Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Protected Filesystem for Linux

Bachelor Thesis

2025
Peter Bruha

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Protected Filesystem for Linux

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	RNDr. Jaroslav Janáček, PhD.

Bratislava, 2025
Peter Bruha

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Peter Bruha
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Protected Filesystem for Linux
Chránený súborový systém pre Linux

Anotácia: Ochrana dôvernosti a integrity uložených údajov je v súčasnosti častou požiadavkou. Mnohé bežne používané riešenia na šifrovanie diskov alebo súborových systémov však vykazujú nedostatky najmä v oblasti chýbajúcej kontroly integrity a opakovaného používania rovnakého kľúča a inicializačného vektora. Predmetom práce je implementácia chráneného súborového systému pre Linux, ktorý bude možné použiť ako nadstavbu nad bežný súborový systém, a ktorý splní minimálne nasledovné požiadavky:
- zachová možnosť náhodného prístupu k jednotlivým blokom súboru pre čítanie aj zápis,
- zabezpečí ochranu dôvernosti obsahu šifrovaním spôsobom, ktorý neumožní identifikáciu rovnakých blokov otvoreného textu v priestore ani v čase,
- zabezpečí ochranu integrity spôsobom, ktorý umožní detekciu neautorizovanej zmeny bloku súboru vrátane jeho nahradenia iným blokom.

Cieľ:
- naštudovať problematiku šifrovania diskov a súborov a zhrnúť slabiny bežne používaných riešení,
- navrhnúť vhodné riešenie spĺňajúce definované požiadavky,
- implementovať riešenie pomocou subsystému FUSE,
- porovnať vlastnosti riešenia s vybranými existujúcimi riešeniami.

Vedúci: RNDr. Jaroslav Janáček, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 26.10.2022

Dátum schválenia: 16.10.2023 doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Peter Bruha
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Protected Filesystem for Linux

Annotation: Protection of confidentiality and integrity of stored data is a frequent requirement today. However, many commonly used solutions for disk or filesystem encryption show shortcomings, especially in the area of missing integrity checks and repeated use of the same key and initialization vector. This thesis focuses on an implementation of a protected filesystem for Linux, which can be used as on top of a regular file system, and which fulfils at least the following requirements:
- preserves the possibility of random access to individual blocks of files for both reading and writing,
- ensures protection of confidentiality of the content via encryption in a way that does not allow identification of the same blocks of plaintext in space or time,
- ensures integrity protection in a way that enables detection of unauthorized changes to a block of a file, including its replacement by another block.

Aim:
- study the issue of disk and file encryption and summarize the weaknesses of commonly used solutions,
- propose a suitable solution fulfilling the specified requirements,
- implement the solution using the FUSE subsystem,
- compare the properties of the solution with selected existing solutions.

Supervisor:	RNDr. Jaroslav Janáček, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	26.10.2022
Approved:	16.10.2023

doc. RNDr. Dana Pardubská, CSc.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgements: I would like to thank my supervisor, RNDr. Jaroslav Janáček, PhD. for allowing me to work on this thesis, for his expertise and invaluable feedback during the implementation of the filesystem. Also, I am grateful for his lectures on many interesting topics such as networks, systems programming, and Linux administration.

Abstrakt

Táto práca podrobne popisuje návrh a implementáciu chráneného súborového systému pre Linux. Rieši rôzne bezpečnostné zraniteľnosti v existujúcich riešeniach, ako napríklad absencia kontrol integrity a opätovné použitie kryptografických kľúčov a inicializačných vektorov. Úvodné kapitoly obsahujú predbežnú diskusiu o súborových systémoch a kryptografii, stanovujú potrebné pojmy a koncepty a definujú požiadavky, ktoré usmerňujú neskoršie fázy vývoja. Nasledujúce sekcie opisujú návrh jednotlivých častí, po ktorých nasleduje implementácia. Tieto časti rozoberajú problémy, ktoré sa vyskytli počas vývoja, a ich príslušné riešenia. Práca končí empirickými porovnávacími testami a porovnaniami s existujúcimi riešeniami. Súčasťou práce je funkčný chránený súborový systém, spĺňajúci požiadavky definované predtým.

Kľúčové slová: súborový systém, FUSE, dôvernosť a integrita

Abstract

This thesis details the design and implementation of a protected filesystem for Linux. It addresses various security vulnerabilities in existing solutions, such as the absence of integrity checks and the reuse of cryptographic keys and initialization vectors. The initial chapters contain preliminary discussion about filesystems and cryptography, establish necessary terms and concepts, and define requirements that guide later phases of development. Subsequent sections describe the design of individual parts, followed by the implementation. These sections discuss the challenges encountered during the development and their respective solutions. The thesis concludes with empirical benchmarks and comparisons against existing solutions. The thesis includes a working protected filesystem satisfying the requirements defined previously.

Keywords: filesystem, FUSE, confidentiality and integrity

Contents

List of Figures

List of Code Listings

List of Charts

Introduction

Modern computing, for the most part, consists of large amount of data. Secure storage of data is becoming increasingly important and the general requirement for its protection is confidentiality and integrity. Although, there exist many established solutions for disk or filesystem encryption, many of them show shortcomings such as missing integrity checks and repeated use of keys and initialization vectors.

The primary focus of our thesis is the implementation of protected filesystem for Linux which can be used on top of commonly used filesystems such as ext family of filesystems, btrfs, ZFS, or others. Additionally, the implementation needs to preserve the random access property of files without incurring any significant performance overhead. It also needs to be secure, ensuring confidentiality and integrity of the stored content, not leaking any information such as identification of same plaintext blocks or, in case of integrity, the permutation or replacement of blocks. The implementation is done using the FUSE - Filesystem in Userspace feature of the Linux Kernel.

The thesis consists of 6 chapters. The *first chapter* focuses on basic introduction to filesystems and cryptography. The primary goal is to give some background into how filesystems and cryptography work. It introduces concepts such as Filesystem in Userspace and encryption algorithms that are used throughout the thesis. The *second chapter* defines requirements of our software, which we adhere to during the design phase. The *third chapter* is largely focused on the design and architecutre of individual parts that the filesystem consists of. Each part is comprised of defining the problem needed to be solved. Following with exploration of possible solutions. Finally, ending with design decision and rationale of why the specific solution was chosen. In the *fourth chapter*, we discuss the decisions made during the implementation of the FUSE-based filesystem and the user facing interface. Finally, the *fifth chapter* focuses on the benchmarks and comparisons against existing solutions and the *sixth chapter* focuses on possible further work and improvements to the filesystem.

Chapter 1

Filesystems and Cryptography

The first part of this chapter gives brief explanation of what a filesystem is, how it closely relates to the virtual filesystem and what it means for a filesystem to be based on FUSE. The second part describes cryptography principles, ciphers, authenticated encryption with associated data, hash functions, and key derivation functions. In the third and the last part, we look and compare similar solutions that exist right now.

1.1 Filesystems

Filesystems are an integral part of the day to day computing. They handle the organization and access to data. They provide shared access to storage medium and usually have a tree-like structure of files. Without a filesystem, every software would have to handle the access to a physical storage medium by itself which is both infeasible and could potentially lead to data corruption, deadlocks or in the worst case to a complete data loss. Multiple different filesystems exist with their respective advantages and disadvatages. Some are optimized for speed, others for data integrity and security, some work better with hard disk drives, others with solid state disks. Nowadays, there even exist distributed filesystems that can operate across the network. Some popular filesystems used on Unix are ext4, btrfs, or ZFS, on Windows it is NTFS, or its possible successor ReFS.

Now we take a closer look at how an Unix-style filesystem such as ext4 works (other filesystems usually work analogously). The fundamental data structure used is called an **inode** that stores metadata about a file or directory, such as permissions, size, timestamps and locations of data blocks. Notably, it does not store the name nor actual content. A file is an entry in a directory that points to an inode which contains the file metadata. Multiple file entries can point to the same inode which is called a **hardlink**. The directory inode contains names of directory entries, such as files and directories, and their respective inodes.

1.1.1 Virtual Filesystem - VFS

Virtual filesystem is an abstraction in Unix-like systems which allows multiple different filesystems be accessed from a single hierarchy, usually starting at root, denoted by `/`. It allows client software to access the underlying filesystems in a common way. In order for it to all work seamlessly the individual filesystems are required to implement common functions such as:

- `mount()` and `unmount()` - mounting a filesystem makes the filesystem contents available at specified path.
- `read()` and `write()` - functions for reading from and writing to a specified file.
- `open()`, `close()`, `stat()` - functions for opening, closing and getting metadata of a file, respectively.
- `chmod()` - allows modifying the permissions of a file.

Virtual filesystem facilitates the Unix philosophy of everything being a file. Concrete filesystems can be mounted. A ramdisk, filesystem that uses RAM as the underlying storage medium, can be mounted. The individual hardware in a computer is accessible through a virtual filesystem. Each process has a corresponding file in the VFS. The FUSE filesystem, which we talk about in the next part, also plug into the virtual filesystem.

1.1.2 Filesystem in Userspace - FUSE

Filesystem in Userspace, as the name suggests, allows non-privileged software from userspace to hook into the virtual filesystem. It allows users to write their own filesystems without having to touch any of the kernel code.

Filesystems implemented using FUSE communicate with the Linux kernel through the FUSE protocol. Figure 1.1 describes how the protocol interacts with the individual parts. As an example, when a user wants to list directory contents with `ls`, from userspace `ls` uses the virtual filesystem function call for reading the directory that then gets passed to kernel's virtual filesystem. The VFS then passes the call to the appropriate driver, in this case the FUSE kernel module, that then communicates with the filesystem implementation that is in userspace. The kernel FUSE module and the concrete userspace filesystem communicate through a shared file descriptor. The concrete filesystem implementation has to provide certain functions for it to work seamlessly. The functions required and their documentation are described in FUSE docs [1].

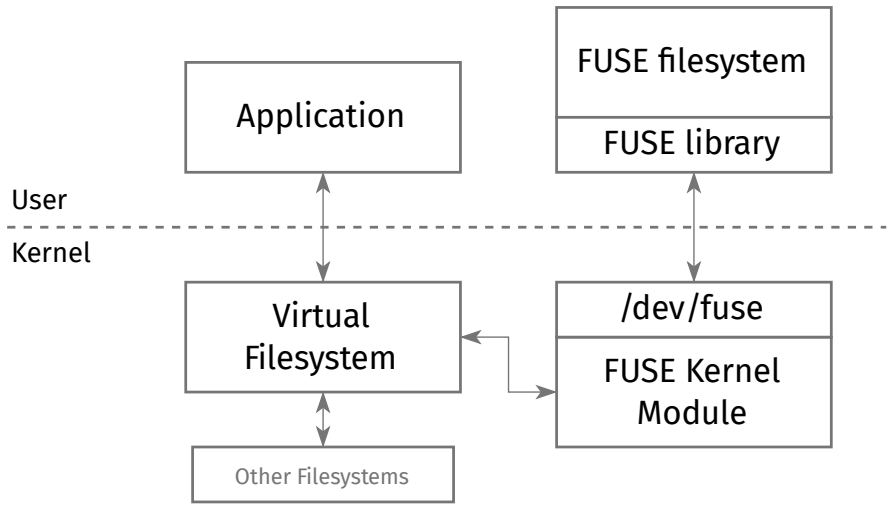

Figure 1.1: Example FUSE communication

1.2 Cryptography

Cryptography, the art of hidden writing, studies various algorithms, schemes and protocols and their security properties such as confidentiality, integrity, authenticity, non-repudiation. The properties we mainly care about are confidentiality and integrity.

- **Confidentiality** ensures that information remains accessible only to authorized parties through the use of encryption.
- **Integrity** ensures that data has not been in any way altered, modified, nor corrupted by unauthorized parties. In our case achieved by using hash functions.

The following text defines some common terminology used in cryptography. The data before encryption is called **plaintext**. The encrypted data is called **ciphertext**. The cryptography algorithms are called **ciphers**. The ciphers usually require a secret **key** that is used when encrypting and decrypting the data. The key, ideally, is a secret between the two or more parties doing communication and should not be known to the outside world.

In the past, cryptography was mostly focused on data confidentiality, that is, the encryption and decryption of data. Some examples of one of the oldest ciphers is the Caesar cipher. It is a simple substitution cipher where each letter in the plaintext is shifted by 3 positions, wrapping around. The name comes from Julius Caesar who, allegedly, used it for communication with his generals. The key in Caesar cipher is the amount the letters are shifted by. Another well-known substitution cipher is the

Vigenère cipher. Compared to the Caesar cipher, the only difference is that each position of a letter in the plaintext is shifted by the same position letter in the key, i.e. plaintext “hello” and key “abcde” would produce ciphertext “igopt”. These ciphers are easily defeated by frequency analysis, a technique that is based on the inherent property of natural languages having certain letter patterns.

Modern cryptography is built upon the rigorous foundation of mathematics. The cipher techniques can be categorized into symmetric cryptography and asymmetric cryptography, the latter is also known as public key cryptography. In the former, both parties use the same key agreed upon beforehand. Whereas in the latter, each party generates their own private and public key pair which they use to compute a shared secret. Asymmetric encryption is based on intractable mathematical problems such as integer factorization. discrete logarithm, etc. The following text primarily focuses on symmetric encryption since it is used in the implementation.

Symmetric encryption ciphers can be categorized into two groups based on the processing of data.

- **block ciphers** process data on fixed-length input called blocks. Example: AES, Blowfish.
- **stream ciphers** process data bit by bit, plaintext is usually combined with a keystream that is produced by the cipher. Example: Salsa20, ChaCha20.

The symmetric ciphers also have different modes of operation. The way of how they combine the plaintext with the cipher. Some block modes require **initialization vector (IV)**, bits of data not requiring secrecy, but are later needed for decryption. Some block modes as an example include:

- Electronic Codebook (ECB) - plaintext block is encrypted into a ciphertext block.
- Cipher Block Chaining (CBC) - plaintext block is XORed with the previous ciphertext block and then is encrypted into a new ciphertext block, uses IV as the initial ciphertext block.
- Counter (CTR) - input into the cipher is an IV and a counter, output from the cipher then is XORed with the plaintext.

Most commonly used block cipher is AES which can use block lengths of 128, 192 or 256 bits. Some stream ciphers worth mentioning are Salsa20 and ChaCha20. Both are add, rotate, XOR (ARX) ciphers. The latter is a modified Salsa20 with more performant round function with increased diffusion. The 20 in their name implies the amount of rounds performed.

None of these modes provide protection against any manipulation of the ciphertext, be it accidental or adversarial. Authenticated encryption is covered in the following section.

1.2.1 Authenticated Encryption with Associated Data

Authenticated encryption provides confidentiality and also authenticity of the encrypted data. These ciphers enable the detection of ciphertext manipulation. They achieve it by additionally computing and saving a hash, or a MAC (message authentication code). Some block modes providing authentications are:

- Counter with CBC-MAC (CCM) - plaintext encrypted using CTR mode and authentication tag computed as CBC-MAC, authenticate-then-encrypt, needs to pass over plaintext twice.
- Galois/Counter mode (GCM) - input into cipher is an IV and a counter, additionally computes a GHASH using associated data and ciphertext.
- Synthetic Initialization Vector (SIV) - generates unique IV from the key, the plaintext, and any associated data, usually using CMAC, encryption using CTR mode.

From block ciphers we have chosen **AES in GCM mode** as it provides both confidentiality and authentication and is best suited for our needs. From stream ciphers the best candidate is **ChaCha20Poly1305**. As the name implies, it is a ChaCha20 cipher that additionally computes a message authentication code using the Poly1305 hash function. It usually has faster performance when compared with AES-GCM without hardware instructions.

1.2.2 Hash Functions

Hash functions turn input of arbitrary length to a fixed size output called **hash**, **digest**, **checksum**. Hash functions are intentionally designed to be irreversible, providing preimage and collision resistance. They also have a property where changing a single bit in input causes each bit in output to have 50% probability of flipping. Merkle-Damgård construction contains a fixed input length compression function with collision resistance. Sponge construction contains a finite state that “absorbs” input of arbitrary length and then “squeezes out” output of fixed length. Some of the common hash functions are:

- MD5 - Merkle-Damgård construction, producing output of 128-bits, it is not secure anymore and should not be used.

- SHA2 - family of Merkle-Damgård hash functions with input limit of 2^{64} bits.
- SHA3 - successor to SHA2 family of hash functions, based on Keccak [2] and the sponge construction, input can be arbitrarily long.
- BLAKE2 - finalist of SHA3 competition, based on ChaCha20 stream cipher, input can be arbitrarily long [3].

In our implementation we are using **BLAKE2** with optionally changing to **SHA3**. We have chosen BLAKE2 primarily due to the speed and strong security guarantees. The hash functions are used for integrity checking of entire files. See Chapter 3.2 for more information.

1.2.3 Key Derivation Functions

Key derivation functions are cryptographic algorithms used for deriving secret keys out of passwords or master keys. The primary use of key derivation functions is to stretch keys into a desired length. The output of KDF is called output key or hash. Common use for key derivation functions is **password hashing**, where they are used for strengthening the input material, usually a user password, which is not as strong as a randomly generated vector of bytes. It is accomplished by combining the password with additional arbitrary input, called **salt**, which together is used to generate cryptographically secure output key. The use of salt makes bruteforce attacks on the input password way harder. It prevents attacker to simply lookup into a precomputed table containing password and its respective hash since the input additionally depends on the randomly generated salt. Modern key derivation functions allow setting the number of iterations and amount of memory to use during the computation. It leads to significantly slower cracking of passwords even when the salt is known since the computation alone is an expensive operation. The commonly used key derivation functions are PBKDF2 [4], bcrypt [5], scrypt [6], argon2 [7]. The first two are easy to parallelise and have small memory footprint, whereas the latter two have controls for managing the amount of memory used during operation.

Our filesystem uses **Argon2id** for the secure storage of master key. Argon2id is a hybrid mode of Argon2d and Argon2i. The first uses data-dependent memory access, the memory accesses later depend on the previous computation. The primary objective of Argon2d is to provide resistance against GPU and ASIC-based cracking. The second uses data-independent memory access, the memory access is predetermined and does not depend on the input data. The goal is to resist side-channel timing attacks. Argon2id is a hybrid between the two, aiming to provide balanced resistance to both GPU cracking and side-channel attacks. It accomplishes this by using data-independent memory access for the first half of the first pass and then

switches to data-dependent memory access for the remainder of the first pass and subsequent passes.

1.3 Comparison with existing solutions

The following part analyses existing solutions. We primarily look at EncFS and gocryptfs. Finally, we briefly mention disk encryption solutions.

EncFS [8] originally released for Linux in 2003. It is written in C++ and runs in userspace using the FUSE library. For cryptography, it is using openssl. It was one of the first solutions that offered encryption at the filesystem level. It uses one key for encryption of all the files, AES in CBC mode, which does not provide any kind of data authenticity and integrity. For ensuring that the data was not tampered with it uses openssl's implementation of HMAC. Although, message authentication codes are a valid solution, in the EncFS case they are using 64 bit MACs, which nowadays can easily be broken by bruteforce. During a security audit of EncFS v1.7.4 in 2014 [9] multiple potential vulnerabilities were found. many of the vulnerabilities exist likely due to its age, since during its inception, the standard security practices were more relaxed compared with today. After the security audit, version 1.8 was released fixing some of the problems found. Unfortunately, most of the security issues remain to this day and the project is no longer maintained.

gocryptfs [10] It is heavily inspired by EncFS. It is written in Go, a memory safe programming language, which is a huge plus from the security point of view. For cryptography, it is using Go's standard library, but optionally can use openssl. Many software vulnerabilities stem from the usage of memory unsafe languages [11]. The project is aiming to fix most of the security shortcomings of EncFS as can be seen in the design [12]. The cryptographic primitives it employs are AES in GCM mode which is known for providing both data confidentiality and authenticity, and scrypt as a key derivation function, which makes a bruteforce attack infeasible due to its high memory requirement. It manages to be way faster than EncFS while using more complicated and stronger ciphers.

Although gocryptfs has many positives, some negatives exist, too. It offers data authenticity and integrity over large amount of blocks, but does not prevent replacing current encrypted blocks with previous versions of them. The other possibly bad design choice is the absence of block alignment. From the design [12] we can see that gocryptfs is using 4KiB blocks where each block gets its own unique initialization vector. The place where it stores these initialization vectors is at the beginning of each 4KiB block. Unfortunately this way of storage may have a negative impact on the amount of disk accesses. The filesystem blocks are

not aligned to the physical medium's blocks, and as such, when we want to access 4KiB filesystem block, in reality, we need to access two 4KiB physical blocks cause the 4KiB filesystem block would actually take 4KiB + additional data for IV and authentication tag. This may have a detrimental effect on disk intensive software such as database management systems. Those systems usually store their data in blocks of certain size. As an example, when looking at PostgreSQL [13], they store their data in 8KiB (or larger) blocks. In a situation where there is a large amount of data accessed the negative impacts may be noticeable.

Disk encryption dm-crypt and LUKS [14] on Linux, and BitLocker [15] on Windows. As the name suggests, these software solutions encrypt entire disks instead of individual files. However, that comes with some drawbacks such as one key used for everything. Additionally, they are constrained by the underlying disk space, usually, ciphertext size has to match with the size of sectors on a disk, which leads to no support for storing extra information for authenticated encryption schemes, such as initialization vectors, hashes, digests or MACs.

Chapter 2

Requirements

This chapter starts with defining the requirements of our software. Afterwards, it discusses prgramming languages and the search for a suitable programming language for our filesystem. We list the specific needs, features, and properites of the mentioned programming languages.

2.1 General Requirements

Requirements are an integral part of every design process. They define what properties and goals our software needs to fulfil and are used as a helpful guide throughout the design phase of the software development. The filesystem must meet these requirements:

1. **Confidentiality** The software shall not reveal any information about the original plaintext to unauthorized parties.
2. **Integrity** The software shall discern if the ciphertext has been tampered with.
3. **Random Access** The software shall allow random access to data without incurring any noticeable performance penalty.
4. **Integrity of entire files** The software shall not allow adversary to permute ciphertext blocks of a file, nor replace with previous versions of the block.
5. **Performance** The software shall be fast enough even for performance critical applications.

When it comes to filenames, we additionally require:

6. **Determinism** The same filename shall encrypt to the same ciphertext given the same encryption key.

7. **Collision Avoidance** Different filenames shall not encrypt to the same ciphertext.

2.2 Programming Language

Choosing the right programming language is essential, as the choice is made only once and changing it later in implementation phase, due to shortcomings, is almost impossible.

There are two groups of languages, compiled and interpreted. As the name suggests, code of compiled languages is directly translated to machine code which later can be executed on a CPU. In interpreted languages, the code is executed with the help of an interpreter.

Since we require the programming language to be suitable for performance critical applications., we are primarily looking at compiled languages. The three candidates we have chosen are C, C++, or Rust. Although Rust is a relatively new language, Rust code is compiled into an LLVM intermediate representation (IR) which is then compiled to machine code. The clang compiler for C++ and C works the same way. It allows Rust to utilize the decades of compiler design and optimizations done to the LLVM compiler infrastructure. All three of the chosen languages are suitable for systems programming.

The C and C++ are languages with decades of history, used in performance critical workloads across many different industries. Most of the software used today is built upon the C family of languages. Probably the most known performance critical software written in C is the Linux Kernel. Although the languages are ubiquitous, they are not perfect. Both are memory unsafe, that is, the compiler does not check for unsafe memory operations such as reading/writing out of bounds, accessing unallocated memory, buffer overflows, dangling pointers, etc. Another drawback is the tooling. Even though the languages have existed for many decades, the tooling is fragmented and archaic. There does not exist any unified way of package management. The compiling of programs is done through build tools such as the simple make, or more complicated CMake which requires to learn an entire new language. The languages do not have any native support for testing. Testing is a great way to catch a lot of bugs, errors, and regressions.

Compared all of the mentioned issues with Rust, one of newer languages, which was designed to prevent, and in some cases, completly eliminate certain types of bugs. It has great tooling with the unified build tool and package manager `cargo`. The language natively supports tests, both unit and integration. It is designed to easily create and run them. Lastly and probably most significantly, it has the concept of ownership and borrowing. The ownership rules are that each value has one owner at a time, and the value will be dropped when the owner goes out of scope [16]. Additionally, the rules for borrowing are that at any given time, you

can have either one mutable reference or any number of immutable references, and references must always be valid [17]. It is similar to the definition of data race, where two or more pointers access the same memory location at the same time, where at least one of them is writing, and no synchronization is done. These simple rules eliminate many of the common bugs that are possible in other languages. Bugs such as accessing and modifying a variable at the same time, use after free errors, dangling pointers, and various data races. Unfortunately Rust also has its negatives, since it is a relatively new language, the ecosystem is not as mature when compared to C/C++. The learning curve is steeper due to differences in syntax and semantic features such as ownership, borrowing, and lifetimes. The memory safety may in certain scenarios introduce noticeable performance overhead. Rust performs bounds checking by default which incurs slight overhead, although most of the time, the bound checking is optimized away by the compiler.

After consideration of the mentioned languages, we have chosen to write the filesystem in the Rust programming language. It provides the needed memory safety when building a cryptosystem. It is a compiled language which is suitable for performance critical software. The drawbacks of slight performance penalty in certain situations and the not as mature ecosystem are an acceptable tradeoffs.

Chapter 3

Design

This chapter focuses on the design of the individual parts that form our filesystem. At the beginning we discuss the data encryption and layout of a file. Then we move onto integrity of files, and how we handle the encryption of directory entries. At the end we take a look at the handling of secrets such as user passwords, key encryption keys and data encryption keys.

3.1 File Layout and Encryption

This section explores the main part of our filesystem, the encryption and secure storage of the file contents. In the beginning, we define the problem, then follow two possible solutions and finally the design decision. At the end we discuss the integrity handling of entire file. Following definitions are used throughout this section.

- **block size** is 4096 bytes.
- **data block** is a block containing raw encrypted data.
- **plaintext block** is a block containing raw plaintext data.
- **master key** is used for the encryption of the entire filesystem, excluding file contents. Depending on the cipher used, it can either mean 256 bits (AES-GCM, ChaCha20Poly1305), or 512 bites (AES-SIV).
- **data encryption key** (DEK) is used for the encryption of file contents, each file is given a unique key. The length of DEKs is 256 bits.
- **key encryption key** (KEK) is used for encrypting the master key. The length of KEK is 256 bits.

Problem: Essential part of encrypted filesystems is the encryption of file contents, the secure usage of existing ciphers and the management of encrypted and auxiliary data used for correct decryption. The primary goal of this section is the the design of efficient layout of the storage medium, in our case, the underlying filesystem's file.

Common features: As we have mentioned previously, our filesystem is customizable to accommodate users with different needs. The main cipher used for encrypting the file contents is configurable between AES-GCM and ChaCha20Poly1305, both with key lengths of 256 bits. Providing acceptable security margins well into the future.

Encryption keys Every encrypted file contains randomly generated encryption key that is exclusively used for the encryption of plaintext blocks. There are multiple reasons for not using one master key for the whole filesystems. First, security, the ciphers used become inherently less secure the more invocations with the same key happen. When using randomly generated IVs it is recommended to rotate keys after invoking the ciphers around 2^{32} times [18]. Second, it enables the rotation of master key without having to reencrypt the whole file. The only necessary part to reencrypt is the data encryption key located in the header block of each file.

Header Block Both of the following layouts discussed contain a header block at the beginning of the file. A block containing information for its respective file, made of two parts, plaintext and ciphertext. Plaintext part contains the logical file size, a size when the file is decrypted, initialization vector and authentication tag of the ciphertext part. Ciphertext part contains data encryption key. See Figure 3.2 for more details. The header block could have been omitted in case of storing the data in the extended attributes of a file. In our case, the underlying filesystem may not support extended attributes. as such, we cannot rely on it.

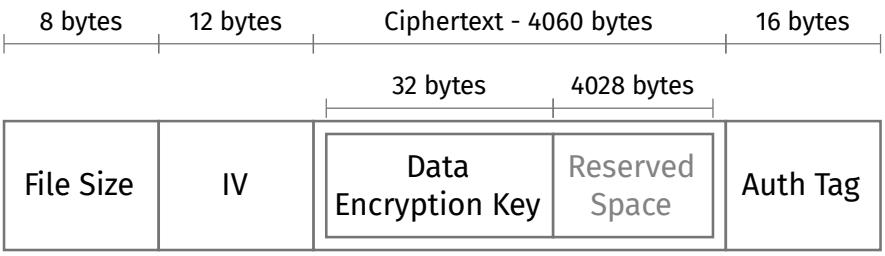

Figure 3.2: Header block layout

Block Oriented Layout: This design approach acknowledges that virtually all modern hardware organizes data into equally sized blocks. It closely resembles the underlying storage medium by aligning the data into blocks of 4096 bytes.

At the beginning of the file we have the previously mentioned header block. Followed by a group consisting of a **metadata block** and 146 data blocks. The metadata block contains initialization vectors and authentication tags that are required for decrypting the subsequent data blocks. The layout can be seen in Figure 3.3. It starts with initialization vector and authentication tag for the first data block, followed for the second data block, etc. The ciphers we selected use 12 bytes for IV and 16 bytes for authentication tag. At first glance, the count of data blocks being 146 may seem arbitrary, but in fact, it is the maximum number of initialization vectors and authentication tags that can fit inside one 4096 bytes large block, $\lfloor \frac{4096}{28} \rfloor = 146$. Unfortunately there is a slight space overhead of 8 bytes due to 28 not dividing evenly into 4096. The grouping of metadata block followed by 146 data blocks continues throughout the entire file. The layout of the file can be seen in Figure 3.4.

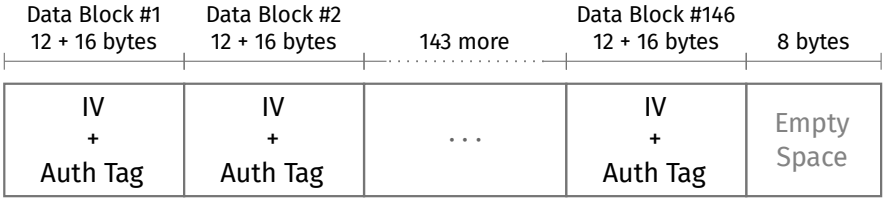

Figure 3.3: Metadata block layout

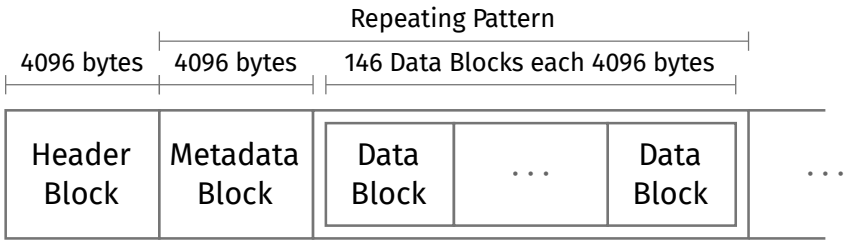

Figure 3.4: Block file layout

Stream Oriented Layout: Instead of closely resembling the underlying storage medium as was the case with the block oriented layout. This design approach is more focused on the CPU. Data locality is an important factor for performance critical systems. In the stream oriented layout, the data is organized inline as it is processed on a CPU. As previously mentioned, the file starts with the header block followed by data blocks in the form of `initialization vector | data block | authentication tag`. See Figure 3.5.

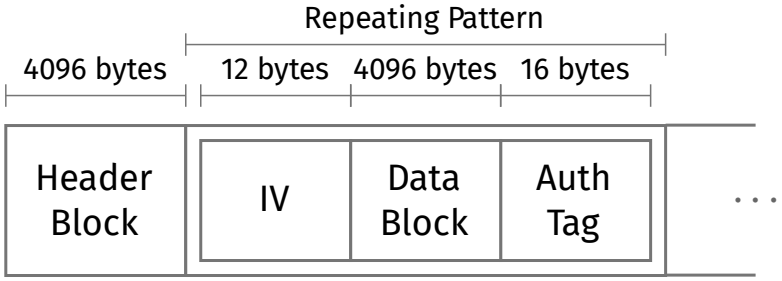

Figure 3.5: Stream file layout

Design decision: After evaluating the possible solutions, we reached to the conclusion of implementing both file layouts. The differences are not insignificant, and as such, they both may be useful for users with different needs.

Both layouts process the encrypted and decrypted data in chunks of 4096 bytes. As mentioned previously in their respective descriptions, the layouts handle the storage of initialization vectors and authentication tags differently. Block oriented layout stores them in metadata blocks, whereas stream oriented layout stores them inline, next to their respective data blocks.

3.2 File digests

For additional security, each file can have a digest computed from the encrypted data. The digest is checked, or updated when accessing, or writing to the file, respectively. It can catch various attacks such as permuting the encrypted blocks in a file, swapping blocks between different files, or replacing current blocks with their previous versions. Although increasing security, this approach may incur non-negligible computational overhead. To address this potential performance impact, users can disable this feature through a configuration flag. We discuss the user configuration later in Chapter 4.4 and computational overhead in Chapter 5.

Problem: In certain situations, the additional security from computing digests may be necessary. The digests are computed using data blocks. The current challenge is how to efficiently handle creation, storage, and deletion of digests, and how many data blocks to include in a single digest.

Store digest in the encrypted file: The first idea is to store the digests inside the encrypted file, be it in the metadata block for block oriented layout, or inline for the stream oriented layout. Both of these variants are challenging to implement, hard to extend and do not allow easy modification of data block count in a single digest, since the amount needs to be closely tied to the layout of metadata or digest block.

Store digest in a separate digest file: The second idea is to store the digests in an auxiliary file that only contains digests of a single file. This method is easily extensible, easier to implement and in case of no digest checking, does not incur any storage overhead.

Design decision: From the mentioned two alternatives, we have chosen the second one. Although the first alternative does not require any additional files, it contains significant drawbacks such as hardwiring the digest count into the metadata/digest block layout, and the storage overhead in case of disabled digest checking. The second alternative does not have any of these drawbacks and is simpler to implement. The count of encrypted data blocks that are included in a single digest is currently set to 28. The value is arbitrary, and further testing needs to be done to find the optimal value.

The amount of digests a file has depends on its size. **Digest index** is the digest position in the digest file. As an example, encrypted blocks from 0 up to 27 are included in digest with index 0, blocks 28 up to 55 in digest with index 1, and so on. Since the digests are saved in plaintext, to prevent adversaries from being able to compute them, we additionally use data encryption key of the corresponding file during the digest computation, To prevent permutation of digests, digest index is also added. The formula used for digest computation is $digest_i = HASH(DEK \mid i \mid data\ block_{28i} \mid data\ block_{28i+1} \mid ... \mid data\ block_{28i+27})$. By default, the hash function used for computing digests is BLAKE2, but can optionally be set to SHA3.

3.3 Directory Entry Encryption

This section is focused on the problem of encryption of directory entries. It starts with describing the problem, then goes through three solution alternatives, and at the end justifies the chosen solution.

Problem: Encrypted filesystems are expected to securely store the names of directory entries while maintaining acceptable performance for common operations. Specifically, the system must support efficient creation, lookup, and deletion of entries. The primary challenge is maintaining reasonable performance for common operations without sacrificing confidentiality or integrity.

AEAD encryption using ciphers from file encryption: If we have already designed the encryption of file data, why not extend it to directory entry encryption? The ciphers give us both confidentiality and integrity. Entry creation is simple. All that is required is to encrypt the name, convert it to valid filename format and

save the entry with produced ciphertext. Unfortunately, when it comes to querying and deleting entries, we cannot simply encrypt the name in question and ask the underlying filesystem. Due to how the ciphers work, each entry must have a unique initialization vector, which results in different ciphertexts when encrypting the same name. It leads to linear search through the directory, needing to decrypt each entry name and checking if it matches the given name. Such strategy becomes infeasible quickly when having to search directories containing hundreds to thousands of entries.

Deterministic AEAD using AES-GCM-SIV: Since we are already using AES-GCM in the file encryption, why not use AES-GCM-SIV? A variant of AES-GCM where the synthetic initialization vector is derived from the supplied initialization vector, plaintext, and associated data. Compared to plain AES-GCM, an initialization vector reuse does not completly break down the cipher. The only information that gets revealed when using the same initialization vector is in the case where the same plaintexts are encrypted multiple times. Unfortunately the cipher is designed for accidental IV reuse, not for intentional use of the same IV for multiple different plaintexts.

Deterministic AEAD using AES-SIV: So what about using AES in SIV mode [19]? This block cipher mode of operation has the same properties as AES-GCM-SIV. Where IV reuse reveals only that the same plaintexts are encrypted multiple times. Additionally it does not have the problem of cryptosystem breakage due to intentionally using same IVs. The downsides are that it is slower compared to AES-GCM and ChaCha20Poly1305 due to two-pass encryption. It also requires twice as long keys for comparable security which forces us to use different key length of 512 bits.

The problem, when using same IV, with revealing if there are two plaintexts encrypted multiple times implies that there are multiple directory entries with the same name. It is impossible in practice, as having two entries with the same name in one directory would lead to ambiguous file accesses.

Design decision: After analyzing the possible solutions, we selected Solution 3 (AES-SIV) as it provides the necessary deterministic encryption properties while being secure enough. Although AES-SIV is computationally slower and requires longer keys (512 bits vs 256 bits), it enables $\mathcal{O}(1)$ lookups, which is essential when accessing directories containing hundreds to thousands of entries.

Each directory contains its own randomly generated initialization vector. When creating a new entry in the given directory, we take the directory specific IV and its name, encrypt it, and use the resulting ciphertext as an entry name. Additionally, we need to encode it in a specific format that consists of allowed path characters. We chose the base64url [20], which is a filename safe encoding, as it contains only

readable characters and none of the special characters such as dots or slashes. When querying or deleting existing entry with a given name, we first take the directory IV, use it to encrypt the name, then base64url encode it and we ask the underlying filesystem if the entry with the given encrypted name exists, or in case of deletion, to delete it. Various filesystem implementations have a limit to the length of a path, usually `4096` bytes long. For the sake of simplicity, we only support lengths smaller than the limit.

3.4 Keys, Passwords, and Secrets

This section focuses on the handling of secrets. It starts with defining the problem. Then continues into providing possible solutions. Finally justifying the chosen solution.

Problem: We can employ any strong, impossible to break ciphers, with all the nice properties such as confidentiality, integrity, and authenticaton, but all of it is useless when the encryption key is saved in plaintext right next to the ciphertext. The challenge now is how to safely manage user supplied credentials such that it does not affect the overall security of our system.

Using user password as a key: The most simple and straightforward way is to ask a user for a password and use it for the encryption. Unfortunately ciphers expect keys of certain size which requires to pass the user password through a secure key derivation function that turns the variable user input into cryptographically secure fixed size output ready to be used with ciphers. Although simple, this method has its drawbacks. The encryption key is tied to the user password. In case of a password change, it would require to reencrypt the whole filesystem which may be possible for small amount of data, but gets computationally costly when encrypting hundreds to thousands of files.

Using randomly generated key: As we discussed in the previous paragraph, using user password for encryption is not feasible. Instead of using the user password for encryption of the whole filesystem, how about using it only for encrypting a master key? Here, the user password serves for deriving a key encryption key, which is then used for the encryption and decryption of the master key. This approach requires to store extra data, specifically, the randomly generated key and configuration for the key derivation function.

Design decision: We selected the second solution as it is more ergonomic to work with. The huge advantage of not having to reencrypt the whole filesystem in case of

a password change far outweighs any tradeoffs with having to save additional data. It is also more extensible into the future, as an example, when the filesystem is used by multiple users, each can have their own password. In the first solution, that would be impossible. With the second, all it requires is to save multiple encrypted copies of the master key where each copy is encrypted with the respective user's password. The cipher used for encrypting the master key is the same cipher used for encrypting the file contents.

Chapter 4

Implementation

This section documents the implementation process of our system. It is divided into four parts. The first part contains discussion about libraries used. The second part describes the structure of how we represent a cryptographic file. The third part describes problems and solutions during the implementation of the FUSE library wrapper, the component that defines common filesystem functions and communicates with the FUSE kernel module. Lastly, the fourth part is focused on the implementation of the main binary that the user interacts with.

4.1 Libraries

This section is divided into two parts. The first part contains discussion about the library facilitating the communication with the FUSE kernel module. The second part revolves around the discussion of library implementing the cryptographic primitives required for the implementation of our filesystem.

4.1.1 FUSE Library

As previously mentioned in Chapter 2, we are using the Rust programming language and the FUSE protocol. In order to communicate with the kernel side of FUSE, we require a library implementing the FUSE protocol. Multiple existing libraries implement the FUSE protocol in Rust, most notably:

- **fuse-rs** [21]: Appears to be one of the oldest FUSE libraries written in Rust. Unfortunately it looks to be unmaintained as the last commit, at the time of writing, is from 2020-07-29.
- **fuser** [22]: Seems to be a continuation of the original fuse-rs project. It appears to be well mainainted as there are recent commits.

- **FUSE-MT** [23]: Originally a multithreaded wrapper on top of fuse-rs which later switched to fuser due to fuse-rs being unmaintained. It provides additional features such as dispatching system calls on multiple threads so I/O does not block and translating inodes to paths which removes the burden of keeping track of inode and path pairs. It might also be unmaintained, as the last commit, at the time of writing, was on 2023-10-11.

In the end, we chose the fuser library as it appears to be the only library that is being actively maintained. We also do not need the additional features and the additional complexity of FUSE-MT. The handling of mapping between inodes and paths is explained later in Chapter 4.3.

4.1.2 Crypto Library

We also require a cryptography library. Rolling own crypto is heavily discouraged in the security industry since it is difficult to get it right and its easy to mess something up rendering the entire system cryptographically insecure. The best practice is to look for already established and security vetted projects implementing the constructs we need, the libraries such as:

- **ring** [24]: Touting itself as safe, fast, small crypto for Rust. It contains a small subset of most commonly used cryptography algorithms. Parts of the library are written in assembly and C, and parts are in Rust. The usage of assembly and C is due to the code being faster than comparable Rust code. But as the Rust compiler and ecosystem gets better, the goal is to eventually replace everything with Rust[1].
- **RustCrypto project** [25]: Contains multiple repositories defining common traits and implementations of the specific algorithms:
  - AEAD repository: Contains ciphers concerning the authenticated encryption with associated data such as the implementations of the AES-GCM, ChaCha20Poly1305 and AES-SIV ciphers.
  - hashes repository: Contains commonly used hash algorithms such as BLAKE2, SHA2, SHA3.
  - password-hashes repository: Contains password hashing and KDF algorithms such as argon2, scrypt, PBKDF2.

After the analysis of both projects implementing the cryptographic algorithms. We chose the RustCrypto project. The ring library, although small and comprising of

[1] At the time of writing the library was still maintained. Shortly afterwards the author posted about an indefinite break. Now it seems to be maintained again and described as “An experiment”.

faster and more optimized algorithms, contains significant drawbacks such as the absence of algorithms we need, in this particular instance BLAKE2 and argon2, and the uncertainty of its future. The RustCrypto project contains all the required algorithms we need, purely implemented in Rust, and overall having greater integrations between the different libraries. The only minor drawback is worse performance of algorithms due to not having optimized C and assembly code.

4.2 Cryptographic file

The cryptographic file is the main structure powering our filesystem. It handles the transparent encryption and decryption of data. Seamlessly integrates into the Rust ecosystem by implementing common I/O traits, specifically `Read` and `Write`, that enable the encrypted file to be used the same way as a regular file. The file is generic over the cipher, digest, and mode. where cipher is the cipher used for encryption, digest is the digest used for optional integrity checking, and mode is the layout of the encrypted data.

```
1  struct CryptoFile<C: Cipher, D: Digest, M: Mode>
2  {
3      file: File,
4      digest_file: Option<File>,
5      cache: FileCache,
6      // snip
7  }
```

Listing 4.1: Snippet of `CryptoFile`

The `CryptoFile` struct encapsulates two regular files. The first contains the encrypted data, the second is optional and contains the digests used for integrity checking. The struct manages the reads and writes of those files. Working with files backed by slower hardware requires some kind of cache, otherwise, especially in our situation of encryption, the I/O would be excruciatingly slow. As such, the struct also contains a `FileCache` that handles the caching of current working block. When accessing the current block, it delegates the I/O to the cache. When accessing different block than the currently cached one, depending on if the block was written to, it either needs to encrypt the cached contents, and write them to the file, or simply load the requested new block, decrypt it, save it in the cache, and return the data.

```
1 struct FileCache {
2     block_idx: u64,   // Block index
3     block: Block,     // Block data, alias to [u8; 4096]
4     dirty: bool,      // Indicator if `block` was written to
5     // snip
6 }
```

Listing 4.2: Snippet of `FileCache`

The reading and writing is handled through the functions of the `Read` and `Write` traits, `fn read(&mut self, mut buf: &mut [u8])` and `fn write(&mut self, mut buf: &[u8])` respectively. Those functions, depending if the block is not in the cache, delegate the work to the `fn update_cache(&mut self)` that handles the loading and unloading, and encryption and decryption of the blocks.

4.3 FUSE wrapper

The FUSE wrapper facilitates the communication between our code and the kernel FUSE module. It is implemented using the fuser library which requires us to implement functions of the `Filesystem` trait. This section focuses on the implementation decisions of filesystem functions. The list is non-exhaustive. Most of these functions have their respective entries in the Linux man-pages.

`fn lookup()` A function that checks if a filename at a given path exists, returns an inode number on success. In order to keep track of individual files we have to somehow uniquely identify each file. We could enumerate each file as we receive lookup calls from the kernel. Although simple in theory, in practice we either have to keep track of all files' indices each time we mount the filesystem, or have them stored in a file somewhere. Keeping track of indices after each mount also complicates the handling of hard links, as remounting the filesystem may result in different file enumeration, and the old links would point to the now invalid enumeration.

Instead we have chosen to reuse the underlying filesystem inodes. The uniqueness is guaranteed by the underlying filesystem. That solves both of the issues with the previous solution of enumerating the files ourselves. The inodes stay the same after remounting the filesystem and there is no need to store the file inode anywhere. When requiring hard links, we reuse the underlying filesystem's ability to create hard links, see `fn link()` description for more information.

`fn read()` and `fn write()` The reading from and writing to files is handled through the previously covered `CryptoFile` which manages the encryption and decryption of file contents. The FUSE functions merely delegate the work to it.

`fn link()` Creates a hard link of a file. Our filesystem handles hardlinks the same way as the underlying filesystem. When creating hardlinks, it delegates the work to the underlying filesystem `link()` which links the specified encrypted and digest files.

`fn symlink()` Creates a symbolic link in our filesystem. We have a couple possible ways to implement the symbolic link storage.

Store inside a special file: The idea is to have special files similar to the way digests are stored, but instead having a `.ln` extension signifying that the file contains a symbolic link inside. It would require us to redesign the handling of lookups and overall the storage of file attributes, as currently we reuse the underlying file object attributes.

Reuse underlying filesystem symlinks: Saves the path that the link is pointing to in the underlying filesystem symlink. This way the implementation is really simple, it only requires us to call a symlink function on the underlying filesystem This solution takes advantage of the fact that symbolic links do not have to point to a valid location, as such, we can encode arbitrary data inside of it.

Implementation Decision: Due to simplicity we have decided to implement the second version, as it does not require us to change the existing code and all the symbolic link handling is delegated to the underlying filesystem. The actual implementation is as follows: We receive call to symlink with the link name and target path that it points to. We encrypt the symlink name same way as we handle the directory entry encryption, see Chapter 3.3 for more information. The target path is encrypted with the file encryption cipher using the master key, and finally encoded to base64url encoding since it needs to be a readable string. With these two encrypted strings, we call the symlink syscall to create a symlink in the underlying filesystem.

`fn readlink()` Reads symbolic links. The kernel expects a non-encrypted path that the symbolic link points to, which is then resolved by the specific filesystem that the symbolic link points to. With the way that the `fn symlink()` is implemented, all that is required is to base64url decode the symlink target path, decrypt it and return the plaintext path to the kernel. It then resolves that path by calling regular functions such as `fn readdir()` and `fn open()`, in case of directory and regular file, respectively.

`fn unlink()` Removes files. Since the operation is destructive, accidentally receiving a path corresponding to a file outside of our encrypted root would lead to unwanted deletion of arbitrary files. As such, we decided that during the initialization of the filesystem, the program changes its root to `encrypted_root` that is received from the user. However, this solution requires the compiled binary to either be run as root, or in better case, only have the `CAP_SYS_CHROOT` capability set. Additionally, it simplifies the path handling in other functions, since all paths now start at root `/`.

`fn fallocate()` Corresponds to the Linux-specific `fallocate()` syscall which manipulates the allocated disk space of a file. Our filesystem supports only the default mode of operation equivalent with the `posix_allocate()` function. It does not support any of the advanced operations such as hole punching using `FALLOC_FL_PUNCH_HOLE` or collapsing file space using `FALLOC_FL_COLLAPSE_RANGE`. It would complicate the encryption handling while not being that useful.

`fn setattr()` and `fn getattr()` Sets and lists the file attributes such as access and modification time, file permissions, owner and group of a file. For simplicity, we reuse the underlying file attributes. In case of setting attributes, we change the underlying file attributes. In case of retrieving attributes, we call the `lstat()` syscall on the underlying file object.

`fn setxattr()`, `fn getxattr()`, `fn listxattr()`, `fn removexattr()` The underlying filesystem may not support extended attributes, as such we do not support them either. The situations in which extended attributes are not available may be if the underlying filesystem is some kind of network filesystem, those usually do not support extended attributes.

4.3.1 FUSE specific settings

This part covers some of the FUSE settings and oddities. The first part covers the caching of data, in particular the caching modes such as writeback and write-through. The second part covers the FUSE file access mechanism.

FUSE I/O modes FUSE supports several caching strategies with different performance implications. The direct-io mode does not perform any caching, it bypasses the page cache completely and does not do any read-ahead. The cached mode makes use of page cache. The kernel may perform read-ahead and keeps the cache in consistent state. The cached mode is divided into two submodes depending on how the writes are handled. The default behavior is write-through where each write gets immediately sent to our filesystem as several write requests, as well as updating any cached pages. The other mode is writeback, where the writes only go to cache. The dirty pages are written back at a later

time, either implicitly during background writeback or page reclaim, or explicitly when calling `close()`, or `fsync()`.

After analysis of the available caching strategies. We have decided to use the default mode of write-through. In case of reading, the data read-ahead is beneficial as there are less context switches since it caches certain amount of data and does not have to call our implementation for more data. In case of writes, the write-through does not create any inconsistent states and the written data always gets encrypted.

FUSE permissions The permission handling of FUSE depends on options supplied during the initialization. By default the filesystem is only accessible to the user mounting the filesystem. The options are:

- `default_permissions` - By default FUSE kernel module does not check file access permissions and the behaviour depends on the implementation. When set, kernel performs permission checking based on file mode.
- `allow_root` - This option allows root user to also access the filesystem.
- `allow_other` - This option allows any user to also access the filesystem.

Using `allow_root` or `allow_other` options can lead to unauthorized file access to the directory containing the encrypted root of our filesystem. The way our filesystem is implemented, mounting user needs to have permissions to access the encrypted files as it delegates the file access to the underlying filesystem. Since we delegate the permission checking to the underlying filesystem file access mechanism, `default_permission` is left unset. The options `allow_root` and `allow_other` can optionally be set to allow root and other users, respectively, access to the mounted filesystem.

4.4 User facing interface

The following section covers the design and implementation decisions of the user interface. At the beginning we describe the requirements of the interface. Afterwards, we move onto the implementation.

For the ease of implementation, we are using the clap library [26], short Command Line Argument Parser. A library that greatly simplifies the implementation of the command line interface, as it handles everything from parsing, validating input, and generating code and documentation. The interface to our filesystem should expose functions for initialization, mounting, and configuring certain properties, such as changing the password, and turning on/off integrity checks. We define all of the mentioned operations as subcommands, where the main command is the executable of our filesystem. The subcommands are:

- `init --cipher <cipher> --digest <digest> --mode <mode> <enc_dir>` - initializes the encrypted filesystem in a directory `dir`, with the specified cipher, digest, and mode of operation
- `mount -o <fuse_options> <enc_dir> <mount>` - mounts the decrypted data at `mount`, `enc_dir` needs to be a valid initialized encrypted root, `fuse_options` contains the FUSE specific options.
- `passwd` - changes the currently used password.
- `status <enc_dir>` - if `<enc_dir>` is a valid encrypted root, lists information about it.
- `digest --check <bool>` - turns on and off digest checking.

Problem: All of the listed commands require a password. The primary challenge now is to implement the handling of passwords without compromising on the security and usability.

Command line argument: This method exposes additional argument in our command line interface in the form of `--password <password>`. Allowing the password to be specified at command invocation enables easier automation and testing. Unfortunately, it also brings reduced security in the form of potential password leakage. Since the password needs to be specified at command line, it may be logged in a history file, stored in various automation scripts, or can be seen in the respective `/proc/` entry.

Prompt user for password: Upon running a subcommand that requires a password, it prompts the user to enter the password. Although better for security as the password never leaves our program, the non-existent support for automation and testing is in certain situations a major deal-breaker.

Implementation decision: After thorough analysis of the two methods. We have decided to implement both of them, as each serves a different purpose. The first method, although worse from the security standpoint, enables easier automation and testing of our filesystem. On the other hand, the second method contains better security at the expense of automation and testing. In most situations, the second method should be preferred.

```rust
Init { /// Initialize the filesystem
    /// Password used for encryption of the filesystem
    password: Option<String>,
    /// Directory where to initialize the encrypted filesystem
    encrypted_root: String,
    /// Cipher algorithm used for encryption
    cipher: Cipher,
    /// Digest algorithm used for checking integrity of a file
    digest: Digest,
    /// File block aiignment
    mode: Mode,
} // snip
```

```rust
Mount { /// Mount the filesystem
    /// Password used for encryption of the filesystem
    password: Option<String>,
    /// Directory containing encrypted filesystem
    encrypted_root: String,
    /// Path where to mount the decrypted filesystem
    mount_point: String,
    /// FUSE options
    fuse_options: String
} // snip
```

Listing 4.3: Snippet of Subcommands

We decided to name the command line tool `fscryptrs`, it contains a help which can be accessed with `help` subcommand. Example usage:

- `fscryptrs help` - shows general help with available subcommands.
- `fscryptrs help <subcommand>` - shows specific description of the supplied subcommand.
- `fscryptrs help init` - shows help for init subcommand, containing possible flags, values and their default values if not specified.

More can be seen in Appendix A which contains the source code of the entire implementation.

Chapter 5

Benchmarks

This chapter covers the benchmarks and comparisons with existing solutions. Initially we look at the performance of our filesystem's different ciphers and modes. Afterwards, we compare the fastest variant of our filesystem with existing solutions.
Disclaimer: The following benchmarks are in no way thorough. They are simple in nature and for informational purposes only. The primary goal of this thesis is to create a protected filesystem, not to thoroughly test multiple filesystems.

All the benchmarks can be found in the main repository of the filesystem in the `benchmarks` directory. It contains main and helper scripts for running them. The following benchmarks were performed on tmpfs on a system with `Intel Core i7 7700k` CPU and `32GB 3000MHz DDR4` RAM.

5.1 Extracting Archive

The main benchmark performed is the extraction of the Linux Kernel archive to a mounted encrypted filesystem. It is a good indicator of how performant the filesystem is with files of various sizes. The main benchmark used is `tar xzf ./linux-6.0.tar.gz -C ./mnt`. The software used for running a repeatable set of benchmarks is `hyperfine` [27]. The main purpose of the tool is benchmarking, more specifically, measuring the running time of software. It contains configurable options such as specifying the amount of runs and warmup runs to do, commands to run before and after the run. Warmup runs are usually done to prime the cache with data.

5.1.1 Comparing Different Modes

As can be seen in Chart 5.1, unsurprisingly, our filesystem is fastest when digest checking is turned off, a bit slower with Blake2 and finally, slowest with SHA3.

Additionally the bar chart shows that in every case block mode of data alignment is slightly faster compared to stream mode. Also, the hardware that the benchmarks were performed on contains `AES-NI` instructions for efficient AES computation which is consistent with the results.

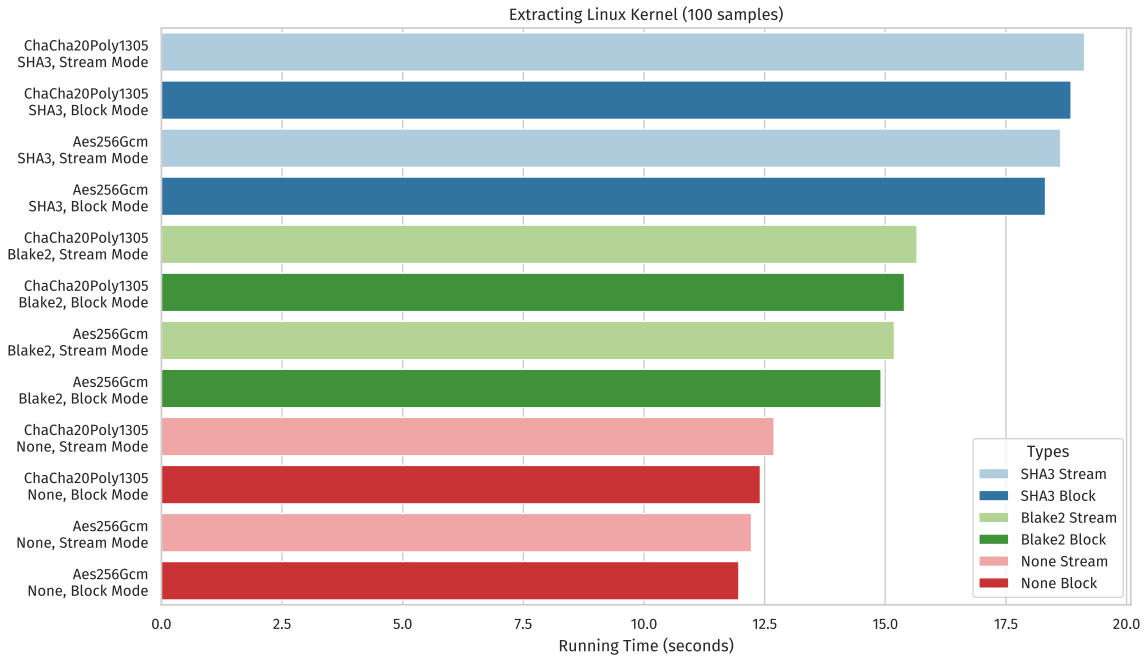

Chart 5.1: Results of different modes of our filesystem

5.1.2 Comparing with Existing Filesystems

The following chart shows comparison between the fastest modes of our filesystem with gocryptfs and no filesystem which is labeled as default. As can be seen, our filesystem is faster when handling a lot of smaller files.

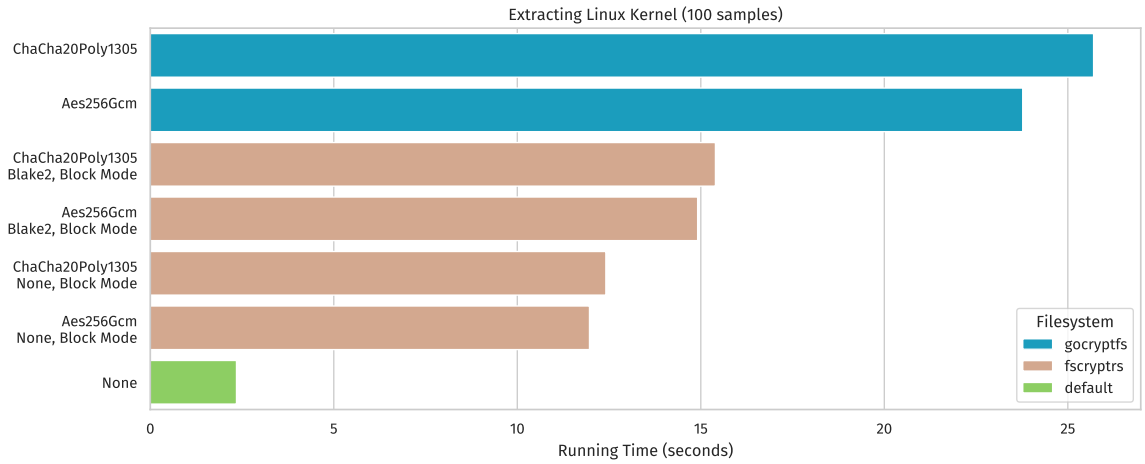

Chart 5.2: Comparing fastest modes with others

5.2 Single File Performance

Previous benchmark focused on the performance of the filesystem when accessing multiple smaller files. The primary goal of the following benchmark is to test single file sequential read and write performance by reading or writing a large amount of data. The command used for benchmarking is the simple Linux utility for converting and copying files `dd`.

The following charts show single file read and write speed. The benchmarks were split into two. The first benchmark tests file I/O when reading and writing a single block of 1GB size, see Chart 5.3. On the other hand, the second benchmark tests reading and writing 1024 blocks of 1MB size, see Chart 5.4. The results show that our filesystem is slower than gocryptfs and, as expected, slowest when digest checking is enabled. There is plenty of room for improvement, especially in the way that our filesystem handles reading and writing to the underlying filesystem. Chapter 6 contains more discussion about future improvements.

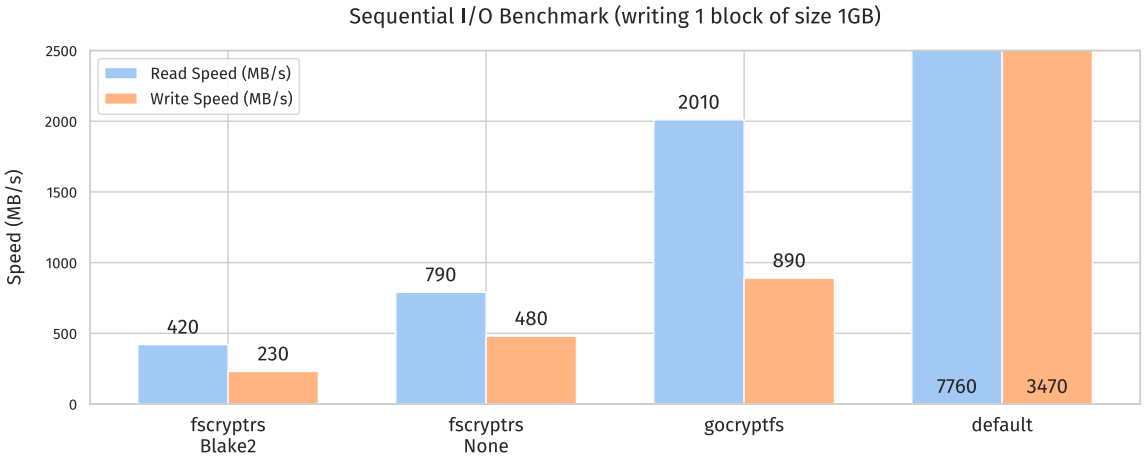

Chart 5.3: Single File Read/Write of 1 block of size 1GB

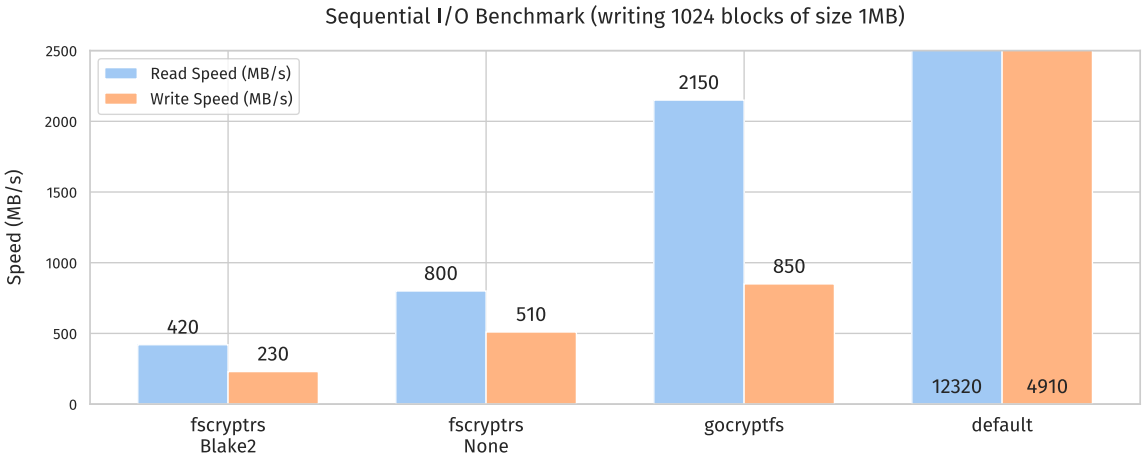

Chart 5.4: Single File Read/Write of 1024 blocks of size 1MB

Chapter 6

Future Work

During the implementation we have made several decisions that warrant a redesign to make it cleaner, more extensible, reduce bugs and increase performance.

Better data caching Currently the filesystem caches one data block of `4096` bytes. We propose to cache a variable amount of data blocks, depending on the length of the buffer kernel requested. Additionally, when reading and writing, write multiple blocks at once, instead of currently writing one block at a time. It would lead to less context switching and increased performance.

Better digest handling Decouple `FileCache` struct and digest handling. Optionally, when no digest checking is performed, there should not be any code paths mentioning digests. Also, design a generic `Filesystem` that accepts `None` as digest to disable digest checking. Currently, there is a boolean flag controlling the behaviour.

Extended attributes support When the underlying filesystem supports extended attributes, use those for storage of directory initialization vectors, encrypted keys, and also implement common functions to support ACLs and other Linux features that use extended attributes.

Long Filename Handling Design and implement support for filenames exact or longer than `255` bytes. Currently, there is no mechanism to handle filenames over the limit, and some longer filenames may silently fail due to the extra length of IVs, authentication tags, and base64url encoding overhead.

Profiling Profiling and benchmarking is an important part of software optimization. With tools such as Linux's perf [28] and DTrace [29] which collect profiling events that can later be analyzed with flamegraph [30] and others. Flamegraph neatly visualizes hot code paths which provide starting point for further optimization of the software.

Testing Improve and write more unit and integraton tests to achieve greater test code coverage. Also make the filesystem work with testing and quality assurance suites such as xfstests [31].

Conclusion

We successfully designed and implemented a FUSE-based protected filesystem for Linux while adhering to the requirements defined in advance in Chapter 2. Also, we performed some benchmarks, as can be seen in Chapter 5, and found that the filesystem has comparable performance with existing solutions.

During design, the primary challenge consisted of how to design a secure system out of the cryptographic primitives. Additionally, the task of how to efficiently access files and directories. Finally, how to design digest checking of entire files, such that it can optionally be turned off, if greater performance is required.

The implementation phase was not without issues. The cryptographic file, in particular, the data caching part of it, is not as performant as we hoped and needs redesign in the future.

Development relied heavily on the Rust programming language and various essential libraries to create the secure FUSE-based filesystem.

Overall, we have created a protected filesystem for Linux, which is performant enough. In the future, it can be expanded upon with various testing and profiling work to make it better and faster, using tools such as xfstests and flamegraph.

Bibliography

[1] "libfuse API documentation." [Online]. Available: https://libfuse.github.io/doxygen/

[2] NIST, "NIST SP 800-53 Rev. 5." [Online]. Available: https://csrc.nist.gov/publications/detail/fips/202/final

[3] M.-J. O. Saarinen and J.-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)." [Online]. Available: https://www.rfc-editor.org/info/rfc7693

[4] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0." [Online]. Available: https://www.rfc-editor.org/info/rfc2898

[5] N. Provos and D. Mazières, "A Future-Adaptable Password Scheme." [Online]. Available: https://www.usenix.org/legacy/events/usenix99/provos/provos.pdf

[6] C. Percival and S. Josefsson, "The scrypt Password-Based Key Derivation Function." [Online]. Available: https://www.rfc-editor.org/info/rfc7914

[7] A. Biryukov, D. Dinu, D. Khovratovich, and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications." [Online]. Available: https://www.rfc-editor.org/info/rfc9106

[8] vgough, "EncFS GitHub Repository." [Online]. Available: https://github.com/vgough/encfs

[9] T. Hornby, "EncFS Security Audit." [Online]. Available: https://defuse.ca/audits/encfs.htm

[10] rfjakob, "gocryptfs website." [Online]. Available: https://nuetzlich.net/gocryptfs

[11] B. Lord, "The Urgent Need for Memory Safety in Software Products." [Online]. Available: https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products

[12] rfjakob, "gocryptfs design." [Online]. Available: https://nuetzlich.net/gocryptfs/forward_mode_crypto/

[13] PostgreSQL, "Database Page Layout." [Online]. Available: https://www.postgresql.org/docs/current/storage-page-layout.html

[14] "Linux Unified Key Setup GitLab Repository." [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup

[15] "BitLocker Documentation." [Online]. Available: https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/

[16] "The Rust Programming Language Book." [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#ownership-rules

[17] "The Rust Programming Language Book." [Online]. Available: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html#the-rules-of-references

[18] M. Dworkin, "NIST SP 800-38D ." [Online]. Available: https://csrc.nist.gov/pubs/sp/800/38/d/final

[19] D. Harkins, "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)." [Online]. Available: https://www.rfc-editor.org/info/rfc5297

[20] S. Josefsson, "The Base16, Base32, and Base64 Data Encodings." [Online]. Available: https://www.rfc-editor.org/info/rfc4648

[21] zargony, "fuse-rs GitHub Repository." [Online]. Available: https://github.com/zargony/fuse-rs

[22] cberner, "fuser, a Rust FUSE library." [Online]. Available: https://github.com/cberner/fuser

[23] wfraser, "FUSE-MT GitHub Repository." [Online]. Available: https://github.com/wfraser/fuse-mt

[24] B. Smith, “ring GitHub Repository.” [Online]. Available: https://github.com/briansmith/ring

[25] “RustCrypto GitHub Organization.” [Online]. Available: https://github.com/RustCrypto

[26] clap-rs, “clap-rs GitHub Repository.” [Online]. Available: https://github.com/clap-rs/clap

[27] sharkdp, “hyperfine GitHub Repository.” [Online]. Available: https://github.com/sharkdp/hyperfine

[28] “perf, Linux profiling with performance counters.” [Online]. Available: https://perfwiki.github.io/main/

[29] “The DTrace Toolkit: A set of scripts for use with DTrace on various systems. .” [Online]. Available: https://github.com/opendtrace/toolkit

[30] B. Gregg, “flamegraph, Stack trace visualizer.” [Online]. Available: https://www.brendangregg.com/flamegraphs.html

[31] “xfstests, FSQA Suite.” [Online]. Available: https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/about/

Appendix A

Filesystem Source Code

The digital attachment contains a repository with the source code of the filesystem described in this thesis. The building and running of the software is described in the `README.md` file.

Appendix B

Benchmarks and Results

The digital attachment, also contains a repository with the benchmark scripts mentioned in Chapter 5. The visualizations are in the form of jupyter notebooks, and can be viewed in various tools supporting the `.ipynb` file format. Additionally, it contains benchmark results in the `benchmarks/archive_extract` and `benchmarks/single_file_performance` for the Linux Kernel archive extraction and single file performance, respectively.