . 14
  - 4.2.1 Keys and re-keying . . . 15
  - 4.2.2 Encrypting file content . . . 15
  - 4.2.3 Initialization vectors . . . 15
  - 4.2.4 Encrypting file names . . . 16
  - 4.2.5 Tagging encrypted files . . . 16

**5 cipherflt** **18**
- 5.1 CryptoAPI . . . 18
  - 5.1.1 Using block ciphers . . . 18
  - 5.1.2 Encrypting pages . . . 19
- 5.2 Data structures . . . 20
  - 5.2.1 Trailer . . . 20
  - 5.2.2 Inode data . . . 21
  - 5.2.3 Block . . . 21
  - 5.2.4 Context data . . . 22

5.3 VFS operations . . . 22
5.3.1 open . . . 22
5.3.2 release . . . 23
5.3.3 write . . . 23
5.3.4 readpages . . . 24
5.3.5 writepages . . . 24
5.3.6 d_iput . . . 25
5.4 Unsolved issues . . . 25

**6 Conclusion** **27**

**A Contents of enclosed CD** **30**

# Chapter 1

# Prologue

Data encryption and cryptography in general play an important role in information security. In today's world, where information is a valuable resource that needs protection, cryptography has become a part of our everyday lives. We come in contact with it unknowingly when using mobile phones, credit cards or the Internet. One of its many uses is the encryption of disk based files in computer systems.

The main goal of this thesis is to design and implement a Linux kernel module for transparent encryption on top of existing file systems. This is to be achieved by replacing operations of the Virtual Filesystem (VFS for short) layer in the Linux kernel using the Redirecting Filesystem Framework.

**Chapter 2** explores modern cryptography with focus on its aspects useful in disk based file encryption. It gives a general overview of the most popular algorithms in use today with references to further reading.

**Chapter 3** introduces the Redirecting Filesystem Framework and serves as a guide how to use and extend the current version. It also brings some new ideas for extending the framework's functionality.

**Chapter 4** evaluates different approaches to file encryption by discussing their advantages and disadvantages.

**Chapter 5** is completely devoted to the implementation of a proof of concept encryption module based on the design outlined in chapter 4.

The reader is expected to have a general understanding of Linux kernel development and VFS. These subjects were intentionally omitted as many publications are freely available. The author recommends the book Understanding the Linux Kernel[1], which covers both subjects extensively. A detailed overview of VFS operations can be found in F. Hrbata's Master thesis[6].

# Chapter 2

# Cryptography

Cryptography is the study of information concealment. Together with cryptanalysis, which is the study on how to compromise cryptographic mechanisms, they form the discipline of cryptology. Until modern times, it referred almost exclusively to encryption and is still used as a synonym for it in many contexts. Unless indicated otherwise, information in this chapter has been compiled from the RSA Laboratories' CryptoFAQ[8].

This chapter focuses on the main concepts of cryptography and encryption in particular.

## 2.1 Encryption

Encryption is the cryptographic term for an algorithmic process of data transformation with the purpose of concealing confidential information from unauthorized parties. The reverse process is called decryption. However, the term encryption usually refers to both of them. In cryptography, the untransformed data are called plaintext without any implication concerning their format, whereas the transformed data are called ciphertext.

### 2.1.1 Algorithms

Algorithms used for performing encryption and decryption of data are called ciphers. They make use of a secret key when generating ciphertext. Different keys produce different results and the original plaintext can only be retrieved with the right key. By the type of key used, ciphers can be divided into two basic groups:

- symmetric ciphers
- asymmetric ciphers

In symmetric ciphers, the same key is used both for encrypting and decrypting data. Therefore all authorized parties must share one secret key. Because of this fact, the group of symmetric ciphers is referred to as shared-key cryptography.

On the contrary, asymmetric ciphers use two separate keys. One of them made public, which enables anyone to encrypt information and the other private used for decryption. Hence this group of ciphers is called public-key cryptography.

Common use for public-key cryptography is communication, where senders encrypt confidential messages by the receiver's public key making it practically impossible for anyone

else to read as long as the private key remains unexposed, effectively protecting the communicating parties from eavesdropping. Prominent examples of public-key ciphers are the RSA and DSA algorithms.

Shared-key cryptography can also be used for communication, but the sender and receiver have to agree on the key used without anyone else finding out. Sophisticated key exchanging mechanisms are often required to keep the key secure. On the other hand, shared-key ciphers are generally faster than their public-key counterparts making them better suited for cases where key exchange is not an issue. They can be further divided into:

- stream ciphers
- block ciphers

Stream ciphers work with continuous streams of data. Individual digits (in practice bits or bytes) are processed one by one according to the current state. They are designed for encrypting data of arbitrary length unknown beforehand such as voice transmissions or media streaming. The main disadvantage of stream ciphers is that their keys (called keystreams) can never be reused. A new keystream has to be generated for every session.

Block ciphers operate on fixed-length chunks of data called blocks. They transform a block of plain text using the provided key into a block of ciphertext of the same length. Because block ciphers are best suited for the purpose of this thesis, the remaining sections of this chapter will be devoted to them.

### 2.1.2 Prominent block ciphers

In this section, we're going to take a look at the basic properties of the most widespread freely available block ciphers in use today.

**Advanced Encryption Standard**
AES is an encryption standard adopted by the U.S. Government in 2001 consisting of three encryption algorithms AES-128, AES-192 and AES-256. They operate on blocks of 128 bits and use keys of 128, 192 or 256 bits respectively.[13] AES-256 is the default encryption algorithm used by TrueCrypt.[19]

**Data Encryption Standard**
DES is another encryption standard developed in the 1970s. It's original form worked with blocks of 64bits and keys of 56bits. With today's state of computing, this form is considered insecure due to its small key size and only the improved variant called Triple DES is used in practice, being nothing more than a triple application with three different keys of the original DES algorithm on each block of data.[14]

**Blowfish**
Blowfish is a public domain block cipher designed in 1993 operating on blocks of 64 bits wit keys of 32 to 448 bits in steps of 8 bits, default being 128 bits.[2] It offers reliable and fast encryption with small memory footprint and it is the algorithm of choice in the cryptfs project.[4]

**Twofish**
Twofish is closely related to Blowfish and was designed in 1998 as its successor, but didn't achieve the same popularity. It uses blocks of 128 bits and keys of 128, 192 or 256 bits. Twofish was one of the finalists competing for standardization with AES.[3]

### 2.1.3 Modes of operation

As stated in section 2.1.1, block ciphers operate on fixed-length blocks of data. Encrypting blocks with the same content using the same key always produces the same result. This behavior is undesirable for plaintext comprising several blocks as it reveals patterns in it's structure leading to highly increased security risks. This disadvantage of block ciphers is reduced by a concept called modes of operation.

Modes of operation are a way of chaining the application of block ciphers on an arbitrary number of blocks by making successive blocks dependent on previously encrypted ones. To provide some randomization to the process and a further increase in security, a dummy block called the initialization vector (often reffered to as IV) are used when encrypting the first real block. Let's take a look at some of the most common modes of operation.[15]

**Electronic Code Book**
ECB isn't a real mode of operation. Plaintext is simply divided into blocks and each of them is encrypted independently. No initialization vector is used.

**Cipher-Block Chaining**
CBC was designed to hide patterns in encrypted messages. The bits of each plaintext block are XORed with the bits of the previous ciphertext block before being encrypted. Plaintext, whose length is not a multiple of block size, needs to be padded when using this mode.

**Cipher Feedback**
In CFB mode the previous ciphertext block is encrypted and XORed with the current plaintext block to produce the current ciphertext block. The main advantage over CBC is that padding is never required.

# Chapter 3

# Redirecting Filesystem Framework

The Redirecting Filesystem Framework (RedirFS for short) is a new layer between file system drivers and VFS implemented as a Linux kernel module. It enables other modules to add new useful functionality to existing file systems such as transparent compression, writing to read-only media, merging contents of several directories into one etc. Modules using RedirFS are called filters.

Filters are like plug-ins in the RedirFS world. They can be registered and unregistered on-the-fly. RedirFS allows them to register callbacks for supported VFS operations and use these callbacks to alter the inner workings of the underlying file system driver.

The first version RedirFS was written by František Hrbata in 2005 as part of his master's thesis at FIT BUT. Its development still continues with the support of AVG Technologies (renamed from Grisoft s.r.o. in 2009).[16]

This chapter is conceived as a guide on how to use and extend the current version of RedirFS. It doesn't explain RedirFS internals – how it works or why. For this kind of information, consult F. Hrbata's Master thesis[6]. Although much has changed since it was written, the described concepts are still valid today.

## 3.1 Filter developement

Before we can jump into filter development, a few prerequisites have to be met. As mentioned at the beginning of this chapter, RedirFS and its filters are kernel modules. They are written in C and need to be compiled against the target kernel. RedirFS needs to be build prior to filters and we can't do this without its source code, which is freely available from `http://www.redirfs.org` along with detailed instructions.[18] After we've successfully compiled RedirFS, we can start writing a filter by creating a new kernel module for it. For more information regarding modules, refer to The Linux Kernel Module Programming Guide[12].

The following subsections go through the basics of filter development by exploring the most important features provided by RedirFS. All described data types and functions are defined in `redirfs.h` unless stated otherwise.

### 3.1.1 Registration

The life cycle of filters start by their registration with RedirFS. The word registration is usually associated with filling out annoying forms revealing personal information about

ourselves. Filters don't have it any easier, but instead of colorful forms, they need to fill in a `redirfs_filter_info` structure.

```
struct redirfs_filter_info {
    struct module *owner;
    const char *name;
    int priority;
    int active;
    struct redirfs_filter_operations *ops;
};
```

`owner`
: A pointer to the module where the filter is implemented. In most cases, `THIS_MODULE` macro is used to fill this member.

`name`
: Name of the filter.

`priority`
: Determines the priority of the filter, lower value means higher priority.

`active`
: If non-zero, the filter will be activated upon registration.

`ops`
: Pointer to a structure containing filter operation callbacks described in section 3.1.3.

Registration is then completed by calling `redirfs_register_filter`, which returns a handle to a RedirFS filter object. Unregistering filters is matter of passing the returned handle to `redirfs_unregister_filter`.

```
redirfs_filter redirfs_register_filter(struct redirfs_filter_info *info);
int redirfs_register_filter(redirfs_filter filter);
```

### 3.1.2 Path lists

RedirFS wasn't designed for filters to be active on the whole file system hierarchy. Although it isn't impossible, filters are usually supposed to be active for selected portions of the directory tree only. This is achieved by the introduction of a concept named path lists.

Each filter has a list of included and excluded paths, where a path is nothing more than a path name such as `/home/user`. Upon registration, the list is empty – there are no files under the filter's influence. When a new path is added, RedirFS attaches a RedirFS root object to the corresponding dentry. If the path name is to be included and it is a directory, the filter becomes active for the whole subtree. On the contrary, if the path name is to be excluded and it is a directory, the filter becomes inactive for the whole subtree. When a file is accessed, the closest (going from child to parent) RedirFS root object is evaluated to determine what filters are active for it.

Before adding a new path to the filter's path list, a `redirfs_path_info` structure has to be filled in.

```
struct redirfs_path_info {
    struct dentry *dentry;
    struct vfsmount *mnt;
```

```
    int flags;
};
```

`dentry`
Dentry object, that represents the path name we want to add.

`mnt`
File system mount point.

`flags`
Either `REDIRFS_PATH_INCLUDE` or `REDIRFS_PATH_EXCLUDE`.

In most cases, we want to add a specific path name such as `/home/user`, but RedirFS provides no way of doing so. Fortunately, there is a handy function named `path_lookup` defined in `<linux/namei.h>`, that retrieves the dentry and mountpoint from path name for us.

After filling in `redirfs_path_info`, a call to `redirfs_add_path` closes the deal and if successful, it returns a handle to a RedirFS path object. Added paths can be removed by calling `redirf_rem_path`.

```
redirfs_path redirfs_add_path(redirfs_filter filter,
        struct redirfs_path_info *info);
int redirfs_rem_path(redirfs_filter filter, redirfs_path path);
```

### 3.1.3 Filter operations

RedirFS provides a way for filters to react to external events triggering operations on them. A typical example of such an event is the inclusion of a new path to the filter's path list. Each filter can define a set of callbacks upon registration by filling in the `ops` member of `redirfs_filter_info` described in the previous section. These callbacks are invoked by RedirFS before triggering the corresponding operation.

The `redirfs_filter_operation` structure is defined as:

```
struct redirfs_filter_operations {
    int (*activate)(void);
    int (*deactivate)(void);
    int (*add_path)(struct redirfs_path_info *);
    int (*rem_path)(redirfs_path);
    int (*unregister)(void);
    int (*rem_paths)(void);
    void (*move_begin)(void);
    void (*move_end)(void);
    int (*dentry_moved)(redirfs_root, redirfs_root, struct dentry *);
    int (*inode_moved)(redirfs_root, redirfs_root, struct inode *);
};
```

`activate`
Called when the filter is about to be activated.

`deactivate`
Called when the filter is about to be deactivated.

`add_path`
: Called when a path is about to be added in the filter's path list.
  The `redirfs_path_info` structure is described in section 3.1.2.

`rem_path`
: Called when a path is about to be removed from the filter's path list.

`unregister`
: Called when the filter is about to be unregistered.

`rem_paths`
: Called when all paths are removed from the filter's path list.

Filters don't have to define all of these callbacks as long as they set unused ones to `NULL`. A callback returning non-zero aborts the corresponding operation.

### 3.1.4 VFS operations

The core functionality of RedirFS revolves around replacing VFS operations with it's own functions and notifying filters when these functions are called. Before invoking the original operation, RedirFS calls pre-callbacks of all concerned filters in order of their priority. When the original operation terminates, post-callbacks are called in reverse order. Filters need to tell RedirFS they want to be notified about VFS operation calls by passing an array of `redirfs_op_info` structures to `redirfs_set_operations`.

```
struct redirfs_op_info {
    enum redirfs_op_id op_id;
    enum redirfs_rv (*pre_cb)(redirfs_context, struct redirfs_args *);
    enum redirfs_rv (*post_cb)(redirfs_context, struct redirfs_args *);
};
```

`op_id`
: Determines the VFS operation and type of file we're registering the callbacks for.
  Accepted values are defined in `enum redirfs_op_id` and are named according to the following pattern:
  `REDIRFS_<file_type>_<object_type>_<operation>`, where:
  `<file_type>` is one of `REG`, `DIR`, `CHR`, `BLK`, `FIFO`, `LNK`, `SOCK` for regular files, directories, character devices, block devices, named pipes, symbolic links, sockets respectively.
  `<object_type>` is one of `FOP`, `DOP`, `IOP`, `AOP` for `file`, `dentry`, `inode`, `address_space` respectively.
  `<operation>` is the operation name in uppercase.
  The last structure in the array passed to `redirfs_set_operations` is required to be a sentinel with this field set to `REDIRFS_OP_END`.

`pre_cb`
: Pointer to the pre-callback.

`post_cb`
: Pointer to the post-callback.

```
int redirfs_set_operations(redirfs_filter filter,
        struct redirfs_op_info ops[]);
```

The first arguments of callbacks being registered is the VFS operation call context used by RedirFS data discussed in section 3.1.5. The second argument is a pointer to `redirfs_args` – a structure containing information about the original operation call.

```
struct redirfs_args {
    union redirfs_op_args args;
    union redirfs_op_rv rv;
    struct redirfs_op_type type;
};
```

`args`
A union of structures storing arguments the original operation was called with.
Pre-callbacks can modify the values stored to alter the original operation's behavior.

`rv`
A union of all types VFS operations can possibly return.
It's value is insignificant in pre-callbacks.

`type`
Type of callback filled automatically by RedirFS.
This allows a single function to act as both pre and post callback.
Valid values are `REDIRFS_PRECALL` and `REDIRFS_POSTCALL`.

### 3.1.5 RedirFS data

Filters often need to store run-time critical data related to specific VFS objects they come in contact with. RedirFS spares them the hardships involved with keeping track of those objects and facilitates object to data lookup by providing a feature called RedirFS data. It allows filters to attach any structure to file, dentry or inode VFS object as well as long as it contains a member of type `struct redirfs_data`. Structures with this property can be also attached to `redirfs_root` (RedirFS root) and `redirfs_context` (VFS operation call context; used to pass data from pre to post callbacks) objects.

RedirFS data are managed using the following set of functions:

```
int redirfs_init_data(struct redirfs_data *data, redirfs_filter filter,
        void (*free)(struct redirfs_data *),
        void (*detach)(struct redirfs_data *));
struct redirfs_data *redirfs_get_data(struct redirfs_data *data);
void redirfs_put_data(struct redirfs_data *data);
struct redirfs_data *redirfs_attach_data_<object_name>(redirfs_filter filter,
        <object_type> <object_name>);
struct redirfs_data *redirfs_get_data_<object_name>)(redirfs_filter filter,
        <object_type> <object_name);
struct redirfs_data *redirfs_detach_data_<object_name>(redirfs_filter filter,
        <object_type> <object_name>);
```

Where `<object_name>` can be either `file`, `dentry`, `inode`, `context`, `root` and `<object_type>` `struct file *`, `struct dentry *`, `struct inode *`, `redirfs_context`, `redirfs_data` respectively. The `redirfs_init_data` function takes two callbacks (or `NULL`) as its last arguments, the first of which is called when the the RedirFS data instance is about to be freed and the second is called when data is being detached from an object. A pointer to

the structure containing a member of type `struct redirfs_data` is ment to be retrieved using the `container_of` macro.

### 3.1.6 sysfs interface

Sysfs is a virtual in-memory file system provided by Linux 2.6. It was designed for kernel drivers to export information about devices into user space, but it also works the other way around and allows device configuration. In sysfs, files represent device attributes and directories represent attribute groups. As a rule attributes are in ASCII format and contain a single value. While this isn't strictly enforced, the size of files in sysfs is usually limited to one page of data. On most Linux distributions, sysfs is mounted at `/sys`.[10]

RedirFS makes use of sysfs for managing registered filters by exporting a basic set of attributes used to perform operations described in section 3.1.3. Filters can also export their own attributes using the following structure and set of functions:

```
struct redirfs_filter_attribute {
    struct attribute attr;
    ssize_t (*show)(redirfs_filter filter,
    struct redirfs_filter_attribute *attr, char *buf);
    ssize_t (*store)(redirfs_filter filter,
    struct redirfs_filter_attribute *attr, const char *buf);
};
```

`attr`
  Kernel structure containing the name of the attribute, its module and access mode.

`show`
  Called when the attribute is read.

`store`
  Called when the attribute is written.

```
int redirfs_create_attribute(redirfs_filter filter,
        struct redirfs_filter_attribute *attr);
int redirfs_remove_attribute(redirfs_filter filter,
        struct redirfs_filter_attribute *attr);
```

## 3.2 Extending RedirFS

At the time of writing this thesis, RedirFS didn't support all VFS operations required to implement a cryptographic filter. This section reflects the author's own experience from extending RedirFS to encompass these operations and ends with some ideas for improving RedirFS in other areas.

### 3.2.1 New supported operations

Adding a new supported operation requires modification of at least two files. In `redirfs.h`, we need to define its unique ID in `enum redirfs_op_id` and a structure to store its arguments in `redirfs_op_args`. Next, depending on what VFS object the implemented operation belongs to, we have to create a new function to override the original operation in `rfs_file.c`, `rfs_dentry.c` or `rfs_inode.c`. This function is required to adhere to a few rules:

- argument list and return value must be the same as the original operation's
- name has to start with "rfs_" prefix followed by the name of the original operation

To be usable by filters, it should invoke the concerned filter's pre-callbacks using the internally defined `rfs_precall_flts`. If it returns non-zero, the original operation should be called, but only if it is set (isn't `NULL`). In some cases, the underlying file system expects VFS to call a generic handler when it isn't set. The only way to know is to investigate VFS source code and see what it does. Fortunately, the checks can usually be found in `do_<operation_name>` routines, where `<operation_name>` is the name of the replaced operation. The next step is to invoke the concerned filter's post-callbacks using `rfs_postcall_flts`.

```
int rfs_precall_flts(struct rfs_chain *rchain, struct rfs_context *rcont,
        struct redirfs_args *rargs);
void rfs_postcall_flts(struct rfs_chain *rchain, struct rfs_context *rcont,
        struct redirfs_args *rargs);
```

The last step is to replace the original operation. This is done in `rfs_<object_name>_set_ops`, where `<object_name>` is either `file`, `dentry` or `inode` depending on the type of object the operation belongs to. A set of macros is defined for this purpose:

```
RFS_SET_<obect_type>(object, operation_id, original_operation_name);
```

Where `<object_type>` is one of `FOP`, `DOP`, `IOP`, `AOP` for `file`, `dentry`, `inode`, `address_space` respectively.

### 3.2.2 Future extension ideas

When implementing the proof of concept cryptographic filter cipherflt discussed in chapter 5, this short list has been compiled with ideas for features that could find their way into future versions of RedirFS.

**Access to original VFS operations**
Filters don't have access to original VFS operations as RedirFS uses opaque data types to hide such information from them. Sometimes a filter needs to invoke a VFS operation it overrides, but it can't do so without serious overhead. Currently the only option is to temporarily disable its own functionality, but it gets rather tricky in a preemptive kernel, because other processes might invoke the operation at anytime.

**Superblock operations**
RedirFS and its filters could benefit from the support of other VFS objects aside from file, dentry, inode and address_space. The first candidate would be the superblock object. Among others, its operations include inode allocation and file system unmounting. Having access to them would give filters more control over the underlying file system.

# Chapter 4

# Cryptographic filter design

At this point, readers are expected to be familiar with all important concepts of encryption and have a general overview of possibilities provided by RedirFS for developing file system extensions on Linux.

This chapter focuses on finding solutions to general challenges related to disk based file encryption from RedirFS filter point of view without going into any implementation details. It tries to discuss different approaches and the motivation behind every decision made while considering both its advantages and disadvantages.

## 4.1 Goals

The first step to any successful design is to make a list of goals we want to achieve with the final product. Some of the proposed goals might prove to be unrealistic, but they should serve as a basis for making design decisions where applicable. Goals set for this project are as follows:

- it should be possible to activate and deactivate the filter at will
- it should be possible to use different encryption for different paths/files
- only files created under the filter's influence should be encrypted
- affected files should remain encrypted on disk at all times
- encrypting/decrypting shouldn't occur on every read/write operation respectively
- security is top priority, but keep it fast and simple; performance and memory footprint is an issue

## 4.2 Encryption mechanism

The cryptographic filter design revolves around choosing the right encryption mechanism. Everything else is of secondary concern.

### 4.2.1 Keys and re-keying

Managing keys is one of the most important aspects of encryption. Decisions need to be made on how users are going to enter their keys and where to store them. It is desirable for keys to be known before any path is included in the filter's path list. The first solution that comes to mind is using module parameters, but it has some serious drawbacks. Keys would have to be entered when loading the module, making automatic loading somewhat problematic without having the key physically stored on disk. We would also loose the possibility to have different keys for different paths, but the real show-stopper is that module parameters are visible to anyone with access to the system. A better approach is to require the user to enter a key when including a new path to the filter's path list. We can attach this key to the corresponding RedirFS root object and retrieve it from there when a file on the corresponding path list is accessed.

Another issue with keys is the process of changing them called re-keying. Keys are much like passwords - they need to be changed on a regular basis in case they got compromised. In contrast to changing passwords, re-keying might be a long and painful task to accomplish, because all data encrypted with the old key needs to be re-encrypted using the new key. With disk based encryption, it could mean re-encrypting anywhere from a few kilobytes to hundreds of gigabytes.

To reduce this amount of data we're going to introduce the concept of master keys. For each new file a master key will be automatically generated and the file will be encrypted using this key. Then we take the user's key, encrypt the master key with it and store it at the end of the file just after the encrypted content. When re-keying takes place we only have to re-encrypt the master password instead of the whole file reducing the amount of processed data considerably.

### 4.2.2 Encrypting file content

The most important question that needs to be answered is: When is the encryption/decryption of file content going to take place? We don't want it to happen on every read/write operation, but we also want data to stay encrypted in the backing store at all times. To achieve both goals, we need to encrypt data just before it is going to be written to disk and decrypt it just after it was read from disk.

We're going to use block ciphers, because they are best suited for the purpose of disk based file encryption as concluded in chapter 2. To provide an acceptable level of confidentiality, a mode of operation needs to be applied, but having every encrypted block dependent on all previously encrypted blocks comes with a few disadvantages. When accessing an arbitrary position, all prior blocks need to be decrypted as well – a serious performance issue. Another drawback is data integrity: A single byte corruption invalidates all following data. It is therefore reasonable to split files into parts and applying the mode of operation only within them. Since files are read/written from/to disk by the means of the readpage/writepage VFS operation, that always handles whole pages of data at the time, we're going to use pages as those parts.

### 4.2.3 Initialization vectors

There is no need for an initialization vector to be kept secret, but it should never be used repeatedly with the same key. Doing so would reveal information about common prefixes shared by two plaintexts or, in the worst case, completely ruin security. Since we're going

to apply a mode of operation to every page, we need as many initialization vectors as there are encrypted pages.

At first glance, the simplest solution is to randomly generate one for every page and storing it somewhere in the file possibly before the encrypted page, but after closer examination, it causes more problems than it solves. It makes the file grow in size. For blocks of size 128 bits and page size 4096 bytes, the ratio is 1:256. Is that a big deal? Readers will surely make their own opinion. More importantly, it breaks the file structure - shifting byte indexes between plaintext and ciphertext. Computing this difference might not be very demanding, but crossing page boundaries is something to be avoided if possible.

After some research, it has been decided to use another approach called ESSIV, which stands for Encrypted Salt-Sector IV. It derives the initialization vector from the sector number (inode number and page index in our case) with a key generated by hashing the key used for page encryption. ESSIV does not specify any particular hash algorithm, but its digest size must be a valid key size for the cipher used.[5] Not only does it save us the hassle of storing initializations vectors, but it also protects the encrypted files from watermarking attacks.

### 4.2.4 Encrypting file names

Encrypting file names presents us with a specific problem, because the resulting ciphertext is very likely to be composed of illegal or unprintable characters. Re-encoding file names after encrypting is therefore necessary to keep them sane. First we need to create a set of allowed characters. The set proposed by D. Lewine's POSIX Programmer's Guide[9] seems to be a reasonable start. It permits the use of digits, lowercase and uppercase letters from the modern Latin alphabet, period, underscore and hyphen giving us a total of 65 characters. It also states that hyphen must not be at the beginning of file names. Removing it from the set seems to be the most convenient option making the number of characters a power of two requiring only 6 bits. Thanks to this property, we can use a re-encoding mechanism similar to uuencode[7] in principle. The least common multiple of 6 and 8 being 24, we can translate 3 characters from the encrypted filename into 4 characters from the allowed set. File name whose length is not divisible by 3 will have to be padded by zeros before being encrypted. The disadvantage of this approach is that file names will become at least one third longer.

Because our cryptographic filter isn't supposed to be active for the whole file system tree, some users might feel uncomfortable with the mix of plaintext and ciphertext file names. Therefore it has been decided to make file names encryption an optional low priority feature.

### 4.2.5 Tagging encrypted files

Every file affected by the filter needs to be tagged. In other words, we need to be able to recognize encrypted and unencrypted files. Encrypted files also need to have additional information attached to them, such as: filter version, algorithm used, key size, initialization vector size, padding, its master key etc.

Traditionally file headers are best suited for this purpose, but using them would shift byte indexes between plaintext and ciphertext. To overcome this disadvantage we're going to use trailers instead. Trailers are the equivalent of headers placed at the end of files after the last data block. Headers are more common, because placing this kind of information at the end of file might negatively affect performance on devices with sequential access. In our case however, it doesn't make any difference. To eliminate the need to read the

trailer from disk on every operation performed on a file, we're only going to read the trailer once when first accessed and attach its data to the file's inode, thus significantly increasing performance.

# Chapter 5

# cipherflt

This chapter is completely devoted to exploring the implementation specifics of the proof of concept cryptographic filter: cipherflt. It doesn't present itself as a fully stable application ready for production usage, but rather as a basis for more robust implementations. Its development has been closely following the design proposed in chapter 4, trying to prove the outlined procedures to be valid and feasible.

## 5.1 CryptoAPI

The Linux kernel provides its own cryptographic framework for performing encryption inside kernel space called CryptoAPI. It includes essentially all of the most popular block ciphers and is designed to operate on whole pages of data through a facility known as scatterlists. These properties make it the ideal candidate for our cause.

CryptoAPI was first introduced in kernel version 2.4.12. At the time, it wasn't part of the standard distribution, because of U.S. export laws concerning cryptographic software and was only available through the international patch patch-int. Since then, the aforementioned restriction have been lifted.[11]

CryptoAPI by itself doesn't perform any encryption, but rather serves as common interface for different algorithms (referred to as transforms) implemented as kernel modules. To make use of it, `<linux/crypto.h>` and `<linux/scatterlist.h>` need to be included in source files.

Probably the biggest disadvantage of CryptoAPI is the lack of documentation. The closest thing to it is the example testing module located in the Linux kernel source files at `crypto/tcrypt.c`. This section tries to make up for it by editing the information extracted from there into human-readable format.

### 5.1.1 Using block ciphers

Block ciphers in CryptoAPI are represented by the structure `crypto_blkcipher`, which is nothing more than a wrapper for the more generic structure `crypto_tfm`. For manipulating them, the following functions are exported:

```
struct crypto_blkcipher *crypto_alloc_blkcipher(const char *alg_name,
        u32 type, u32 mask);
int crypto_blkcipher_setkey(struct crypto_blkcipher *tfm, const u8 *key,
        unsigned int keylen);
```

```
void crypto_blkcipher_set_iv(struct crypto_blkcipher *tfm, const u8 *src,
        unsigned int len);
void crypto_free_blkcipher(struct crypto_blkcipher *tfm);
```

`crypto_alloc_blkcipher` takes the name of the requested algorithm including the mode of operation if any as its first argument. For example, to request the AES cipher in CBC, the string `"cbc(aes)"` is used. The meaning of the remaining arguments is unclear, but their values can be safely set to zero.

### 5.1.2 Encrypting pages

All data to be encrypted by CryptoAPI needs to be referenced through scatterlists. Scatterlists are a special interface in the Linux kernel used to handle physically non-contiguous parts of memory as if they were a single contiguous block. It was primarily designed with direct memory access I/O in mind, as it often needs to perform operations on buffers fragmented all around physical memory. The term used for such operations is scatter/gather I/O and that's where the name scatterlist comes from.[17] These are the basic functions used to setup scatterlists:

```
void sg_set_page(struct scatterlist *sg, struct page *page,
        unsigned int len, unsigned int offset);
void sg_set_buf(struct scaterlist *sg, const void *buf,
        unsigned int buflen);
```

After getting a page or buffer into the scatterlist, one more step is required before we can start encrypting. A valid pointer to `crypto_blkcipher` needs to be wrapped into the following structure:

```
struct blkcipher_desc {
    struct crypto_blkcipher *tfm;
    void *info;
    u32 flags;
};
```

`tfm`
A pointer returned by `crypto_alloc_blkcipher`.

`info`
Used by CryptoAPI internally.

`flags`
This member is set by the encryption functions described further in this chapter.
If an error occurs, it is filled with one or more of the following bit flags:
CRYPTO_TFM_RES_BAD_KEY_LEN
CRYPTO_TFM_RES_BAD_KEY_SCHED
CRYPTO_TFM_RES_BAD_BLOCK_LEN
CRYPTO_TFM_RES_BAD_FLAGS

The encryption/decryption is then performed using these functions:

```
int crypto_blkcipher_encrypt(struct blkcipher_desc *desc,
        struct scatterlist *dst, struct scatterlist *src,
        unsigned int nbytes);
int crypto_blkcipher_decrypt(struct blkcipher_desc *desc,
        struct scatterlist *dst, struct scatterlist *src,
        unsigned int nbytes);
```

Both of them can work in-place by taking a pointer to the same scatterlist in both of their `dst` and `src` arguments.

## 5.2 Data structures

This section describes data structures specific to cipherflt. All definitions can be found in `cipherflt.h` unless stated otherwise.

### 5.2.1 Trailer

The structure `cipherflt_trailer` is used to store data read from the trailer placed at the end of encrypted files as proposed in chapter 4.

```
struct cipherflt_trailer {
    u8 version;
    u8 algorithm;
    u8 key_size;
    u8 iv_size;
    u8 padding;
    u16 block_size;
    unsigned char *key;
};
```

`version`
cipherflt version.

`algorithm`
Encryption algorithm used; 0 for AES, 1 for Triple DES, 2 for Blowfish, 3 for Twofish.

`key_size`
Size of the encryption key in bytes.

`iv_size`
Size of the initialization vector in bytes.

`padding`
Size of padding (required by some modes of operation) in bytes.

`block_size`
Size of blocks encrypted at once using the mode of operation in bytes.
The only supported value is currently 4096.

`key`
The master key's material.

If an unsupported value is read from the file into this structure, the trailer is considered invalid and the file is left unencrypted. The master key's material after every other value has been read sucessfully.

### 5.2.2 Inode data

The structures `cipherflt_inode_data` are attached to encrypted files inodes after first access to them.

```
struct cipherflt_inode_data {
    struct redirfs_data rfs_data;
    struct inode *host;
    struct list_head inodes;
    atomic_t trailer_written;
    struct cipherflt_trailer trailer;
};
```

`rfs_data`
  RedirFS data anchor.

`host`
  Pointer to the inode this structure is attached to.

`inodes`
  List of all `struct inode_data` used to keep track of files under the filters influence.

`trailer_written`
  Non-zero if the trailer has already been written to the file.
  Used to prevent race conditions and writting the trailer more than once.

`trailer`
  The trailer data read from the file.

### 5.2.3 Block

The filter was designed to encrypt blocks of page size at once. In practice however, this isn't always the case as page sizes vary on different architectures. On most (if not all) of them, it is a multiple of 4096 bytes. When larger than this, pages are split into parts. Block structures are used to build a list of these parts to encrypt/decrypt coupled with generated initialization vectors.

```
struct cipherflt_block {
    struct list_head blocks;
    struct page *page;
    unsigned char *iv;
    unsigned int len;
    unsigned int offset;
};
```

`blocks`
  The list of blocks about to be encrypted/decrypted.

`page`
  The page this block is part of.

`iv`
  The generated initialization vector.

`len`
  Size of the block. This is currently always 4096, except for the last block.

`offset`
Offset in the page.

The initialization vector is generated from the page index and it's inode number when the block is created.

### 5.2.4 Context data

`struct cipherflt_context_data` is used for VFS operations to carry over the list of blocks and information about the encryption about the be performed between pre and post callbacks.

```
struct cipherflt_context_data {
    struct redirfs_data rfs_data;
    struct list_head blocks;
    struct blkcipher_desc desc;
    struct trailer *trailer;
};
```

`rfs_data`
RedirFS data anchor.
`blocks`
List of blocks about to be encrypted/decrypted.
`desc`
The block cipher used.
`trailer`
Pointer to trailer data read from the file. Used for encryption parameters.

This structure is supposed to be created in pre-callbacks as lists of blocks to be encrypted/decrypted. They usually need to be constructed there, but the actual process is triggered in the post-callback.

## 5.3 VFS operations

In this section, we're going to look into the VFS operation callbacks cipherflt registers with RedirFS. The goal is to explain in detail its procedures, effectively describing the filters functionality. All of the mentioned functions are defined in `rfs.c` unless stated otherwise.

### 5.3.1 open

When a file is opened, ciperflt has to determine if it is encrypted or not. If the file has been open before and it is encrypted, its inode is going to have the filter's data attached to it. Otherwise we're going to make an attempt at reading the trailer. If successful, we assume the file is encrypted. By reading the trailer, at least one page must have been read from disk. The utility function `decrypt_already_read_pages` builds blocks from all the file's pages in the page cache and decrypts them.

#### Pre-callback `cipherflt_pre_open`

1. Check if the file's inode has the filter's data attached to it. If it doesn't, go to step 3.
2. If the trailer has been written to the file, delete it by altering the inode size as if the trailer wasn't there. Return.
3. Allocate new inode data. If the inode size is zero, assume this is a new file. If it isn't, go to step 5.
4. Generate a master key for the file. Go to step 7.
5. Check if the file is encrypted by trying to read the trailer. If it isnt' goto step 8.
6. Delete the trailer by altering the inode size as if it wasn't there and call `decrypt_already_read_pages`.
7. Attach inode data and return.
8. Release the inode data allocated in step 3 and return.

### 5.3.2 release

The original intent was to write the trailer when the last process using the file closes it. Unfortunatly, there is no way to tell the number of processes, that have the file open. Because of this fact, we need to write the trailer everytime the file is released by a processes in the case it wasn't written already. To prevent race conditions an atomic check is made.

#### Post-callback `cipherflt_post_release`

1. Check if the file's inode has the filter's data attached to it. If it does and the trailer hasn't been written to the file yet, do it now.

### 5.3.3 write

To save space on disk, some operating systems including Linux use a concept called spare files. They are like any other file, except some of their blocks are empty. Spare files are created by writing past the file size leaving empty blocks behind. These blocks are refered to as holes. When a hole is read into a buffer, its corresponding positions are filled with zeros. While holes might be useful in some cases, they mess up our encryption mechanism. As a solution, we're going to fill all potential holes with zeros before they are created.

#### Pre-callback `cipherflt_pre_write`

1. Check if the file's inode has the filter's data attached to it. If it doesn't, return.
2. Compare write offset to the file's size to check if this write operation would create a hole in the file. If it would, fill the gap between file's size and offset with zeros first.

### 5.3.4 readpages

When pages of an encrypted file are read from disk into memory, we need to decrypt them. We cannot do this on the basis of a single page, because some file systems do not use readpage at all. Due to the way readpages works, we need to make a list of pages that are going to be read before invoking the original operation.

**Pre-callback** `cipherflt_pre_readpages`

1. Check if the file's inode has the filter's data attached to it. If it doesn't, return.
2. Allocate new context data.
3. Go through the LRU list pointed to by the pages argument and build blocks out of it. We need to do this now, because file system implementations of readpages remove entries from the list while processing it. Note that at this point, the page structures in the list are newly allocated and only their index member is valid.
4. Attach context data and return.

**Post-callback** `cipherflt_post_readpages`

1. Retrieve the context data.
2. Decrypt all blocks.
3. Detach context data and return.

### 5.3.5 writepages

When pages of an encrypted file are written to disk, we need to encrypt them. After they have been encrypted and written to disk, we need to decrypt them back, so they remain intelligible in memory. This might seem like waste of CPU cycles, but the only way around would be to create new pages for encrypted data. Since we can't use writepage, because some file systems don't use it, we would have to duplicate all the pages being written at once, consuming a considerable amount of memory.

**Pre-callback** `cipherflt_pre_writepages`

1. Check if the file's inode has the filter's data attached to it. If it doesn't, return.
2. Allocate new context data.
3. Retrieve the range of pages designated for writeback from the wbc argument.
4. If trailer has been written to the file, alter its inode size as if the trailer wasn't there.
5. Lookup all dirty pages of the file in the range and build blocks out of them.
6. If trailer has been written to the file, restore its inode size
7. Encrypt all blocks.
8. Mark all `buffer_heads` of affected pages as dirty, because we want to write them whole. This is especially necessary on ext3 file system as discussed in section 5.4.

9. Set `sync_mode` of the `wbc` argument to `WB_SYNC_ALL` to make sure the encrypted pages will be really written to disk.
10. Attach context data and return.

**Post-callback** `cipherflt_post_writepages`

1. Retrieve the context data.
2. Decrypt all blocks. Before decrypting each of them, check if their page has the writeback flag set. If it does, wait for it to clear. This prevents overwrites before the pages are flushed to disk.
3. Detach context data and return.

### 5.3.6 d_iput

When an inode is released, we need to release the data attached to it, if there is any.

**Pre-callback** `cipherflt_pre_d_iput`

1. Check if the file's inode has the filter's data attached to it. If it doesn't, return.
2. Detach inode data and return.

## 5.4 Unsolved issues

This section outlines the most important issues of cipherflt, some of which had no solution at the time of writing this thesis or an intermediate solution unfit for real life situations.

**Synchronization**
What is meant by synchronization in this context, is making sure that nobody touches the pages we are currently processing in readpages/writepages callbacks. This applies to both read and write operations. Unfortunately, it is very hard to tell who has access to a specific page at any given moment and the problem can't be solved by using conventional synchronization methods such as spinlocks, semaphores or completions. The only reliable way is to lock the page, but it doesn't protect it from being read. Even if it did, writepages always unlocks pages, making space for race conditions before we lock them again.

**Journaling file systems**
Journaling file systems such as ext3 are problematic when it comes to writing data to disk. Their writepages operations are called as with any normal file system, but they rarely use it for data integrity. Instead, they log all changes made to the journal and write them later using `submit_bh` directly. As a result, we can't capture and encrypt data before it is written to disk by overriding VFS operations. A way around this is to force data synchronization in writepages by dirtying all `buffer_head`s manually, but this results in duplicate writes and performance loss.

**Memory-mapped files**
Memory-mapped files are a feature of modern operating systems allowing on disk files to be mapped to primary memory. The benefits are increased I/O performance as memory

operations are generally much faster than making a system call. Memory-mapped files require special treatment beyond readpages/writepages replacement. On Linux, they are used by the process loader and thus cipherflt isn't suitable for encrypting executable files.

**Direct I/O**

Direct I/O mode is activated by using the `O_DIRECT` flag when opening a file with the open system call. In this mode, data is read/written directly from/to disk on every I/O operation making our encryption mechanism obsolete. It is currently ignored by cipherflt and considered as a way to access the raw encrypted data without deactivating the filter.

# Chapter 6

# Conclusion

The main goal of this thesis was achieved to a certain extent. cipherflt - the Linux kernel module providing transparent encryption for the ext2 and ext3 file systems was implemented. While it is far from being a production ready solution, it proves the proposed design to be valid and it is a good start for further development and future projects. A contribution to RedirFS was made by implementing support for new VFS operations and describing some of the undocumented features of the current version.

The author values the knowledge and experience gained.

# Bibliography


[1] Bovet, D. P.; Cesati, M.: *Understanding the Linux Kernel, Third Edition*. O'Reilly, 2006, ISBN 978-0-596-00565-8.

[2] Bruce Schneier: The Blowfish Encryption Algorithm. `http://www.schneier.com/blowfish.html`.

[3] Bruce Schneier: Twofish. `http://www.schneier.com/twofish.html`.

[4] Erez Zadok, I. B.; Shender, A.: *Cryptfs: A Stackable Vnode Level Encryption File System*. Columbia University, 1998, CUCS-021-98.

[5] Fruhwirth, C.: *New Methods in Hard Disk Encryption*. Vienna University of Technology, 2005.

[6] Hrbata, F.: *Callback Framework for VFS layer*. FIT BUT, 2005, Master's Thesis.

[7] IEEE, T.; Group, T. O.: *IEEE Std 1003.1*. The Open Group, 2004.

[8] Laboratories, R.: *RSA Laboratories' Frequently Asked Questions About Today's Cryptography, Version 4.1*. RSA Security, Inc., 2000.

[9] Lewine, D.: *POSIX Programmer's Guide: Writing Portable UNIX Programs*. O'Reilly, 1991, ISBN 0-93717-573-0.

[10] Mochel, P.: *The sysfs Filesystem*. Ottawa Linux Symposium, 2005.

[11] Patůček, D. J.: *CryptoAPI Linux*. 2003, lecture from cryptofest, video record available at `http://www.avc-cvut.cz/`.

[12] Peter Jay Salzman, M. B.; Pomerantz, O.: *The Linux Kernel Module Programming Guide*. Linux Documentation Project, 2007.

[13] of Standards, N. I.; Technology: *Advanced Encryption Standard (AES)*. FIPS PUBS, 2001, FIPS 197.

[14] of Standards, N. I.; Technology: *Advanced Encryption Standard (AES)*. FIPS PUBS, 2001, FIPS 197.

[15] of Standards, N. I.; Technology: *Recommendation for Block Cipher Modes of Operation*. U.S. Government Printing Office, 2001, NIST Special Publication 800-38A.

[16] Web page: AVG Technologies CZ. `http://www.avg.cz`.

[17] Web page: LWN.net: Scatterlist chaining. http://lwn.net/Articles/234617.

[18] Web page: Redirecting Filesystem Framework. http://www.redirfs.org.

[19] Web page: TrueCrypt. http://www.truecrypt.org.

# Appendix A

# Contents of enclosed CD

The enclosed CD contains:

**cipherflt**
  Source files of the proof of concept encryption filter for RedirRFS

**redirfs**
  Modified snapshot of RedirFS with new supported VFS operations required by cipherflt

**tex**
  LaTeX source files used to compile this document

**xzunap00.pdf**
  This document in Portable Document Format.