



Berliner Hochschule für Technik  
Fachbereich VI - Informatik und Medien  
Studiengang Medieninformatik

**Implementierung einer transparenten  
Verschlüsselung auf Linux-Basierten  
Betriebssystemen**

**Masterarbeit**  
zur Erlangung des akademischen Grades  
**Master of Engineering (M.Eng.)**

Sarmad Ahmad (Matrikelnummer: 940714)  
geboren am 28.07.1987 in Bagdad

Betreuer: Prof. Dr. Christian Forler  
Gutachter: Prof. Dr.-Ing. Thomas Loewel

Abgabetermin: 18.11.2024

## **Vorwort**

Diese Arbeit ist das Ergebnis meiner intensiven Beschäftigung mit einem Thema, das in der heutigen digitalen Gesellschaft immer wichtiger wird, der Schutz sensibler Daten. Die Entscheidung, mich mit der Entwicklung eines Systems zur transparenten Verschlüsselung zu befassen, basiert sowohl auf meiner persönlichen Faszination für IT-Sicherheit als auch auf der Überzeugung, dass der Datenschutz für alle zugänglich und unkompliziert sein sollte.

Während des Forschungs- und Entwicklungsprozesses, habe ich nicht nur fachlich viel gelernt, sondern auch meine eigenen Grenzen kennengelernt und überwunden. Die Arbeit hat mir gezeigt, wie viel Beharrlichkeit und Detailgenauigkeit erforderlich sind, um eine Lösung zu schaffen, die sowohl technisch anspruchsvoll als auch nutzerfreundlich ist.

# Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese MASTERARBEIT selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Sarmad Ahmad, Berlin den 17.11.2024  
Sarmad Ahmad

## **Danksagung**

**Mein besonderer Dank** gilt einer außergewöhnlichen Frau, die mit unermüdlicher Stärke drei Kinder als berufstätige und alleinerziehende Mutter großgezogen hat. Sie hat nicht nur allen Herausforderungen des Lebens getrotzt, sondern uns auch immer wieder gezeigt, was es bedeutet, bedingungslos zu unterstützen und niemals aufzugeben. Ohne ihre Hilfe, ihren Glauben an uns und ihre Liebe wären wir nicht dort, wo wir heute sind.

Danke, Mama.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>v</b>
<b>Tabellenverzeichnis</b>	<b>vi</b>
<b>Verzeichnis der Listings</b>	<b>vii</b>
<b>Abkürzungsverzeichnis</b>	<b>viii</b>
<b>1. Einleitung</b>	<b>1</b>
1.1. Problemstellung und Motivation . . . . .	1
1.2. Ziel der Arbeit . . . . .	2
1.3. Aufbau der Arbeit . . . . .	2
<b>2. Grundlagen</b>	<b>4</b>
2.1. Filysystem in Userspace (FUSE) . . . . .	4
2.2. Verschlüsselung . . . . .	8
2.3. Catena als KDF . . . . .	20
2.4. Pluggable Authentication Modules (PAM) . . . . .	23
<b>3. Konzept</b>	<b>25</b>
3.1. Softwareanalyse . . . . .	26
<b>4. Implementierung</b>	<b>30</b>
4.1. Automatisierung und Sicherheit durch Python und PAM . . . . .	30
4.2. Catena als KDF: Zielsetzung und Sicherheit . . . . .	34
4.3. FUSE Implementierung . . . . .	36
4.4. Implementierung von POET in FUSE . . . . .	38
<b>5. Ergebnisse</b>	<b>54</b>
5.1. Analyse und Evaluation . . . . .	54
<b>6. Zusammenfassung und Ausblick</b>	<b>60</b>
6.1. Zusammenfassung der Ergebnisse . . . . .	60
6.2. Wesentliche Erkenntnisse . . . . .	61
6.3. Fazit . . . . .	63
<b>A. Sourcecode von <code>bb_write</code> und <code>bb_read</code></b>	<b>70</b>
<b>B. Schreibgeschwindigkeit Screenshots</b>	<b>77</b>
<b>C. Lesegeschwindigkeit Screenshots</b>	<b>79</b>

# Abbildungsverzeichnis

2.1. FUSE-Architektur [30, S.9] . . . . .	6
2.2. AES-Blockchiffre[21, S.113] . . . . .	9
2.3. Struktur vor dem Verschieben der Blöcke, Runde null . . . . .	11
2.4. Struktur nach dem Verschieben der Blöcke, Runde eins . . . . .	11
2.5. MixColumns Konstante . . . . .	12
2.6. Schlüsselfahrplan für AES in der ersten Runde[21, S.125] . . . . .	14
2.7. ECB-Mode[13, S.90] . . . . .	16
2.8. CBC-Mode[13, S.91] . . . . .	16
2.9. POET-ProcessHeader[7, S.17] . . . . .	18
2.10. POET-Block[7, S.14] . . . . .	19
3.1. Komponentendarstellung SafeBayFS . . . . .	25
4.1. Terminalausgabe vor dem Mounten . . . . .	39
4.2. Terminalausgabe nach dem Mounten . . . . .	39
4.3. FUSE und POET . . . . .	48
4.4. Ablauf von SafeBayFS . . . . .	52
5.1. Hexdump von zwei Dateien mit identischen Inhalt . . . . .	55
B.1. Ubuntu Schreibgeschwindigkeit ohne Fuse . . . . .	77
B.2. Ubuntu Schreibgeschwindigkeit mit FUSE . . . . .	77
B.3. Ubuntu Schreibgeschwindigkeit mit SafeBay . . . . .	78
C.1. Ubuntu Lesegeschwindigkeit ohne FUSE . . . . .	79
C.2. Ubuntu Lesegeschwindigkeit mit FUSE . . . . .	79
C.3. Ubuntu Lesegeschwindigkeit mit SafeBayFS . . . . .	79

# Tabellenverzeichnis

2.1. AES-S-Box: Substitutionswerte in hexadezimaler Notation für die Eingabebytes (xy) . . . . .	10
5.1. Schreibgeschwindigkeit Ubuntu ohne FUSE . . . . .	58
5.2. Schreibgeschwindigkeit Ubuntu mit FUSE . . . . .	58
5.3. Schreibgeschwindigkeit Ubuntu mit SafeBay . . . . .	58
5.4. Lesegeschwindigkeit Ubuntu ohne FUSE . . . . .	59
5.5. Lesegeschwindigkeit Ubuntu mit FUSE . . . . .	59
5.6. Lesegeschwindigkeit Ubuntu mit SafeBay . . . . .	59

# Verzeichnis der Listings

4.1. PAM-Konfiguration von common-auth <code>pam_stdin.py</code> . . . . .	31
4.2. PAM-Konfiguration für die Sitzungsphase mit Python-Skript . . . . .	32
4.3. Ändern der Benutzerrechte in Python mit UID und GID . . . . .	33
4.4. Definition des Salt-Werts in Hexadezimaldarstellung . . . . .	35
4.5. FUSE Operationen in <code>bb_oper Struct</code> . . . . .	38
4.6. POET-Verschlüsselungs- und Entschlüsselungsprozess . . . . .	40
4.7. Funktion <code>bb_write keysetup</code> und Blocknummerberechnung der Daten . . . . .	42
4.8. Generierung und Verarbeitung einer neuen Nonce . . . . .	42
4.9. Padding-Implementierung . . . . .	43
4.10. Kein überschreiben, bereits verschlüsselter Blöcke . . . . .	43
4.11. Blockverschlüsselung und Tag-Generierung . . . . .	44
4.12. Überprüfung und Anpassung, der Dateigröße sowie Berechnung der Start- und Endblöcke . . . . .	45
4.13. Berechnung des verschlüsselten Dateioffsets <code>current_pos</code> . . . . .	46
4.14. Block entschlüsseln und Tag überprüfen . . . . .	47
4.15. Padding im entschlüsselten Puffer finden und entfernen . . . . .	47
 A.1. Die vollständige <code>bb_write</code> Funktion . . . . .	 70
A.2. Die vollständige <code>bb_read</code> Funktion . . . . .	73



# Abkürzungsverzeichnis

<b>GPU</b>	Graphics Processing Unit
<b>CPU</b>	Central Processing Unit
<b>FUSE</b>	Filysystem in Userspace
<b>POET</b>	Pipelineable On-line Encryption with authentication Tag
<b>AE</b>	Authenticated Encryption
<b>KDF</b>	Key-Derivation-Funktion
<b>PC</b>	Personal Computer
<b>VFS</b>	Virtual File System)
<b>SSH</b>	Secure Socket Shell
<b>AES</b>	Advanced Encryption Standard.
<b>SPN</b>	Substitution-Permutation-Netzwer
<b>S-Box</b>	Substitutions-Box
<b>RC</b>	Rundenkoeffizienten
<b>XOR</b>	Exklusive-ODER-Gatter
<b>ECB</b>	Electronic Codebook
<b>CBC</b>	Cipher Block Chaining
<b>IV</b>	Initialisierungsvektors
<b>NONCE</b>	Number used once
<b>MAC</b>	Message Authentication Codes
<b>TLS</b>	Transport Layer Security
<b>GCM</b>	Galois/Counter Mode)
<b>OCB</b>	Codebook Mode)
<b>AEAD</b>	uthenticated Encryption with Associated Data
<b>TLS</b>	Transport Layer Security
<b>PBKDF2</b>	Password-Based Key Derivation Function 2
<b>DBH</b>	Double-Butterfly-Hashing-Struktur
<b>FPGA</b>	TField Programmable Gate Array
<b>RAM</b>	Random-Access Memory
<b>ASan</b>	AddressSanitize

# 1. Einleitung

Mit der rasanten Entwicklung digitaler Technologien und der wachsenden Anzahl an Anwendungen, die sensible Informationen verarbeiten, nimmt die Bedeutung des Datenschutzes stetig zu. Sowohl im privaten als auch im beruflichen Umfeld, verlassen sich Menschen zunehmend auf Computer und digitale Speichermedien, um wichtige und persönliche Daten sicher zu verwahren. Während Verschlüsselungstechniken schon lange existieren, werden sie häufig aufgrund ihrer Komplexität nicht ausreichend genutzt, was Sicherheitsrisiken begünstigt.

Die Arbeit befasst sich mit der Entwicklung eines benutzerfreundlichen Systems, das auf modernen kryptografischen Methoden basiert und in den alltäglichen Umgang mit digitalen Verzeichnissen integriert werden kann. Der Fokus liegt dabei darauf, eine Lösung zu schaffen, die ohne zusätzliche technische Anforderungen oder Eingriffe verwendet werden kann und so auch Nutzern ohne tiefgehendes Fachwissen ermöglicht, ihre Daten zuverlässig zu schützen.

In den folgenden Kapiteln werden die theoretischen Grundlagen, die Anforderungen und der Entwicklungsprozess dieser Lösung detailliert beschrieben. Ziel ist es, einen umfassenden Einblick in die Methodik und Umsetzung zu geben und damit einen Beitrag zu leisten, wie moderne Verschlüsselungstechniken auf einfache und transparente Weise in alltägliche Systeme integriert werden können.

## 1.1. Problemstellung und Motivation

Im digitalen Zeitalter, speichern immer mehr Menschen persönliche und sensible Daten auf ihren Geräten. Sei es auf dem privaten Computer oder in gemeinsam genutzten Systemen. Diese Daten umfassen oft vertrauliche Informationen wie finanzielle Dokumente, Gesundheitsdaten oder persönliche Korrespondenz, die einen hohen Schutzbedarf haben. Allerdings fehlt vielen Nutzern das technische Verständnis für Verschlüsselung und die Einrichtung sicherer Speichersysteme. In der Praxis bedeutet das, dass selbst Daten, die eigentlich geschützt sein sollten, oft ohne Verschlüsselung abgelegt werden und damit anfällig für unbefugten Zugriff sind.

Ein solcher Ansatz ist besonders wichtig in Szenarien, in denen Computer gemeinsam genutzt werden, wie etwa in Familienhaushalten, an öffentlichen Arbeitsplätzen oder bei geteilten PCs.

Durch die Entwicklung einer Lösung, die nahtlos in das Linux-Dateisystem integriert wird, soll sichergestellt werden, dass die verschlüsselten Verzeichnisse für den Anwender wie reguläre Ordner aussehen und sich genauso einfach nutzen lassen. So können auch Menschen ohne spezielles

Fachwissen ihre Daten schützen. Diese Arbeit bietet damit einen wichtigen Beitrag zu mehr digitaler Sicherheit und Datenschutz für eine breite Nutzerbasis.

## 1.2. Ziel der Arbeit

Das Ziel dieser Arbeit ist die Entwicklung einer Linux-basierten Lösung zur transparenten Verschlüsselung von Verzeichnissen im Dateisystem, sodass ein technisch unbedarfter Benutzer keinen Unterschied zwischen einem regulären und einem verschlüsselten Verzeichnis bemerkt. Die Lösung soll vollständig in das Dateisystem integriert werden und eine transparente Nutzung der verschlüsselten Verzeichnisse ermöglichen.

Zur Sicherstellung der Datensicherheit sollen die Inhalte der Verzeichnisse mithilfe eines modernen und robusten Authenticated Encryption (AE)-Schemas, wie Pipelineable On-line Encryption with authentication Tag (POET), verschlüsselt werden. Der für die Verschlüsselung erforderliche geheime Schlüssel wird aus dem Benutzer-Login-Passwort durch eine sichere und zeitgemäße Key-Derivation-Funktion (KDF) wie Catena abgeleitet.

Neben der Entwicklung der Verschlüsselungslösung ist vorgesehen, den vollständigen Quellcode auf GitHub bereitzustellen. Darüber hinaus soll das Projekt als .deb-Paket veröffentlicht werden, um eine einfache Installation und Nutzung unter Linux zu ermöglichen.

## 1.3. Aufbau der Arbeit

1. Die erste Komponente, Filysystem in Userspace (FUSE), ermöglicht die Implementierung eines Dateisystems im Userspace. FUSE dient hier dazu, die Dateisystemoperationen abzufangen und zu verwalten, wodurch eine transparente Verschlüsselung gewährleistet werden kann. Das bedeutet, dass alle Lese- und Schreibvorgänge auf dem Dateisystem verschlüsselt und entschlüsselt werden, ohne dass der Benutzer manuell eingreifen muss. Die Integration von POET und Catena in das FUSE-basierte Dateisystem bildet die Grundlage für die sichere und automatische Verschlüsselung aller Dateien.
2. Die zweite Komponente, POET, wird für die Verschlüsselung und Authentifizierung der Daten im Dateisystem verwendet. POET ist ein Authenticated Encryption (AE) Schema, das sowohl die Vertraulichkeit als auch die Integrität der Daten sicherstellt, was besonders wichtig für die Datenintegrität und den Schutz vor Manipulationen ist.
3. Die dritte Komponente, Catena, wird für die Schlüsselableitung eingesetzt. Catena bietet eine sichere Möglichkeit, kryptografische Schlüssel aus Passwörtern abzuleiten und schützt dabei vor Brute-Force-Angriffen und anderen Angriffen auf Passwörter. In diesem Projekt wird Catena verwendet, um sichere und einzigartige Schlüssel für die POET-Verschlüsselung zu generieren, wodurch ein hohes Maß an Sicherheit gewährleistet wird.

Zusammen ermöglichen diese drei Komponenten die Entwicklung eines robusten Dateisystems, das Daten zuverlässig verschlüsselt, Authentifizierung bereitstellt und sichere Schlüsselableitungen aus Passwörtern gewährleistet.

In Kapitel 2 werden die Grundlagen vorgestellt und in Kapitel 3 wird das Konzept der Anwendung präsentiert. Wie diese umgesetzt wurde wird in Kapitel 4 erläutert. In Kapitel 5 werden die Ergebnisse vorgestellt und im letzten Kapitel ist eine Zusammenfassung der Arbeit 6.

## 2. Grundlagen

### 2.1. FUSE

#### 2.1.1. Einführung in FUSE

FUSE bietet eine innovative und flexible Plattform für die Entwicklung von Dateisystemen, die vollständig im Benutzermodus läuft, anstatt wie herkömmliche Dateisysteme direkt im Kernel. FUSE wurde ursprünglich entwickelt, um Entwicklern eine sichere und einfache Möglichkeit zu geben, eigene Dateisysteme zu erstellen, ohne dabei in die Kernel-Ebene eingreifen zu müssen. Die Struktur ermöglicht es, Dateisysteme in einer Vielzahl von Programmiersprachen zu entwickeln und erleichtert gleichzeitig das Debugging und die Weiterentwicklung, da der gesamte Code in einer isolierten Userspace-Umgebung ausgeführt wird. Ein entscheidender Vorteil von FUSE ist die erhöhte Sicherheit, da Fehler im Userspace nicht den Kernel beeinträchtigen und somit das Risiko eines Systemabsturzes minimiert wird. Durch seine Flexibilität und die breite Unterstützung in Linux- und Unix-Umgebungen ist FUSE heute eine zentrale Technologie für eine Vielzahl von Anwendungen – von Cloud-Speichern bis hin zu verschlüsselten oder verteilten Dateisystemen. Diese Möglichkeit reduziert die Komplexität und das Risiko erheblich, da alle Dateisystemoperationen im Userspace ausgeführt werden. Für Entwickler bedeutet dies eine größere Kontrolle und Flexibilität bei der Umsetzung von Dateisystemfunktionen, die über die traditionellen Optionen hinausgehen [30, S.1-9] [26, S.1-2] .

Ein weiterer Vorteil von FUSE ist die breite Palette an unterstützten Programmiersprachen. Während traditionelle Kernel-basierte Dateisysteme zumeist in C geschrieben sind, erlaubt FUSE die Nutzung von Sprachen wie Python, Java, oder auch Go. Das eröffnet zusätzliche Gestaltungsmöglichkeiten, da viele Entwickler von den erweiterten Bibliotheken und Frameworks profitieren können, die in diesen Sprachen verfügbar sind. Zudem bietet die Userspace-Umgebung eine erhöhte Effizienz bei der Entwicklung und Wartung, da Änderungen sofort getestet und umgesetzt werden können, ohne das Risiko eines Systemabsturzes[20, 16, 24].

Durch diese Architektur können Entwickler flexibel und sicher Dateisysteme erstellen, die im Userspace laufen, ohne tiefgehende Kernel-Programmierung. Das erleichtert die Entwicklung sowie das Testen und die Wartung von Dateisystemen erheblich.

Insgesamt erweist sich FUSE somit als leistungsfähige und vielseitige Plattform, die es Entwicklern erlaubt, spezielle und komplexe Dateisystemlösungen umzusetzen, wie sie etwa für

Verschlüsselung, verteilte Speicherlösungen oder sogar Virtualisierungsumgebungen erforderlich sind[26, S.2-5].

### 2.1.2. Funktionsweise und Architektur von FUSE

Die Architektur von FUSE besteht aus einem Kernel-Modul, einem Userspace-Daemon und einer API. Das Kernel-Modul agiert als Vermittler zwischen dem Betriebssystem und dem benutzerdefinierten Dateisystem, indem es Systemaufrufe abfängt und an den Userspace-Daemon weiterleitet. Dieser Daemon verarbeitet die Anfragen und führt die eigentliche Logik des Dateisystems aus. Die Kommunikation zwischen Kernel und Userspace erfolgt über eine spezielle API, die es ermöglicht, Dateisystemoperationen wie Lesen, Schreiben oder Löschen im Userspace zu handhaben. Diese Trennung bietet Entwicklern Flexibilität und Sicherheit[15, S.4].

**Das Kernel-Modul** ist dafür verantwortlich, Systemaufrufe wie Dateioperationen abzufangen und diese Anfragen an den Userspace-Daemon weiterzuleiten. Es agiert somit als Bindeglied zwischen den Anwendungen und dem benutzerdefinierten Dateisystem und stellt sicher, dass der Benutzer wie bei jedem nativen Dateisystem darauf zugreifen kann.

**Der Userspace-Daemon** hingegen ist der Ort, an dem die eigentliche Dateisystemlogik implementiert wird. Hier wird definiert, wie Anfragen verarbeitet werden, welche Dateistrukturen verwendet werden und wie die Daten abgelegt oder verwaltet werden. Der Daemon empfängt die vom Kernel-Modul weitergeleiteten Anfragen, verarbeitet diese gemäß der im Userspace definierten Logik und gibt eine Antwort an das Kernel-Modul zurück, dass die Daten dann an die Anwendung übermittelt. Die Trennung zwischen Kernel und Userspace erleichtert die Entwicklung von Dateisystemen erheblich, da sich Entwickler auf die eigentliche Logik konzentrieren können, ohne tiefgreifende Eingriffe in dem Kernel vorzunehmen[31, S.2-3].

Wie bereits erwähnt, umfasst die Architektur von FUSE nicht nur das Kernel-Modul und den Userspace-Daemon, sondern stellt auch zwei unterschiedliche API-Ebenen zur Verfügung, die sogenannte High-Level-API und Low-Level-API. Diese APIs bieten unterschiedliche Abstraktionsstufen für die Implementierung eines benutzerdefinierten Dateisystems und ermöglichen es Entwicklern, entweder auf einfache Weise grundlegende Dateisystemoperationen zu implementieren oder tiefere Kontrolle über Dateizugriffe zu erhalten.

Die High-Level-API von FUSE ist darauf ausgelegt, die Entwicklung eines Dateisystems so einfach wie möglich zu gestalten. Sie bietet abstrahierte Funktionen für typische Dateisystemoperationen wie open, read, write und close. Diese API-Ebene ist besonders für Entwickler geeignet, die ein Dateisystem mit den gängigen Standardoperationen umsetzen möchten, ohne dabei eine detaillierte Kontrolle über die interne Kommunikation und Verwaltung von Inodes und Dateistrukturen zu benötigen. Die High-Level-API von FUSE ist eine hervorragende Wahl für Projekte, bei denen die Komplexität des Dateisystems gering gehalten werden soll, etwa bei der Implementierung von benutzerdefinierten Cloud-Speicherlösungen oder verschlüsselten

Verzeichnissen.

Im Gegensatz dazu bietet die Low-Level-API eine wesentlich feinere Kontrolle über die Operationen auf Inode-Ebene, wodurch Entwickler direkt mit Inodes, Dentries und den nativen VFS-Strukturen (Virtual File System) arbeiten können. Die API ist komplexer, da der Entwickler für die Verwaltung von Inodes und die Verknüpfung von Dateisystemknoten verantwortlich ist. Der Vorteil dieser API liegt jedoch in der Möglichkeit, die Datenverwaltung und -verarbeitung präzise anzupassen und zu optimieren. Die Low-Level-API eignet sich daher besonders für leistungsintensive oder sehr spezifische Anwendungen, bei denen eine feingranulare Kontrolle über die Dateistrukturen und den Dateizugriff erforderlich ist, etwa bei verteilten Speichersystemen oder datenbankgestützten Dateisystemen, die eine hohe Performance und Effizienz erfordern[30, S.8-10].

Zusammen bieten die High-Level- und Low-Level-API Entwicklern, die nötige Flexibilität, ein breites Spektrum an Anwendungsfällen zu bedienen. Die Trennung der API-Schichten erlaubt es, den Entwicklungsaufwand und die Komplexität eines FUSE-basierten Dateisystems je nach Projektanforderungen anzupassen, was FUSE zu einem vielseitigen und leistungsstarken Werkzeug für die Implementierung benutzerdefinierter Dateisystemlösungen macht. Die nächste Abbildung verdeutlicht nochmals, wie die einzelnen FUSE-Komponenten miteinander interagieren und wie die Architektur von FUSE aufgebaut ist [30, S.1-12].

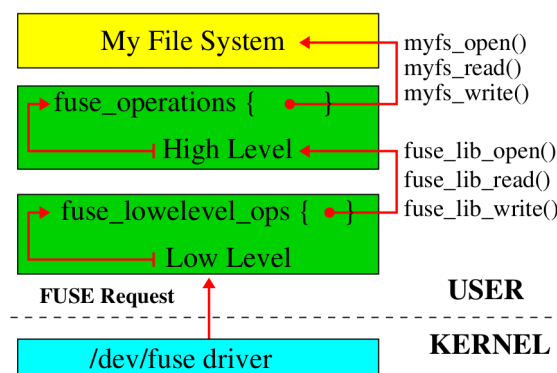


Abbildung 2.1.: FUSE-Architektur [30, S.9]

### 2.1.3. Anwendungsfälle und Vorteile von FUSE

FUSE wird in einer Vielzahl von Anwendungsfällen eingesetzt, da es Entwicklern erlaubt, maßgeschneiderte Dateisystemlösungen ohne direkte Kernel-Modifikationen zu implementieren. Ein bedeutender Einsatzbereich liegt in verteilten Dateisystemen, bei denen Daten über mehrere Systeme hinweg gespeichert und synchronisiert werden. Beispiel hierfür ist ein verteiltes Dateisystem wie GlusterFS, die FUSE nutzt, um eine flexible und erweiterbare Infrastruktur für skalierbare Speichersysteme bereitzustellen [9].

Auch bei verschlüsselten Dateisystemen ist FUSE eine beliebte Wahl. Hier bietet FUSE den Vorteil, Verschlüsselungslogiken vollständig im Userspace zu verarbeiten, was die Sicherheit erhöht, da Fehler im Benutzermodus die Stabilität des Betriebssystems nicht direkt beeinträchtigen. Dateisysteme wie EncFS nutzen FUSE, um Verschlüsselungsschichten einzuführen, die die Daten für den Benutzer transparent ver- und entschlüsseln, während der Benutzer nahtlos auf die Daten zugreifen kann[10].

Ein weiterer Anwendungsfall für FUSE liegt in der Implementierung von Netzwerkspeicherlösungen. Durch FUSE können entfernte Speicherorte wie lokale Ordner gemountet werden, was die Entwicklung von Netzwerkspeichern vereinfacht. SSHFS ist ein Beispiel für eine solche Anwendung, bei der entfernte Dateisysteme über das Netzwerk über das SSH-Protokoll zugänglich gemacht werden, was die Verwaltung und den Zugriff auf entfernte Dateien effizient und sicher gestaltet[26, S.1][4, 5].

Die Hauptvorteile von FUSE, wie die Isolation im Userspace und die hohe Flexibilität, bieten Entwicklern erhebliche Freiheiten bei der Implementierung individueller Funktionen. Durch die Ausführung im Userspace wird das Risiko eines Systemabsturzes minimiert. Dies ist besonders vorteilhaft für experimentelle und sicherheitsrelevante Dateisysteme, die Verschlüsselungslösungen einführen. Die Flexibilität, die FUSE bietet, ermöglicht Entwicklern außerdem die Nutzung verschiedenster Programmiersprachen und Bibliotheken, was FUSE besonders für spezielle Anwendungen wie verschlüsselte oder netzwerkbasierte Dateisysteme geeignet macht[30, S.4-6].

#### **2.1.4. Verschlüsselung mit FUSE**

FUSE bietet eine leistungsstarke Grundlage für die Implementierung transparenter Verschlüsselungslösungen in Dateisystemen. Durch seine Architektur, die es ermöglicht, Dateisystemlogik im Userspace zu entwickeln, wird die Realisierung einer benutzerdefinierten Verschlüsselungsschicht erheblich vereinfacht. Transparente Verschlüsselung bedeutet, dass Daten beim Speichern automatisch verschlüsselt und beim Lesen automatisch entschlüsselt werden, ohne dass der Benutzer aktiv auf die Verschlüsselungsprozesse zugreifen muss. Der Zugriff auf Dateien bleibt so fließend und intuitiv wie bei einem normalen Dateisystem, wobei die Sicherheit der Daten jedoch erheblich gesteigert wird.

Gerade für Entwickler und Organisationen, die die Sicherheit von sensiblen Daten gewährleisten müssen, bietet FUSE damit die Möglichkeit, benutzerdefinierte Verschlüsselungslösungen anzuwenden, die spezifische Sicherheitsanforderungen erfüllen sollten. Des Weiteren bietet FUSE eine Ausführung in Echtzeit sowie eine leichte Integration in bestehende IT-Infrastrukturen, was die Anwendbarkeit von FUSE-basierten Dateisystemen in Bereichen wie Cloud-Speicherung, vertraulichen Datenbanken und personalisierten Backup-Systemen stark erhöht[23, S. 6-10].

Das entwickelte FUSE-basierte Dateisystem orientiert sich stark an der Vorlage „Writing a FUSE Filesystem“ vom "Department of Computer Science New Mexico State University"[22],



einer beliebten und ausführlichen Einführung in die Erstellung von FUSE-Dateisystemen. Diese Vorlage bietet eine solide Grundlage, um die grundlegende Architektur und Funktionsweise von FUSE zu verstehen, einschließlich der Implementierung von Funktionen wie Lesen, Schreiben und Mounten. Sie dient nicht nur als wertvolles Lernmaterial, sondern ist auch eine Referenz für Entwickler, die eigene Dateisysteme implementieren möchten.

## **2.2. Verschlüsselung**

Das FUSE-Framework bietet eine flexible Möglichkeit, Dateisysteme im Userspace zu entwickeln und stellt sicher, dass Daten unabhängig vom zugrunde liegenden Dateisystem verarbeitet werden können. Um die Vertraulichkeit und Integrität der Daten in diesem FUSE-basierten System zu gewährleisten, wird eine transparente Verschlüsselung implementiert, bei der die Sicherheit der Daten ohne direkten Eingriff der Benutzer sichergestellt wird. Um die Vertraulichkeit der Daten sicherzustellen, wird AES eingesetzt, um die Daten zu verschlüsseln.

### **2.2.1. Advanced Encryption Standard (AES)**

AES wurde entwickelt, um die Sicherheitsstandards in der Datenverschlüsselung zu erhöhen und war das Ergebnis eines umfangreichen Auswahlprozesses der US-Regierung in den späten 1990er Jahren. AES ist mittlerweile der weltweit anerkannte Standard für symmetrische Verschlüsselung und löste den veralteten DES-Algorithmus (Data Encryption Standard) ab. Es bietet durch seine Blockverschlüsselungsstruktur und flexiblen Schlüsselgrößen von 128, 192 und 256 Bit ein hohes Maß an Sicherheit.

AES ist eine symmetrische Blockchiffre, die Daten in Blöcken fester Größe von 128 Bit verarbeitet, Abbildung 2.2 zeigt den Aufbau einer AES Blockchiffre[13, S.238-239]. Wenn die zu verschlüsselnden Daten nicht exakt dieser Blockgröße entsprechen, wird ein Verfahren namens Padding angewendet, um die Daten auf die erforderliche Länge zu bringen[21, S.148].

#### **Grundlegende Merkmale einer Blockchiffre bei AES:**

- **Feste Blockgröße:** Daten werden in gleich großen Blöcken verarbeitet. Hierdurch wird eine systematische Transformation des Klartexts in Chiffretext und eine einfachere Verwaltung der Verschlüsselung ermöglicht[21, S.113-114].
- **Schlüsselabhängige Transformation:** Bei jeder Verschlüsselung wird ein geheimer Schlüssel verwendet, um den Klartext so zu transformieren, dass er für Dritte unlesbar wird. Diese Transformation ist stark von der jeweiligen Schlüsselauswahl abhängig, was die Sicherheit der verschlüsselten Daten gewährleistet[21, S.130-131].
- **Rundenbasierte Verarbeitung:** Eine Blockchiffre wie AES verschlüsselt den Klartext in

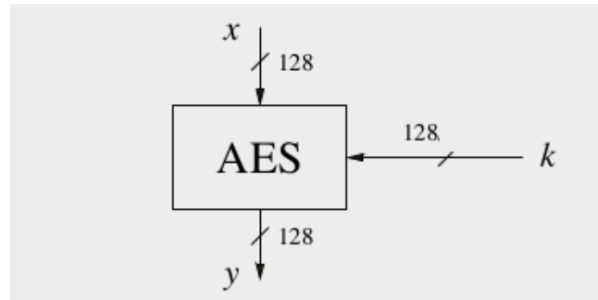


Abbildung 2.2.: AES-Blockchiffre[21, S.113]

mehreren Runden, die jeweils aus bestimmten Operationen wie Substitutionen und Permutationen bestehen. Diese mehrstufige Verschlüsselung sorgt dafür, dass die Daten vollständig verschleiert werden und für Angreifer schwer zu entschlüsseln sind[21, S.114-116].

**AES basiert auf einer Substitution-Permutation-Netzwerk (SPN) Struktur**, die entwickelt wurde, um Daten auf sichere und effiziente Weise zu transformieren. In einem SPN werden die Daten durch eine Folge von Substitutions- und Permutationsoperationen verarbeitet, die durch mehrere Runden von Rechenschritten ausgeführt werden. Diese Struktur sorgt für eine starke Verschlüsselung, indem sie sowohl Confusion als auch Diffusion der Daten erreicht[13, S.239-240].

**AES basiert auf vier Rechenoperationen und ihre Funktion im SPN-Netzwerk.** Die SubBytes-Operation in AES ist eine zentrale Schicht in jeder Runde und dient dazu, jedes Byte in der Datenmatrix durch ein neues Byte zu ersetzen, das von einer speziellen Substitutionsbox, der S-Box, berechnet wird. Die S-Box ist so konzipiert, dass sie Verwirrung (Confusion) erzeugt, was bedeutet, dass der Zusammenhang zwischen dem ursprünglichen Klartext und dem resultierenden Chiffretext verschleiert wird. Diese nichtlineare Transformation stellt sicher, dass selbst kleine Änderungen im Klartext signifikante Unterschiede im Chiffretext verursachen und erhöht so die Widerstandsfähigkeit von AES gegen bestimmte Arten von Angriffen.

Die nichtlineare Natur der S-Box ist entscheidend für die Sicherheit des Verfahrens, da sie sicherstellt, dass AES nicht anfällig für Angriffe ist, die auf linearen Beziehungen im Datenfluss basieren. Da die S-Box eine bijektive Abbildung ist, hat jeder der 256 möglichen Eingabewerte (für ein Byte) genau einen eindeutigen Ausgabewert.

Die Bijektivität erlaubt es, die Substitution im Entschlüsselungsprozess rückgängig zu machen – eine Eigenschaft, die für die Wiederherstellung des Klartextes entscheidend ist. In softwarebasierten Implementierungen wird die S-Box oft als Tabelle mit 256 Einträgen gespeichert, sodass jeder Bytewert direkt nachgeschlagen und in den entsprechenden Ausgabewert umgewandelt werden kann[21, S.125-126][1, S.97-98].

Beispiel für die Byte-Substitution in AES:

Nehmen wir an, das Eingangsbyte A hat den hexadezimalen Wert  $BC_{16}$  was in Binärerform 1011 1100 entspricht. Für die Substitution wird das Byte in zwei 4-Bit-Hälften aufgeteilt:

- Die ersten 4 Bits (1011) repräsentieren den X-Wert für die S-Box.
- Die letzten 4 Bits (1100) repräsentieren den Y-Wert für die S-Box.

Die S-Box verwendet diese X- und Y-Koordinaten, um den neuen Bytewert zu bestimmen. Jede Kombination von X und Y führt zu einem eindeutigen Wert in der S-Box-Tabelle, der das Substitutionsbyte für diesen spezifischen Eingabewert darstellt[27, S.174-176]. Der Eingangsbyte-Wert  $BC_{16}$  ist dann auf  $65_{16}$  abgebildet. Wie in der Abbildung von der S-BOX von AES zu sehen ist.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Tabelle 2.1.: AES-S-Box: Substitutionswerte in hexadezimaler Notation für die Eingabebytes (xy)

Bevor ein Klartext-Byte überhaupt in die Byte-Substitution-Schicht eintritt, wird es im AES-Algorithmus mit dem Schlüssel durch eine XOR-Operation verknüpft. Diese Anfangs-XOR-Operation sorgt dafür, dass der ursprüngliche Klartextwert modifiziert wird, bevor er durch die S-Box verarbeitet wird. Dadurch wird die Unvorhersehbarkeit des Ergebnisses erhöht, da der Schlüsselwert jedes Byte beeinflusst und somit die Wirkung der S-Box zufällig erscheinen lässt.

Da die S-Box die einzige nichtlineare Komponente im AES-Algorithmus ist, verstärkt diese Transformation den Schutz.

In der Diffusionsschicht des AES-Algorithmus wird eine Transformation angewendet, die sicherstellt, dass Änderungen an einem einzigen Bit möglichst viele andere Bits beeinflussen. Die Diffusion im AES-Algorithmus erfolgt in zwei Schritten: der ShiftRows-Transformation und der MixColumns-Transformation. Im Gegensatz zur S-Box sind diese Operationen linear, was bedeutet, dass die Addition zweier transformierter Zustände dem transformierten Ergebnis der

beiden Zustände entspricht[21, S.114].

Die ShiftRows-Transformation verschiebt die Bytes in jeder Zeile der Zustandsmatrix zyklisch, um sicherzustellen, dass Byte-Werte weiter über den gesamten Zustand verteilt werden. Hieraus ergeben sich folgende Zustände:

- Die erste Zeile bleibt unverändert.
- Die zweite Zeile wird um ein Byte nach links verschoben.
- Die dritte Zeile wird um zwei Bytes nach links verschoben.
- Die vierte Zeile wird um drei Bytes nach links verschoben.

Durch diese Verschiebungen wird die Diffusion erhöht, da die Positionen der Bytes in der Zustandsmatrix verändert werden, was dazu führt, dass Byte-Werte aus einer einzigen Spalte sich über die gesamte Matrix verteilen[27, S.179].

Die folgenden Abbildungen veranschaulichen den Originalzustand der Matrix, sowie den Zustand nach der ersten Verschiebung.

**Vor dem Verschieben (Original)**

$B_0$	$B_4$	$B_8$	$B_{12}$
$B_1$	$B_5$	$B_9$	$B_{13}$
$B_2$	$B_6$	$B_{10}$	$B_{14}$
$B_3$	$B_7$	$B_{11}$	$B_{15}$

Abbildung 2.3.: Struktur vor dem Verschieben der Blöcke, Runde null

**Nach dem Verschieben**

$B_0$	$B_4$	$B_8$	$B_{12}$
$B_5$	$B_9$	$B_{13}$	$B_1$
$B_{10}$	$B_{14}$	$B_2$	$B_6$
$B_{15}$	$B_3$	$B_7$	$B_{11}$

Abbildung 2.4.: Struktur nach dem Verschieben der Blöcke, Runde eins

## MixColumns-Transformation

Die MixColumns ist eine lineare Transformation, die auf jeder Spalte der Zustandsmatrix angewendet wird. In diesem Schritt wird jede Spalte als Vektor von vier Bytes behandelt und mit einer festen  $4 \times 4$ -Matrix multipliziert. Diese Matrix besteht aus konstanten Werten, die im Galois-Feld  $GF(2^8)$  berechnet werden. Die MixColumns-Transformation sorgt dafür, dass jedes Byte einer Spalte die anderen drei Bytes der Spalte beeinflusst.

Die Diffusion wird durch MixColumns weiter verstärkt, da Byte-Werte innerhalb einer Spalte stark miteinander verwoben werden. Nach nur drei Runden sind durch die Kombination von ShiftRows und MixColumns alle 16 Bytes der Zustandsmatrix so verteilt, dass eine vollständige Defusion erreicht wird.

## Berechnung der MixColumns-Transformation

Angenommen, die vier Bytes einer Spalte nach der ShiftRows-Transformation sind  $B_0, B_5, B_{10}, B_{15}$  werden diese mit der festen Matrix multipliziert:

Hier werden die Werte  $B_0, B_5, B_{10}, B_{15}$  durch Multiplikationen und Additionen mit den Matrixkonstanten kombiniert, um den Ausgabewert  $C$  zu berechnen. Für die Koeffizienten werden

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix}$$

Abbildung 2.5.: MixColumns Konstante

Werte wie 01, 02 und 03 verwendet, die die einfache Multiplikationen ermöglichen.

In der Galois-Feld-Arithmetik  $GF(2^8)$ , die AES verwendet, werden Multiplikationen und Additionen auf besondere Weise durchgeführt, um die Berechnungen effizient zu halten.

Eine Multiplikation mit zwei stellt eine links Verschiebung um ein Bit dar. Das bedeutet, dass alle Bits eine Position nach links rücken, wodurch sich der Wert verdoppelt (im Sinne der Binärarithmetik).

Wohingegen eine Multiplikation mit drei eine Verschiebung nach links wie bei der Multiplikation mit zwei und eine Addition des ursprünglichen Wertes kombiniert. In der Praxis wird dies durch eine XOR-Verknüpfung zwischen dem Ergebnis der Verschiebung und dem Originalwert erreicht [27, S.180-183].

### AddRoundKey

In diesem Schritt wird der jeweilige Runden-Schlüssel (eine Ableitung des ursprünglichen Schlüssels) durch eine XOR-Operation mit der Datenmatrix kombiniert. Diese Operation hängt direkt vom geheimen Schlüssel ab, sodass die Sicherheit des gesamten Prozesses maßgeblich von der Geheimhaltung des Schlüssels abhängt. AddRoundKey ist der Schritt, der die Verknüpfung zwischen den Klartextdaten und dem Schlüssel herstellt, was die Confusion verstärkt und verhindert, dass der Schlüssel leicht erraten werden kann.

Der Schlüsselfahrplan von AES ist ein Prozess, der den ursprünglichen Schlüssel in mehrere Rundenschlüssel umwandelt. Diese Rundenschlüssel werden für jede Runde des AES-Algorithmus benötigt. Sie sorgen dafür, dass jede Runde eine unterschiedliche Verschlüsselung durchführt [27, S.183-184].

### Funktionsweise des Schlüsselfahrplan

Je nach Schlüssellänge (128, 192 oder 256 Bit) werden unterschiedliche Anzahlen von Unterschlüsseln benötigt:

- **128-Bit-Schlüssel:** 10 Runden  $\rightarrow$  11 Rundenschlüssel
- **192-Bit-Schlüssel:** 12 Runden  $\rightarrow$  13 Rundenschlüssel
- **256-Bit-Schlüssel:** 14 Runden  $\rightarrow$  15 Rundenschlüssel

Diese zusätzlichen Rundenschlüssel werden durch eine Technik namens *Key Whitening* erzeugt, die sicherstellt, dass der Schlüssel am Anfang und am Ende des Prozesses die Daten beeinflusst[21, S.131].

### Beispiel für den Schlüsselfahrplan bei einem 128-Bit-Schlüssel

Nehmen wir als Beispiel einen 128-Bit-Schlüssel. Der Originalschlüssel wird zunächst in vier 32-Bit-Wörter unterteilt, die den ersten Rundenschlüssel bilden. In AES wird die Berechnung der weiteren Unterschlüssel in Einheiten von 32-Bit-Wörtern durchgeführt und jeder Unterschlüssel besteht aus vier Wörtern.

#### Erster Rudenschlüssel

Der ursprüngliche Schlüssel selbst bildet den ersten Unterschlüssel.

#### Berechnung der weiteren Rudenschlüssel

Um einen neuen Rudenschlüssel zu berechnen, nehmen wir das letzte 32-Bit-Wort des vorherigen Unterschüssels und verarbeiten es durch eine Funktion, die als *g-Funktion* bezeichnet wird. Die *g-Funktion* besteht aus drei Schritten:

1. **Rotation:** Die Bytes des Wortes werden zyklisch um eine Position nach links verschoben, wenn das Wort zum Beispiel  $W = [A, B, C, D]$  ist, wird es zu  $[B, C, D, A]$ .
2. **Substitution durch die S-Box:** Jedes Byte wird durch die AES-S-Box geschickt, um es zu ersetzen. Dies sorgt für die notwendige Nichtlinearität.
3. **Addition eines Rundenkoeffizienten (RC):** Ein fester Wert (abhängig von der Runde) wird dem ersten Byte des Wortes hinzugefügt. Dieser Rundenkoeffizient sorgt dafür, dass jeder Rundenschlüssel unterschiedlich ist, auch wenn der ursprüngliche Schlüssel dieselbe Struktur aufweist.

Das Ergebnis der *g-Funktion* wird dann mit dem ersten Wort des vorherigen Unterschüssels XOR-verknüpft, um das erste Wort des neuen Unterschüssels zu erzeugen. Dieser Prozess wird rekursiv fortgesetzt, um alle vier Wörter des neuen Unterschüssels zu berechnen[27, S.183-187].

Die Anzahl der Rundenschlüssel in AES ist um eins höher als die Anzahl der Runden, weil es am Anfang des Algorithmus einen zusätzlichen Schlüsselverknüpfungsschritt gibt. Dieser zusätzliche Unterschlüssel dient dazu, die Sicherheit zu erhöhen und wird in einem Prozess namens *Key Whitening* verwendet. Die nächste Abbildung zeigt den schlüsselnfahrplan in der ersten Runde.

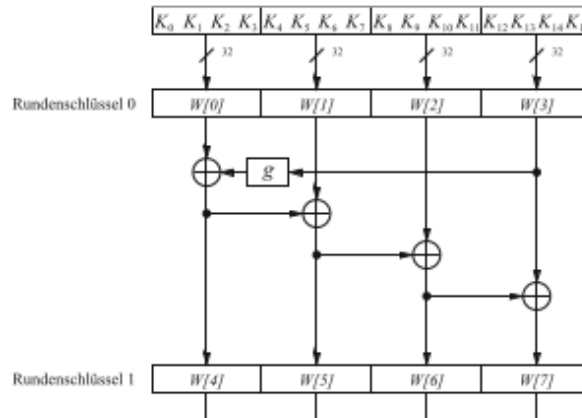


Abbildung 2.6.: Schlüsselfahrplan für AES in der ersten Runde[21, S.125]

## Warum sind 11 Rundenschlüssel erforderlich, wenn es nur 10 Runden gibt?

### 1. Initiale Key-Addition (AddRoundKey) vor der ersten Runde:

Bevor die erste Runde von AES beginnt, wird der Klartext (die Eingabedaten) zunächst mit dem ersten Rundenschlüssel XOR-verknüpft. Dieser erste Rundenschlüssel ist der sogenannte Initialschlüssel oder  $k_0$ . Diese Operation wird als Key Whitening bezeichnet und sorgt dafür, dass der Klartext vor den Runden schon einmal mit dem Schlüssel vermischt wird.

### 2. 10 Runden in AES-128:

Jede der folgenden 10 Runden benötigt ihren eigenen Rundenschlüssel, um den Zustand der Daten in jeder Runde zu modifizieren.

- Ein Rundenschlüssel wird vor der ersten Runde hinzugefügt (Initialschlüssel). 10 weitere Rundenschlüssel werden in den 10 Runden des AES-128 verwendet. Deshalb benötigt AES-128 11 Rundenschlüssel, obwohl es nur 10 Runden gibt: Einen für die initiale XOR-Verknüpfung vor der ersten Runde und einen für jede der folgenden 10 Runden. Dieser zusätzliche Rundenschlüssel sorgt für eine zusätzliche Sicherheitsschicht, da die Daten mit dem Schlüssel bereits vor Beginn der eigentlichen Runden "verwässert" werden[21, S131-132].

## Bedeutung von Confusion und Diffusion in AES

In AES dienen die Operationen SubBytes und AddRoundKey zur Confusion, um die Beziehung zwischen Klartext und Chiffretext möglichst unklar zu machen. Dies erschwert es Angreifern, eine Korrelation zwischen dem ursprünglichen und dem verschlüsselten Text zu erkennen. Die Operationen ShiftRows und MixColumns hingegen sorgen für Diffusion, indem sie die Byte-Werte in der Datenmatrix neu anordnen und verteilen. So wird sichergestellt, dass eine kleine Änderung im Klartext den gesamten Chiffretext beeinflusst.

Durch diese Kombination aus Confusion und Diffusion, erreicht durch die vier Operationen und das SPN-Netzwerk, ist AES in der Lage, einen sehr hohen Sicherheitsstandard zu bieten. Die wiederholte Anwendung dieser vier Schritte in jeder Runde sorgt dafür, dass der AES-Algorithmus extrem schwer zu durchbrechen ist.

In AES wird die Sicherheit durch die wiederholte Anwendung der vier grundlegenden Rechenoperationen in mehreren Runden erreicht.

Jede Runde verstärkt die Confusion und Diffusion des Klartexts und sorgt dafür, dass der resultierende Chiffretext für Unbefugte äußerst schwer zu entschlüsseln ist[21, S.114-115].

In jeder dieser Runden werden die vier Rechenoperationen SubBytes, ShiftRows, MixColumns und AddRoundKey nacheinander ausgeführt. Die Abfolge wird bis zur letzten Runde beibehalten, mit einer Ausnahme, in der letzten Runde wird auf die MixColumns-Operation verzichtet. Diese Anpassung sorgt dafür, dass der finale Chiffretext ohne zusätzliche Diffusion im letzten Schritt erzeugt wird, was die Entschlüsselung der letzten Verschlüsselungsrunde erleichtert, ohne die Sicherheit zu beeinträchtigen[13, S.239].

### **2.2.2. Betriebsmodi von AES**

Bei der Verschlüsselung von Daten, die größer als diese Blockgröße sind, wird der Klartext in entsprechende Blöcke aufgeteilt, die dann einzeln verschlüsselt werden. Die Art und Weise, wie diese Blöcke verarbeitet werden, wird durch sogenannte Betriebsmodi definiert.

Ein grundlegender Betriebsmodus ist der Electronic Codebook (ECB)-Modus. Hierbei wird jeder Klartextblock unabhängig mit demselben Schlüssel verschlüsselt. Das führt dazu, dass identische Klartextblöcke in identische Chiffretextblöcke umgewandelt werden. Bei der Verschlüsselung von Bildern kann das problematisch sein, wenn bestimmte Muster oder Strukturen im Bild mehrfach vorkommen, bleiben diese im verschlüsselten Bild erkennbar, da die entsprechenden Chiffretextblöcke ebenfalls identisch sind. Das beeinträchtigt die Sicherheit, da Informationen über den ursprünglichen Inhalt preisgegeben werden können[14, S.14-16].



Die folgende Abbildung verdeutlicht nochmal die Schwächen, der reinen AES-Verschlüsselung(ECB):

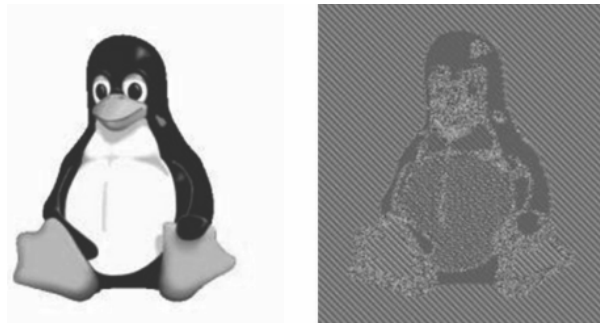


Abbildung 2.7.: ECB-Mode[13, S.90]

Um die Schwächen des ECB-Modus zu vermeiden, der für identische Klartextblöcke stets dieselben Chiffretextblöcke erzeugt, wurden alternative Betriebsmodi entwickelt. Diese Betriebsmodi verbessern die Sicherheit, indem sie sicherstellen, dass gleiche Klartextblöcke zu unterschiedlichen Chiffretextblöcken führen, wodurch die Musterbildung effektiv verhindert wird. Ein Beispiel hierfür ist der Cipher Block Chaining (CBC)-Modus, bei dem eine Verkettung der Klartextblöcke mithilfe eines Initialisierungsvektors (IV) oder auch in manchen Lektüren wird es Nonce (Number used once) genannt, erfolgt. Der IV sorgt dafür, dass der erste Block zufällig verändert wird und jeder nachfolgende Block mit dem vorherigen Chiffretextblock kombiniert wird. Dadurch bleibt die Verschlüsselung auch bei wiederholten Klartextblöcken einzigartig und sicherer gegenüber statistischen Angriffen[13, S.90-91].

Die Wahl des geeigneten Betriebsmodus ist entscheidend für die Sicherheit der Verschlüsselung und sollte sorgfältig an die spezifischen Anforderungen der Anwendung angepasst werden. Die folgende Abbildung zeigt den Aufbau vom CBC-mode

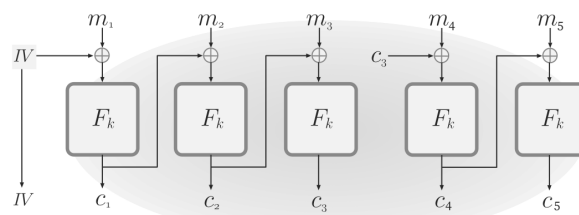


Abbildung 2.8.: CBC-Mode[13, S.91]

Sowohl der Initialisierungsvektor (IV) als auch der Nonce erfüllen im Wesentlichen denselben Zweck. Sie sorgen dafür, dass keine Muster im verschlüsselten Text erkennbar sind, selbst wenn identische Klartextdaten mehrfach verschlüsselt werden. Dadurch wird verhindert, dass Angreifer Rückschlüsse auf den Inhalt oder die Struktur der verschlüsselten Daten ziehen können[27, S.214].

### 2.2.3. Einführung in Authenticated Encryption (AE)

Authenticated Encryption (AE) ist ein Verfahren der symmetrischen Kryptografie, das sicherstellt, dass Daten während der Übertragung oder Speicherung sowohl verschlüsselt als auch authentifiziert werden. Damit erfüllt AE zwei entscheidende Sicherheitsziele gleichzeitig[18, S.2].

Im Vergleich zu herkömmlichen Verschlüsselungsverfahren, die lediglich Vertraulichkeit gewährleisten, bietet AE zusätzliche Schutzmaßnahmen durch die Einbindung eines Message Authentication Codes (MAC) oder Tag. Dieser Tag wird zusammen mit dem verschlüsselten Text generiert und gibt dem Empfänger die Möglichkeit, die Echtheit und Unversehrtheit der Nachricht zu prüfen. Die Funktion schützt vor Angriffen, bei denen ein Angreifer Datenpakete abfangen und unbemerkt manipulieren könnte. Indem der Tag bei der Entschlüsselung auf seine Gültigkeit überprüft wird, kann jede Art von Manipulation sofort erkannt werden[27, S.35].

AE wird vor allem in sicherheitskritischen Anwendungen eingesetzt, bei denen sowohl die Vertraulichkeit als auch die Integrität von Informationen gewährleistet sein müssen. Typische Anwendungsgebiete sind sichere Kommunikationsprotokolle wie Transport Layer Security (TLS), die Verschlüsselung in Datenbanken sowie FUSE-basierte verschlüsselte Dateisysteme. Hierdurch ist AE zu einem unverzichtbaren Werkzeug in der modernen Kryptografie geworden, um die Vertraulichkeit und Integrität von Daten in einem einzigen Schritt sicherzustellen. Hier ist es besonders wichtig, dass verschlüsselte Dateien nicht nur sicher gespeichert, sondern auch vor Manipulationen geschützt sind[12, S.1-3].

In modernen AE-Schemata, wie sie in POET verwendet werden, wird zudem eine Nonce (Number used once) verwendet, um jede Verschlüsselungsoperation eindeutig zu machen. Die Nonce wird typischerweise in jedem Verschlüsselungsvorgang neu generiert und kann mit dem verschlüsselten Text übertragen werden, ohne die Sicherheit zu gefährden.

Um die Vertraulichkeit und Integrität von Daten zu gewährleisten, wurde eine Vielzahl von AE Verfahren entwickelt, die sich in ihrer Effizienz, Sicherheit und praktischen Anwendung unterscheiden. Jedes AE-Schema ist dabei auf spezifische Anforderungen ausgelegt und bietet besondere Vorteile je nach Einsatzgebiet. Zu den gängigsten Verfahren gehören AES-GCM (Galois/Counter Mode) und OCB (Offset Codebook Mode), die in zahlreichen modernen Sicherheitsanwendungen weit verbreitet sind. Ein weiteres Verfahren, welches diese Vorteile mitbringt ist das POET-[12, S.4].

### 2.2.4. Beschreibung des POET-Algorithmus

Der POET-Algorithmus (Pipelineable Online Encryption with Authenticated Tag) wurde entwickelt, um sowohl die Vertraulichkeit als auch die Integrität von Daten sicherzustellen. Durch seine einzigartige Kombination von AES-Blockverschlüsselung und einem Mechanismus zur Authentifizierung bietet POET eine effiziente und robuste Lösung für Anwendungen, die kontinuierliche Verschlüsselung und Schutz vor Manipulation erfordern. Hier liegen die wesentlichen

Bestandteile und die Funktionsweise des POET-Algorithmus [7, 1-2].

### Rolle der Nonce und des Headers bei der Verschlüsselung in POET

Die Verarbeitung des Headers im POET-Algorithmus umfasst die Einbettung zusätzlicher Daten (Associated Data) der Nachricht sowie ein n-Bit-Nonce am Ende, wodurch der gesamte Header als Nonce interpretiert werden kann. Die Einbettung eines Nonce im Header ist für den POET-Algorithmus von zentraler Bedeutung, da sie verhindert, dass identische Nachrichten denselben Chiffretext erzeugen, was Replay- oder Wiederholungsangriffe verhindert. Der Header kann darüber hinaus zusätzliche Verifikationsinformationen enthalten, die bei der Entschlüsselung nützlich sind. Diese Datenstruktur stellt sicher, dass jeder Chiffretext eindeutig bleibt und dass jede Verschlüsselungsoperation in POET unabhängig von anderen Operationen abläuft, was zur Gesamtintegrität und Sicherheit des Algorithmus beiträgt.[7, S.15-17]. Die nächste Abbildung zeigt die Header Verarbeitung.

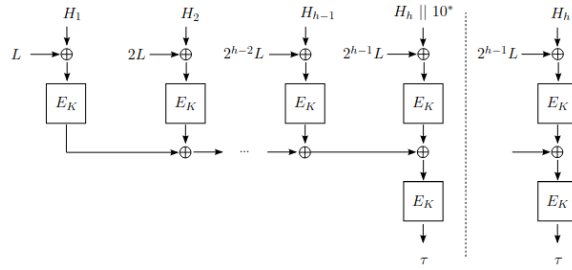


Abbildung 2.9.: POET-ProcessHeader[7, S.17]

### POET Verschlüsselung

Der Verschlüsselungsprozess im POET-Algorithmus besteht aus drei Hauptschichten, die zusammenarbeiten, um die Nachricht in Blöcke aufzuteilen und sicher zu verschlüsseln. Der Prozess beginnt mit den Initialwerten  $X_0$  und  $Y_0$ , die den Ausgangspunkt für die Verkettung in den oberen und unteren Schichten bilden.  $X_0$  dient dabei als Nonce, also als einmaliger Wert, der für jede Nachricht neu generiert wird.  $Y_0$  wird üblicherweise durch die Verschlüsselung von  $X_0$  mit der Blockchiffre erzeugt, also  $Y_0 = E_K(X_0)$ , was sicherstellt, dass die beiden Startwerte miteinander verknüpft, aber dennoch unterschiedlich sind.

In der ersten Schicht, der sogenannten oberen Schicht oder *Top-Row Layer*, wird jeder Nachrichtenblock  $M_i$  mit der Funktion  $F_{K_F}$  kombiniert, die auf den vorherigen Verkettungswert  $X_{i-1}$  angewendet wird. Das Ergebnis dieser Funktion wird dann mit dem aktuellen Nachrichtenblock  $M_i$  per XOR verknüpft, wodurch ein neuer Verkettungswert  $X_i$  entsteht. Dieser Wert  $X_i$  dient anschließend als Eingabe für die Blockchiffre  $E_K$ . Die Blockchiffre  $E_K$  ist eine Verschlüsselungsfunktion, die den Wert  $X_i$  unter Verwendung des Schlüssels  $K$  verschlüsselt, wodurch der verschlüsselte Wert  $Y_i$  entsteht, der an die nächste Schicht weitergegeben wird.

Die zweite Schicht, die mittlere Schicht oder *Middle Layer*, besteht im Wesentlichen aus der

Blockchiffre  $E_K$  selbst. Diese Schicht verstärkt die Sicherheit, indem der in der ersten Schicht erzeugte Verkettungswert  $X_i$  weiter verschlüsselt wird, was dazu führt, dass die Daten noch schwerer analysierbar und gegen Angriffe geschützt sind.

In der unteren Schicht oder *Bottom-Row Layer* wird der zuvor verschlüsselte Wert  $Y_{i-1}$  ebenfalls durch  $F_{K_F}$  verarbeitet. Das Ergebnis dieser Funktion wird dann mit dem verschlüsselten Wert  $Y_i$  per XOR kombiniert, um den endgültigen Chiffretextblock  $C_i$  zu erzeugen. Für den letzten Nachrichtenblock wird zusätzlich der Wert  $S$ , der die verschlüsselte Länge der Nachricht darstellt, mit dem letzten Nachrichtenblock  $M_m$  kombiniert und durch die gleichen Schritte wie die vorherigen Blöcke verarbeitet.

Am Ende des Verschlüsselungsprozesses wird ein spezieller Wert  $T$ , das sogenannte Authentifizierungstag, generiert, dass die Integrität der verschlüsselten Nachricht sicherstellt. Dieses Tag wird aus dem Zwischenwert  $\tau$ , der durch die Header-Verarbeitung erzeugt wurde, und dem letzten Chiffretextblock  $C_m$  abgeleitet. Das Diagramm verdeutlicht, wie die drei Schichten zusammenwirken, um sowohl die Vertraulichkeit als auch die Authentizität der Daten zu gewährleisten. Der Aufbau dieser Schichten verstärkt die Sicherheit, da jede Schicht die Daten weiter verschlüsselt und die Abhängigkeit der Blöcke untereinander erhöht, wodurch der POET-Algorithmus besonders widerstandsfähig gegen kryptografische Angriffe ist. [7, S.14-16].

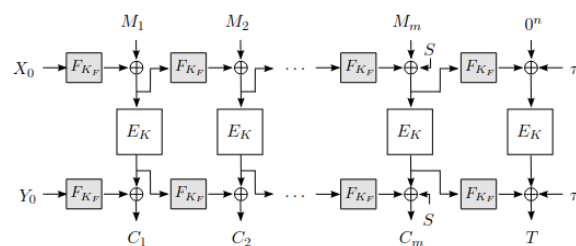


Abbildung 2.10.: POET-Block[7, S.14]

Ein entscheidender Schritt im POET-Algorithmus ist die Erzeugung eines Tags am Ende des Verschlüsselungsprozesses. Dieser Tag fungiert als eine Art Message Authentication Code (MAC) und wird zusammen mit dem Chiffretext gespeichert. Bei der Entschlüsselung prüft POET den Tag, um sicherzustellen, dass die Daten nicht manipuliert wurden. Nur wenn der Tag mit dem beim Verschlüsseln generierten übereinstimmt, wird die Integrität der Daten bestätigt und der Empfänger kann sicher sein, dass die Nachricht unverändert und authentisch ist.

Die Kombination von Nonce, AES-Blöcken und dem Tag macht POET zu einem umfassenden AEAD-Verfahren (Authenticated Encryption with Associated Data), das eine vollständige Absicherung der Daten bietet. Die Tag-Erzeugung und -Überprüfung verleihen POET seine Authentifizierungsfunktion, die verhindert, dass Angreifer unbemerkt Veränderungen am Chiffretext vornehmen können[7, S.17-19].

## **Authenticated Encryption with Associated Data (AEAD) im Bezug auf POET**

In einem AEAD-Schema wie POET wird die Nonce, eine einmalig generierte Zufallszahl, verwendet, um sicherzustellen, dass jede Verschlüsselungsoperation ein einzigartiges Ergebnis liefert. Diese Nonce wird jedoch nicht selbst verschlüsselt, sondern im Klartext zusammen mit dem Chiffretext gesendet. Ähnlich verhält es sich mit dem Tag, der als Authentifizierungssignatur dient und eine Prüfsumme über den gesamten verschlüsselten Datenblock bildet. Der Tag wird bei der Entschlüsselung verwendet, um zu verifizieren, dass der Chiffretext authentisch und unverändert ist.

Obwohl Nonce und Tag im Klartext übertragen werden, spielen sie eine entscheidende Rolle für die Integrität und Sicherheit des Chiffretexts. Der Tag wird durch eine Authentifizierungsfunktion erstellt, die über die gesamte Nachricht berechnet wird und so jede Manipulation am Chiffretext aufdeckt. Da der Chiffretext nur korrekt entschlüsselt werden kann, wenn die Nonce und der Tag stimmen, gewährleisten Nonce und Tag im AEAD-Schema, dass die Integrität und Authentizität der verschlüsselten Daten auch ohne zusätzliche Verschlüsselung sichergestellt sind[7, S.1].

## **2.3. Catena als KDF**

### **2.3.1. Erklärung der Notwendigkeit sicherer Key Derivation Functions (KDFs)**

Die sichere Ableitung kryptografischer Schlüssel aus Passwörtern ist essenziell, um die Integrität und Vertraulichkeit von Daten zu gewährleisten. Passwörter allein sind oft nicht ausreichend sicher, da sie kurz oder leicht zu erraten sein können. Durch die Anwendung von Schlüsselableitungsfunktionen werden Passwörter in starke, kryptografisch sichere Schlüssel umgewandelt. Diese Funktionen erhöhen die Komplexität und Länge des Schlüssels, was Brute-Force- und Wörterbuchangriffe erheblich erschwert.

Ein gängiges Verfahren ist die Verwendung von PBKDF2 (Password-Based Key Derivation Function 2), das durch mehrfache Iterationen und die Hinzufügung eines Salt-Werts die Sicherheit weiter steigert. Ohne solche Verfahren wären verschlüsselte Daten anfälliger für Angriffe, da schwache Passwörter direkt zu unsicheren Schlüsseln führen könnten. Daher ist die sichere Ableitung von Schlüsseln aus Passwörtern ein unverzichtbarer Bestandteil moderner Kryptografie[21, S.487-490].

Die Schlüsselstärke wird in der Regel in Bits angegeben und beschreibt die Anzahl möglicher Kombinationen, die ein Angreifer durchprobieren müsste, um den Schlüssel zu erraten. Für die meisten modernen Anwendungen wird ein 128-Bit-Schlüssel als ausreichend sicher betrachtet, da die Anzahl möglicher Kombinationen besonders hoch ist  $2^{128}$ . Ein Schlüssel dieser Länge würde bei heutigen technischen Mitteln selbst für leistungsstarke Angreifer und Systeme Jahrtausende zum Durchprobieren erfordern. Jedoch ist ein starker Schlüssel mehr als nur eine ausreichende

Bitlänge. Es ist ebenso wichtig, dass der Schlüssel tatsächlich zufällig und ohne Vorhersehbarkeit generiert wird, um eine Schwächung der Sicherheit zu verhindern. Ein schlecht gewählter 128-Bit-Schlüssel kann trotz ausreichender Länge potenziell schwächer sein, als die bloße Bitgröße vermuten lässt[21, S.13-14]

### 2.3.2. Key Derivation Functions (KDF)

Um sicherzustellen, dass Schlüssel eine geeignete Qualität haben, insbesondere wenn sie von Passwörtern abgeleitet werden, kommen sogenannte KDFs zum Einsatz. KDFs sind kryptografische Funktionen, die aus einem Passwort einen starken Schlüssel generieren, indem sie komplexe Algorithmen und Zusatzparameter wie Salts und Iterationen nutzen, um die Sicherheit zu erhöhen. Es hilft sicherzustellen, dass ein aus einem Passwort abgeleiteter Schlüssel eine hohe Entropie aufweist und schwer zu erraten ist[6, S.11-12].

Ein Salt ist eine zufällige, eindeutige Zeichenfolge, die einem Passwort hinzugefügt wird, bevor es durch eine KDF verarbeitet wird. Der Hauptzweck eines Salts besteht darin, sicherzustellen, dass selbst bei identischen Passwörtern unterschiedliche Schlüssel generiert werden. So wird verhindert, dass Angreifer sogenannte Rainbow Tables – vorberechnete Tabellen mit Passwort-Hash-Paaren – nutzen können, um Passwörter schnell zu erraten. Ein Salt macht also jeden abgeleiteten Schlüssel individuell und erhöht die Sicherheit.

In modernen KDFs wie PBKDF2, Argon2 und Catena werden deutliche Unterschiede in der Widerstandsfähigkeit gegenüber spezialisierten Angriffen und in der Effizienz deutlich, insbesondere hinsichtlich der Cache-Miss-Resistenz, die Grafikprozessor-angetriebene Brute-Force-Angriffe erschwert.

Grafikprozessoren (GPUs) sind besonders leistungsstarke Prozessoren, die für parallele Berechnungen ausgelegt sind. Ihre Architektur mit vielen Kernen eignet sich hervorragend für rechenintensive Aufgaben, insbesondere wenn sich Berechnungen leicht parallelisieren lassen, wie es bei der Brute-Force-Berechnung von Hash-Werten der Fall ist. Die Schwäche von GPUs im Kontext sicherheitsrelevanter Berechnungen liegt jedoch oft im Speicherzugriff, insbesondere bei der sogenannten Cache-Miss-Resistenz[3, S.76-78].

#### Cache-Miss-Resistenz und GPUs

Cache Misses entstehen, wenn die Daten, die für eine Berechnung benötigt werden, nicht im schnellen Cache-Speicher vorhanden sind und stattdessen aus dem Hauptspeicher abgerufen werden müssen. Dieser Zugriff auf den Hauptspeicher ist langsamer und verursacht Verzögerungen, was besonders auf GPUs zu einem Performance-Engpass führt. GPUs haben zwar leistungsstarke Rechenkerne, jedoch relativ begrenzten Cache-Speicher pro Kern. Dies bedeutet, dass Berechnungen, die häufige und zufällige Speicherzugriffe erfordern, auf GPUs erheblich an Effizienz verlieren, da sie immer wieder den Hauptspeicher abfragen müssen[29].

## **Cache-Miss-Resistenz bei KDFs**

KDFs wie Argon2 und Catena machen sich dieses GPU-Problem zunutze, indem sie Speicherzugriffsmuster verwenden, die gezielt Cache Misses verursachen. Das Konzept ist die Fähigkeit eines Algorithmus, durch viele gezielte Speicherzugriffe den Cache zu überlasten und die Berechnung dadurch auf die langsameren Hauptspeicherzugriffe umzulenken. Diese Art der Speicherzugriffsmuster führt zu einer verstärkten Memory-Hardness und stellt sicher, dass GPUs ihre Rechenleistung nicht effizient nutzen können, weil sie durch die ständigen Cache Misses gebremst werden[8, S.17].

### **Techniken zur Erhöhung der Cache-Miss-Resistenz**

Argon2 und Catena verwenden große Tabellen und zufällige Speicherzugriffe über diese Tabellen hinweg. Diese Tabellen[8, S.49], die oft mithilfe von Parametern wie dem Garlic-Parameter in Catena oder der Memory-Hardness-Variable in Argon2[2] skaliert werden, erzeugen eine große Menge an Speicherzugriffen, die nicht vollständig im Cache gehalten werden können. Dadurch steigt die Anzahl der Cache Misses, was es schwierig macht, den Hashing-Prozess auf parallelen GPU-Kernen effizient auszuführen[8, S.6].

Catena geht noch einen Schritt weiter und verwendet die Double-Butterfly-Hashing-Struktur (DBH), die Cache Misses absichtlich erzeugt, um die Speichernutzung zu intensivieren und die Cache-Timing-Resistenz zu erhöhen. Mit dem Garlic-Parameter kann der Speicherbedarf flexibel erhöht werden, was Angriffe mit GPUs zusätzlich erschwert. Durch die Kombination von passwortunabhängigen Speicherzugriffen und einer gezielten Memory-Hardness bietet Catena starke Sicherheit gegen parallele Angriffe, einschließlich solcher, die GPUs oder spezialisierte Hardware verwenden[8, S.23].

### **2.3.3. Eigenschaften und Sicherheitsmerkmale von Catena (z.B. Brute-Force-Resistenz und FPGA-Resistenz)**

Catena zeichnet sich durch seine starke Memory-Hardness aus, was bedeutet, dass der Algorithmus einen großen Speicherbedarf für die Berechnung eines Hashes voraussetzt. Dies macht ihn widerstandsfähig gegen Brute-Force-Angriffe, insbesondere solche, die auf parallelen oder spezialisierten Hardware-Ressourcen wie FPGAs basieren. Diese Angriffe benötigen eine erhebliche Menge an Speicher, um den Prozess zu beschleunigen. Da FPGAs und GPUs nur begrenzt schnellen Speicherzugriff bieten, werden durch den hohen Speicherbedarf von Catena und die gezielte Erzeugung von Cache Misses potenzielle Geschwindigkeitsvorteile solcher Hardware nahezu ausgeglichen[8, S.12-13].

## **Flexibilität und Einfluss auf die Sicherheit**

Catena ist flexibel in der Konfiguration und kann durch Parameter wie den Garlic-Parameter an verschiedene Sicherheitsanforderungen angepasst werden. Der Garlic-Parameter bestimmt den Speicherbedarf des Algorithmus, wobei höhere Werte exponentiell mehr Speicher und Rechenzeit erfordern. Diese Flexibilität ermöglicht es, Catena auf unterschiedlich leistungsfähigen Geräten zu betreiben, wobei die Sicherheit durch die Anpassung des Speicherbedarfs kontrolliert werden kann. Catena ist außerdem so konzipiert, dass es resistent gegenüber Cache-Timing-Angriffen ist, da die Speicherzugriffe unabhängig von der Eingabe erfolgen[S.26-27][8].

## **Double-Butterfly-Hashing-Struktur**

Ein zentrales Element in Catena ist die Double-Butterfly-Hashing-Struktur (DBH), die für zusätzliche Sicherheit und Effizienz sorgt. Die DBH-Struktur organisiert die Speicherzugriffe in einem Schmetterlingsmuster, bei dem der Algorithmus über große Speicherbereiche hinweg in zufälligen Intervallen Daten abrufen. Dieser Prozess erzeugt viele Cache Misses und erschwert es Angreifern, den Zugriffsmuster zu folgen oder den Prozess effizient zu parallelisieren. Das Double-Butterfly-Muster verstärkt die Memory-Hardness und erschwert parallele Berechnungen, da die Datenzugriffe nicht im Cache gehalten werden können und ständig auf den Hauptspeicher zugegriffen werden muss[8, S.39-40].

## **2.4. Pluggable Authentication Modules (PAM)**

Die Pluggable Authentication Modules (PAM) bieten unter Linux ein flexibles und vielseitiges Framework zur Benutzer-Authentifizierung und ermöglichen eine sichere und anpassbare Zugriffssteuerung auf Systeme und Anwendungen. Mit PAM lässt sich der Authentifizierungsprozess modular gestalten, verschiedene Module können kombiniert werden, um eine Vielzahl an Sicherheitsfunktionen umzusetzen. Dazu gehören etwa die Verifizierung von Passwörtern, die Integration von Mehrfaktor-Authentifizierungsmethoden oder das Einbinden externer Skripte, die zusätzliche sicherheitsrelevante Prozesse steuern[11].



Diese modulare Struktur erlaubt es Administratoren, spezifische Sicherheitsanforderungen für verschiedene Anwendungen und Benutzergruppen umzusetzen, ohne dabei das Betriebssystem selbst anpassen zu müssen. PAM gewährleistet so eine hohe Flexibilität und ermöglicht, den Authentifizierungsprozess auf individuelle Sicherheitsanforderungen abzustimmen und gleichzeitig die Komplexität der Verwaltung zu reduzieren.

In diesem Projekt wird PAM genutzt, um den Zugang zum verschlüsselten Dateisystem zu steuern und abzusichern. Das PAM-Framework überprüft die Identität des Benutzers. Durch diese Integration wird sichergestellt, dass nur der authentifizierte Benutzer Zugriff auf seine verschlüsselten Daten erhält und der Anmeldeprozess zugleich benutzerfreundlich und sicher abläuft. PAM besitzt Konfigurationsdateien, die es ermöglichen, indem sie die Reihenfolge und Bedingungen festlegt, unter denen die einzelnen PAM-Module im Authentifizierungsprozess ausgeführt werden[28].

#### **2.4.1. Python mit PAM**

Python bietet eine Vielzahl an vorgefertigten Bibliotheken, die die Entwicklung und Implementierung von Projekten erheblich erleichtern. Eine besonders nützliche Bibliothek in diesem Kontext ist `python-pam`, die eine einfache Schnittstelle bereitstellt, um PAM-Funktionalitäten direkt in Python-Programmen zu nutzen. Diese Bibliothek vereinfacht die Authentifizierung von Benutzern, die Verwaltung von Sitzungen und die Implementierung zusätzlicher sicherheitsrelevanter Prozesse.

Der Einsatz von Python für PAM-Module bringt mehrere Vorteile mit sich, insbesondere hinsichtlich Flexibilität. Python erlaubt es, den Authentifizierungsprozess an spezifische Anforderungen anzupassen und erweiterte Automatisierungsprozesse zu realisieren. Dies ist besonders hilfreich für Systeme, bei denen Anpassungen an die Sicherheitsanforderungen erforderlich sind und in denen Authentifizierung und Sitzungsmanagement automatisiert und flexibel gestaltet werden sollen[25].

Durch die Kombination von Python und PAM wird der Entwicklungsprozess schneller und effizienter, zahlreiche Automatisierungsfunktionen und eine große Anzahl von Bibliotheken zur Verfügung stellt.

### 3. Konzept

In diesem Abschnitt wird eine Gesamtübersicht über das System dargestellt, um die Architektur und die Zusammenarbeit der einzelnen Komponenten zu verdeutlichen. Die Analyse zeigt, wie jede Komponente zum sicheren und effizienten Ablauf der Verschlüsselung beiträgt und wie die Datenintegrität und -sicherheit gewährleistet wird.

Das System ist modular aufgebaut und verwendet eine Kombination aus Authentifizierung, Schlüsselgenerierung, Datenverschlüsselung und -speicherung, um die Daten des Benutzers in einer sicheren Umgebung zu verarbeiten und zu speichern. Durch diese Struktur ist das System sowohl skalierbar als auch flexibel für zukünftige Anpassungen. Im Folgenden wird die Funktionsweise und Interaktion der einzelnen Komponenten dargestellt.

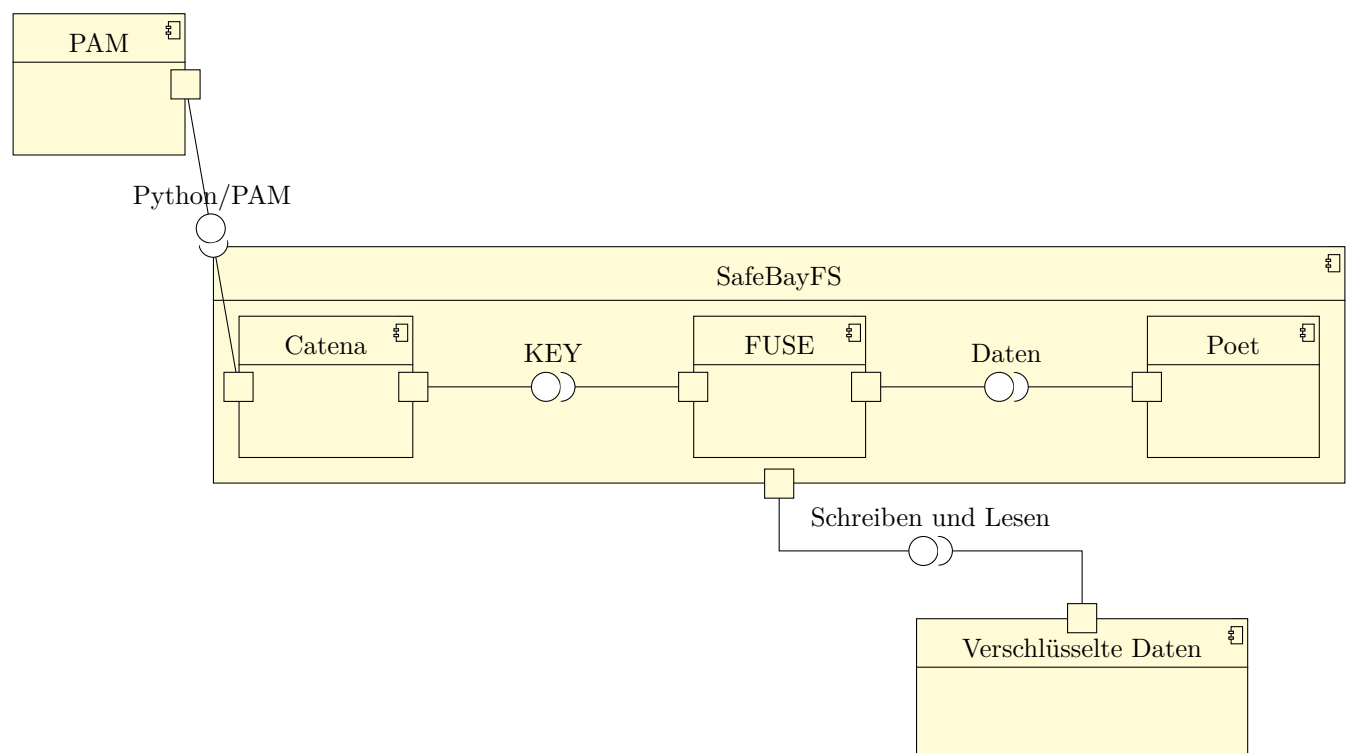


Abbildung 3.1.: Komponentendarstellung SafeBayFS

In dem oben gezeigten Komponentendiagramm wird dargestellt, wie die einzelnen Module zusammenarbeiten und wie das System gestaltet ist.

1. Der Benutzer wird über das PAM-Modul authentifiziert und das Passwort wird mit Python/PAM an Catena weitergeleitet.
2. Die Catena-Komponente generiert einen Schlüssel, der an die FUSE-Komponente weitergegeben wird. Dieser Schlüssel dient als Verschlüsselungsschlüssel für die Poet-Komponente.
3. Die Daten werden an Poet gesendet. Poet verschlüsselt diese mit dem generierten Schlüssel von Catena und schickt sie wieder zurück an FUSE.
4. Bei jedem Schreib- und Lesevorgang werden die Daten von FUSE an Poet weitergeleitet, und Poet verschlüsselt oder entschlüsselt diese.

### **3.1. Softwareanalyse**

Die Softwareanalyse ist ein zentraler Schritt bei der Entwicklung eines Verschlüsselungssystems. Sie dient dazu, die Anforderungen zu identifizieren, geeignete Technologien zu validieren und potenzielle Risiken im Vorfeld zu erkennen, um sie durch entsprechende Maßnahmen zu minimieren.

#### **3.1.1. Zieldefinition**

Das Hauptziel des Systems ist es, ein transparentes Verschlüsselungssystem zu implementieren, das auf Linux-basierten Betriebssystemen funktioniert. Der Fokus liegt darauf, den Inhalt der Daten sicher zu verschlüsseln, ohne dass der Endbenutzer technische Kenntnisse oder zusätzliche Schritte benötigt. Der gesamte Prozess – von der Schlüsselerzeugung bis zur Verschlüsselung und Entschlüsselung der Daten – soll automatisch und im Hintergrund ablaufen. Dadurch wird gewährleistet, dass das System einfach zu bedienen ist und dennoch hohe Sicherheitsstandards erfüllt.

#### **3.1.2. Stakeholder-Analyse**

##### **Endbenutzer**

Die primäre Zielgruppe des Systems sind technisch nicht versierte Nutzer, die ein sicheres Verschlüsselungssystem benötigen, das ohne zusätzliche Schritte funktioniert. Für den Benutzer muss der Zugriff auf verschlüsselte Verzeichnisse genauso einfach sein wie bei unverschlüsselten Verzeichnissen.

##### **Entwickler**

Die das System in der Zukunft erweitern oder warten müssen, benötigen eine robuste und gut dokumentierte Architektur, die es ermöglicht, neue Sicherheitsfunktionen hinzuzufügen oder das System auf zukünftige Bedrohungen anzupassen.

### 3.1.3. Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die wesentlichen Funktionen, die das Verschlüsselungssystem bieten muss:

- **Transparente Verschlüsselung und Entschlüsselung**

Das System muss sicherstellen, dass alle Dateien in den verschlüsselten Verzeichnissen automatisch verschlüsselt werden, ohne dass der Benutzer dies bemerkt. Lese- und Schreibvorgänge in diesen Verzeichnissen müssen so transparent wie bei unverschlüsselten Verzeichnissen ablaufen.

- **Automatische Schlüsselerzeugung**

Beim Benutzerlogin wird mithilfe von Catena automatisch ein kryptografischer Schlüssel aus dem Passwort des Benutzers abgeleitet. Dieser Schlüssel wird dann für die Verschlüsselung und Entschlüsselung der Dateien verwendet.

- **FUSE-Integration**

Die Integration von FUSE sorgt dafür, dass alle Dateioperationen (Lese- und Schreibvorgänge) abgefangen werden und die verschlüsselten Daten sicher auf das Dateisystem geschrieben oder von diesem gelesen werden können.

- **Integration mit PAM**

Das System muss sicherstellen, dass beim Benutzerlogin automatisch der Schlüssel abgeleitet und das verschlüsselte Verzeichnis gemountet wird, sodass der Benutzer sofortigen Zugriff auf seine Daten hat.

- **Fehlerprotokollierung**

Es müssen Protokolle erstellt werden, die sicherstellen, dass auf mögliche Fehler oder Sicherheitsprobleme schnell reagiert werden kann.

### 3.1.4. Nicht-funktionale Anforderungen

Neben den funktionalen Anforderungen gibt es mehrere nicht-funktionale Anforderungen, die erfüllt werden müssen:

- **Sicherheit**

Der abgeleitete Schlüssel darf niemals im Klartext gespeichert oder unverschlüsselt übertragen werden. Das System muss resistent gegen typische Attacken wie Brute-Force-Angriffe sein.

- **Performance**

Die Verschlüsselung und Entschlüsselung der Dateien darf die Systemleistung nur minimal beeinträchtigen. Der Zugriff auf verschlüsselte Verzeichnisse darf nur mit einer geringen Verzögerung erfolgen, wie auf unverschlüsselte Verzeichnisse.

- **Benutzerfreundlichkeit**

Das System muss für den Endbenutzer vollständig transparent arbeiten. Der Benutzer soll keine zusätzlichen Schritte unternehmen müssen, um auf verschlüsselte Verzeichnisse zuzugreifen.

- **Skalierbarkeit**

Das System muss in der Lage sein, mit großen Datenmengen effizient zu arbeiten, ohne die Arbeit des Users am System zu beeinträchtigen.

- **Wartbarkeit**

Das System muss einfach zu warten und erweiterbar sein, insbesondere im Hinblick auf neue Verschlüsselungsalgorithmen oder Sicherheitsupdates.

### 3.1.5. Technologische Analyse

Die Technologien, die in diesem System zum Einsatz kommen, sind sorgfältig ausgewählt, um die gewünschten Funktionen und Sicherheitsanforderungen zu erfüllen:

- **FUSE**

Ermöglicht es, ein benutzerdefiniertes Dateisystem im Userspace zu implementieren, ohne den Kernel anzupassen. Es bietet die Möglichkeit, Dateioperationen abzufangen und sicherzustellen, dass die Daten beim Schreiben verschlüsselt und beim Lesen entschlüsselt werden.

- **POET**

Ist ein AE Schema, das sicherstellt, dass die Daten sowohl verschlüsselt als auch authentifiziert werden. Es schützt die Daten nicht nur vor unbefugtem Zugriff, sondern stellt auch sicher, dass Manipulationen an den Daten erkannt werden.

- **Catena**

Ist eine moderne Key-Derivation-Funktion, die resistent gegen Brute-Force-Angriffe ist. Sie sorgt dafür, dass der kryptografische Schlüssel sicher aus dem Benutzerpasswort abgeleitet wird.

- **Python**

Wird verwendet, um die Automatisierung des Schlüsselaustauschprozesses und des Mountens der verschlüsselten Verzeichnisse zu steuern.

- **PAM**

Übernimmt den Authentifizierungsprozess und leitet automatisch den Schlüssel mit Catena ab, sobald sich der Benutzer anmeldet.

Ein Angreifer könnte versuchen, das Benutzerpasswort zu stehlen und damit den Verschlüsselungsschlüssel zu erhalten. Catena bietet Schutz, indem es durch speicherintensive Berechnungen den Aufwand für Brute-Force-Angriffe deutlich erhöht.

- **Systemverlust**

Wenn ein Gerät verloren geht oder gestohlen wird, darf der Angreifer keinen Zugriff auf die verschlüsselten Verzeichnisse haben. Da der Schlüssel nur während des Logins abgeleitet wird und nicht auf dem Gerät gespeichert ist, bietet das System zusätzlichen Schutz.

- **Datenmanipulation**

Manipulationen an den verschlüsselten Daten müssen erkannt werden. POET stellt sicher, dass jede Datei authentifiziert wird, um sicherzustellen, dass keine Daten unbemerkt verändert wurden.

**Technische Risiken:**

- Performance-Engpässe: Durch die Verwendung von Verschlüsselung könnte die Systemleistung bei großen Datenmengen beeinträchtigt werden.
- Komplexität der Integration: Die Integration von FUSE, PAM, Catena und POET könnte technisch anspruchsvoll sein.

Zusammen bilden FUSE, POET und Catena eine sichere und transparente Verschlüsselungsinfrastruktur, die sowohl die Vertraulichkeit als auch die Integrität der Daten gewährleistet und gleichzeitig sicherstellt, dass der Benutzer keinen zusätzlichen Aufwand bei der Nutzung des verschlüsselten Dateisystems bemerkt.

## 4. Implementierung

Alle Implementierungen wurden auf einem ThinkPad T490s entwickelt, das mit Ubuntu 22.04.5 LTS (64-Bit) als Betriebssystem betrieben wird. Der Laptop verfügt über einen Intel Core i7-8665U Prozessor mit einer maximalen Taktfrequenz von 4,8 GHz. Das System ist mit 16 GB Arbeitsspeicher ausgestattet. Die Kernel-Version des Betriebssystems ist 6.8.0-48-generic.

Zum Kompilieren des Codes wird der Clang-Compiler in der Version 14.0.0-1ubuntu1.1 verwendet. Die Entwicklungsumgebung ist CLion. GNOME 42.9 wird als Desktop-Umgebung genutzt und das GNOME-Terminal dient zur Ausführung von Kompilierungs- und Testprozessen. Alle Tests und Messungen sind ebenso auf dem gleichen System durchgeführt, die in Kapitel 5 vorgestellt werden.

### 4.1. Automatisierung und Sicherheit durch Python und PAM

Um alle Anforderungen aus Abschnitt 3 zu gewährleisten, wird die Implementierung so gestaltet, dass der gesamte Ablauf – von der Anmeldung bis zur sicheren Schlüsselableitung, sowie dem automatischen Mounten des verschlüsselten Dateisystems – automatisiert und benutzerfreundlich erfolgt. Das Python-Skript und das PAM-Modul werden so konfiguriert, dass die Authentifizierung nahtlos abläuft und nur der angemeldete Benutzer Zugriff auf die verschlüsselten Daten erhält.

Um das Passwort des Benutzers bei der Erstanmeldung sicher an Catena weiterzuleiten, müssen spezifische Anpassungen in PAM vorgenommen werden. Das verschlüsselte Dateisystem wird wie jedes andere Dateisystem direkt nach der Anmeldung bereitgestellt und steht sofort zur Verfügung. Diese Automatisierung sorgt für eine benutzerfreundliche und transparente Integration von SafeBayFS in den Anmeldeprozess, ohne die Sicherheit des Systems zu gefährden.

Das Übergeben eines Passworts von PAM an ein Python-Skript erfordert eine sorgfältige und präzise Konfiguration. Standardmäßig erlaubt PAM keine Weitergabe sensibler Daten wie Passwörter an externe Programme. Allerdings kann diese Funktionalität mithilfe der Option `expose_auth_tok` ermöglicht werden.

## Überblick über benutzte PAM Funktionen

PAM ist ein flexibles Framework zur Authentifizierung, das über Module angepasst werden kann. Eines dieser Module ist `pam_exec.so`. Es ermöglicht das Ausführen externer Programme während der verschiedenen Phasen der Authentifizierung oder Sitzungsverwaltung.

Die Option `expose_authtok` erlaubt es, das vom Benutzer eingegebene Passwort (das sogenannte `authtok`, Authentication Token) an ein Programm weiterzugeben. Dieses ist jedoch mit Vorsicht zu verwenden, da die Weitergabe sensibler Daten immer sicher und kontrolliert erfolgen muss.

## Konfiguration in den PAM-Dateien

Die relevanten Konfigurationsdateien für PAM befinden sich im Verzeichnis `/etc/pam.d/`. Um das Passwort erfolgreich an ein Python-Skript zu übergeben, sind Änderungen in zwei spezifischen Dateien erforderlich

- **common-auth:** Zuständig für die Authentifizierung des Benutzers.
- **common-session:** Verwaltet Aktionen, die während der Sitzungsverwaltung ausgeführt werden.

## Anpassung in common-auth

Die Datei `common-auth` steuert die Authentifizierungsphase. Die folgende Konfiguration ermöglicht es, das Passwort an ein externes Python-Skript zu senden

---

```
1 auth      optional      pam_exec.so expose_authtok debug /usr/bin/python3 /home/
    uni/Dokumente/Master/VL/SafeBayFS/pam_stdin.py >> /tmp/pam_exec_session.
    log 2>&1
```

---

*Listing 4.1:* PAM-Konfiguration von `common-auth` `pam_stdin.py`

- **auth:** Die Zeile wird während der Authentifizierung des Benutzers ausgeführt.
- **optional:** Das Modul ist optional, d. h. auch wenn das Programm fehlschlägt, wird die Authentifizierung fortgesetzt.
- **pam\_exec.so:** Führt das angegebene externe Programm aus.
- **expose\_authtok:** Übergibt das Passwort des Benutzers sicher über die Standardeingabe (`stdin`) an das Skript.
- **debug:** Aktiviert zusätzliche Debugging-Informationen, die in den PAM-Logs gespeichert werden.
- **/usr/bin/python3 /PythonScript.py:** Startet das Python-Skript mithilfe von `pam_exec.so`.
- **/tmp/pam\_exec\_auth.log 2>&1:** Leitet die Standardausgabe und Fehlerausgabe des Skripts in eine Log-Datei um.



## Anpassung in common-session

Die Datei common-session steuert die Aktionen, die während der Sitzungsverwaltung durchgeführt werden. Die folgende Konfiguration sorgt dafür, dass das Python-Skript mit den Rechten des authentifizierten Benutzers ausgeführt wird

---

```
1 session optional pam_exec.so setuid debug /usr/bin/python3 /PythonScript.py
   >> /tmp/pam_exec_session.log 2>&1
```

---

*Listing 4.2:* PAM-Konfiguration für die Sitzungsphase mit Python-Skript

- session: Diese Zeile wird während der Sitzungsverwaltung ausgeführt.
- optional: Die Ausführung des Moduls ist optional.
- setuid: Führt das Skript mit den Rechten des authentifizierten Benutzers aus, anstatt mit Root-Rechten.
- debug: Aktiviert Debugging-Informationen.

Die Konfiguration von PAM mit pam\_exec.so und expose\_authtok ermöglicht eine sichere Übergabe von Passwörtern an ein Python-Skript.

## Python zur Automatisierung

Das Python-Skript ist darauf ausgelegt, das Passwort eines Benutzers von PAM zu empfangen, es durch Catena zu verarbeiten und anschließend SafeBayFS mit dem generierten Schlüssel zu starten.

Das Passwort wird von PAM über stdin an das Python-Skript weitergegeben. Dadurch wird sichergestellt, dass das Passwort sicher und ohne Umwege an das Skript geliefert wird.

Eine der zentralen Herausforderungen bei der Integration des Python-Skripts bestand darin, sicherzustellen, dass SafeBayFS nicht mit Root-Rechten gestartet wird. Dies ist notwendig, da SafeBayFS Sicherheitsprobleme erkennt und den Start verweigert, wenn es mit Administrator-rechten ausgeführt wird. Um dieses Problem zu lösen, wird das Python-Skript so angepasst, dass die Benutzer- und Gruppenrechte explizit auf die des authentifizierten Benutzers gesetzt werden. Mithilfe der Funktionen os.setuid() und os.setgid() wird die UID (Benutzerkennung) und GID (Gruppenkennung) des Prozesses auf die des Benutzers geändert, der sich authentifiziert hat. Diese Maßnahme garantiert, dass SafeBayFS unter den richtigen Berechtigungen läuft und somit sicher verwendet werden kann, ohne potenzielle Sicherheitslücken durch Root-Rechte zu riskieren wie im folgenden Listing zu sehen ist.

---

```

1 USER_UID = 1000 # UID des Benutzers
2 USER_GID = 1000 # GID des Benutzers
3
4 try:
5     os.setgid(USER_GID)
6     os.setuid(USER_UID)
7     log_message(f"Rechte geändert: UID={os.getuid()}, GID={os.getgid()}")
8 except PermissionError as e:
9     log_message(f"Fehler beim Setzen der Rechte: {e}")
10    sys.exit(1)

```

---

*Listing 4.3: Ändern der Benutzerrechte in Python mit UID und GID*

## Ablauf beim Starten des verschlüsselten Dateisystems

Der folgende Ablauf beschreibt, wie das PAM-Modul und das Python-Skript in den Anmeldeprozess integriert sind und wie der Benutzer Zugriff auf das verschlüsselte Verzeichnis erhält.

### 1. Benutzer-Login

Der Benutzer meldet sich an. PAM startet die Authentifizierung über `pam_unix.so` und überprüft die Zugangsdaten.

### 2. Start des Python-Skripts

Wenn die Anmeldung erfolgreich ist, führt PAM über `pam_exec.so` das Python-Skript aus. Das Skript übernimmt:

- Die Ableitung des Schlüssels aus dem Passwort mit `Catena` und
- das automatische Mounten des verschlüsselten Verzeichnisses über `FUSE`.

### 3. Transparenter Zugriff

Nach Abschluss der Schlüsselableitung und des Mountings hat der Benutzer sofortigen Zugriff auf das verschlüsselte Verzeichnis. PAM und das Python-Skript sorgen dafür, dass dieser Prozess im Hintergrund abläuft und für den Benutzer keine zusätzlichen Schritte erforderlich sind.

Diese PAM-Konfiguration erfüllt die Anforderungen aus Abschnitt 3 und stellt sicher, dass das Verschlüsselungssystem einfach zu bedienen, sicher und effizient ist. Sie ermöglicht eine transparente Authentifizierung und Zugriffskontrolle, die für den Benutzer keine zusätzlichen Schritte erfordert und gleichzeitig einem hohen Sicherheitsstandard entspricht.

## 4.2. Catena als KDF: Zielsetzung und Sicherheit

Die spezielle Implementierung und die Konfigurierbarkeit wichtiger Parameter sorgen dafür, dass Catena folgende Sicherheitsziele erfüllt:

- Schutz gegen brute-force-Angriffe
- Anpassbare Sicherheitseinstellungen Parameter wie Garlic und Lambda ermöglichen eine Feinabstimmung der Sicherheits- und Leistungsanforderungen.

### 4.2.1. Erläuterung der Parameter: Garlic und Lambda

Der Garlic-Wert (hier als GARLIC definiert) gibt die Anzahl der Speicherzugriffe in der KDF an. Er steht in einer Beziehung zur Speicheranforderung: Je höher der Garlic-Wert, desto mehr Speicher benötigt Catena für die Berechnung.

- **Erhöhte Sicherheit**

Ein hoher Garlic-Wert zwingt das System dazu, größere Speicherbereiche zu nutzen. Hierdurch verlangsamt die Berechnung erheblich und macht brute-force-Angriffe schwieriger. Für die Implementierung wird  $\text{GARLIC} = 22$  festgelegt, was zu einer deutlichen Verlangsamung führt, so dass die Berechnung selbst auf leistungsfähigen Systemen mehrere Sekunden dauert.

- **Warum Garlic anpassen?**

Je nach verfügbarem Arbeitsspeicher des Zielsystems kann Garlic erhöht werden, um den Schutz zu maximieren oder zu verringern, falls die Berechnung beschleunigt werden soll.

Der Lambda-Wert bestimmt die Anzahl der Iterationen, die für die Passwortableitung verwendet werden.

- **Erhöhte Komplexität**

Ein höherer Lambda-Wert bedeutet, dass Catena das Passwort mehrmals iterativ verarbeitet. Dies erhöht die Zeit, die ein Angreifer für jeden brute-force-Versuch benötigt.

- **LAMBDA = 2**

In der Implementierung ist  $\text{LAMBDA} = 2$  definiert, was eine moderate Balance zwischen Sicherheit und Berechnungszeit darstellt. Je nach Sicherheitsanforderungen kann dieser Wert angepasst werden, um die Schwierigkeit für brute-force-Angreifer weiter zu erhöhen.

### 4.2.2. Catena-Butterfly

Die Butterfly-Struktur, die bei Catena verwendet wird, schafft eine zusätzliche Sicherheitsebene durch die Art und Weise, wie Speicherpositionen in der Berechnung miteinander verbunden werden. Die Butterfly-Variante verlangsamt die Berechnung aufgrund der vielen nicht-sequenziellen Speicherzugriffe. Dies macht den Angriff komplexer, da Angreifer gezwungen werden, die Berechnungsschritte für jeden Speicherschritt nachzubilden.

In diesem Projekt wurde Butterfly gewählt, da

1. Höhere Sicherheit durch komplexe Speicherzugriffe, die Butterfly-Struktur sorgt für eine komplexe Verbindung der Daten im Speicher. Dies führt dazu, dass jeder Berechnungsschritt auf mehreren Speicherwerten basiert, was brute-force-Angriffe verlangsamt.
2. Einmalige Berechnung beim Start, da die Berechnung nur einmalig beim Start durchgeführt wird, ist es akzeptabel, höhere Garlic- und Lambda-Werte zu verwenden, ohne die Systemperformance während der Nutzung zu beeinträchtigen. Die Berechnung dauert dann etwa 15 Sekunden bei einem typischen 16GB RAM und einem i7-Prozessor.

### 4.2.3. Salt-Wert: Verhinderung von Rainbow-Table-Angriffen

Das Salt ist ein zusätzlicher Wert, der dem Passwort hinzugefügt wird, bevor die eigentliche KDF-Berechnung startet. Dadurch werden Rainbow-Table-Angriffe (vorgefertigte brute-force-Tabellen) verhindert, da das gleiche Passwort bei jedem Benutzer zu einem unterschiedlichen Schlüssel führt, wenn das Salt variiert. Salt in der Implementierung: Ein 16-Byte-Salt (SALT\_LEN = 16) ist hartkodiert in

---

```
1 const uint8_t salt[SALT_LEN] = {  
2     0x53, 0x61, 0x66, 0x65, 0x42, 0x61, 0x79, 0x49,  
3     0x73, 0x54, 0x68, 0x65, 0x42, 0x65, 0x73, 0x74  
4 };
```

---

*Listing 4.4:* Definition des Salt-Werts in Hexadezimaldarstellung

das repräsentiert den Text SafeBayIsTheBest.

Die Implementierung nutzt eine vorgefertigte Catena-Testdatei. Es wurden nur minimale Änderungen vorgenommen, um die Integration zu erleichtern und die KDF über die Argumentzeile aufzurufen. Das Passwort wird über die Argumente übergeben. Diese Änderung stellt sicher, dass das Passwort direkt als Argument an die Funktion übergeben wird, was den Ablauf automatisiert.

## 4.3. FUSE Implementierung

### Verzeichnisse: SafeBayFS-ENC und SafeBayFS

- **SafeBayFS-ENC**

In diesem Verzeichnis werden die verschlüsselten Daten abgelegt. Jedes Mal, wenn eine Datei geschrieben wird, verarbeitet die `bb_write`-Funktion von FUSE die Daten mit der POET-Verschlüsselung und speichert sie sicher in SafeBayFS-ENC.

- **SafeBayFS**

Dieses Verzeichnis bietet dem Benutzer Zugriff auf die entschlüsselten Dateien. Wenn eine Datei im SafeBayFS gelesen wird, lädt `bb_read` die verschlüsselten Daten aus SafeBayFS-ENC, entschlüsselt sie mithilfe von POET und gibt sie im Klartext an den Benutzer weiter. So bleibt die Trennung zwischen verschlüsselten und entschlüsselten Daten strikt gewahrt und sorgt dafür, dass verschlüsselte Daten nur im SafeBayFS-ENC liegen, während der Benutzer in SafeBayFS direkt und sicher auf die Daten zugreifen kann.

#### 4.3.1. Aufbau des FUSE-basierten Dateisystems

Die Implementierung dieses Dateisystems in FUSE orientiert sich eng an den im Tutorial beschriebenen Schritten und besteht aus verschiedenen Funktionen, die FUSE bei jedem Zugriff auf das Dateisystem ausführt.

Der Ablauf ist wie folgt:

1. **Main-Funktion**

Die Hauptfunktion `fuse_main()` initialisiert das FUSE-Dateisystem und bindet die im Userspace implementierten Funktionen an die Dateioperationen, die das Betriebssystem aufruft.

Registrierung der Dateioperationen: Jede Dateioperation wie `bb_read`, `bb_write`, `bb_open` und `bb_release` wird bei FUSE registriert. Wenn das Betriebssystem Systemaufrufe wie `read()` oder `write()` sendet, ruft FUSE die entsprechenden Funktionen im Userspace auf.

2. **Mounten des Dateisystems**

Beim Start wird das FUSE-Dateisystem auf dem SafeBayFS-Verzeichnis gemountet. Dadurch werden alle Dateioperationen, die im SafeBayFS ausgeführt werden, durch FUSE geleitet und mit den entsprechenden Funktionen verarbeitet. Somit wird jede Lese- oder Schreib Anforderung zunächst von FUSE abgefangen und durch die verschlüsselnden Funktionen geleitet.

3. Die `bb_read`- und `bb_write`-Funktionen sind zentrale Bestandteile der Implementierung und ermöglichen das verschlüsselte Speichern und das sichere Lesen der Daten.

**bb\_read:** Erfasst die Leseanforderungen und entschlüsselt die in SafeBayFS-ENC gespeicherten verschlüsselten Daten, bevor sie dem Benutzer angezeigt werden. Diese Operationen entsprechen den Systemaufrufen `read()` und `write()` und werden von FUSE transparent ausgeführt.

#### 4.3.2. Verbindung mit der POET-Verschlüsselung

Die Integration der POET-Verschlüsselung in die FUSE-basierten Lese- und Schreiboperationen ist ein zentraler Bestandteil des Projekts. Im Gegensatz zum reinen FUSE-Tutorial, welches lediglich die Grundlagen der Dateisystemerstellung behandelt, bringt die Erweiterung um POET zusätzliche Komplexität durch die Verschlüsselung und Authentifizierung. Jeder Block wird beim Schreiben automatisch verschlüsselt, während beim Lesen die Entschlüsselung erfolgt, bevor die Daten an den Benutzer weitergegeben werden.

- Zusätzliche Schritte für die POET-Integration: In den Funktionen `bb_read` und `bb_write` sind erweiterte Prozesse zur Initialisierung des Verschlüsselungskontextes, zur Verwaltung des Nonce (zufällige Werte zur Sicherheit) sowie zur sicheren Verschlüsselung und Entschlüsselung der Daten implementiert. Hierdurch wird sichergestellt, dass die Datenintegrität und -vertraulichkeit durchgängig gewährleistet sind.

#### 4.3.3. Aufbau und Integration von FUSE: Grundlagen

FUSE stellt eine API bereit, die es ermöglicht, Dateioperationen (wie Lesen und Schreiben) abzufangen und an benutzerdefinierte Funktionen weiterzuleiten. Damit lassen sich Funktionen wie Verschlüsselung, Zugriffsrechte und zusätzliche Logik kontrolliert im Userspace umsetzen.

- Die Initialisierung erfolgt durch `fuse_main(argc, argv, &operations, NULL)`; Hierbei registriert sich FUSE und übergibt die `operations`-Struct, die auf die einzelnen Dateioperationen (z.B. `open`, `read`, `write`, `release`) verweist. Jede Dateioperation wird in einer eigenen Funktion implementiert, welche alle Lese- und Schreib Anforderungen in SafeBayFS steuert.

```

1 static struct fuse_operations bb_oper = {
2     .getattr = bb_getattr,
3     .readdir = bb_readdir,
4     .open = bb_open,
5     .read = bb_read,
6     .write = bb_write,
7     .release = bb_release,
8     .truncate = bb_truncate,
9     // Weitere Funktionen wie rename, mkdir, unlink, usw.
10 };

```

*Listing 4.5: FUSE Operationen in bb\_oper Struct*

- Jeder Eintrag in diesem Struct verweist auf eine spezifische Implementierung der Dateioperation im Dateisystem.
- FUSE übernimmt das Mounting des Dateisystems und ruft für jede Dateisystemoperation die entsprechende Funktion auf. Beispielsweise wird `bb_read` ausgeführt, wenn ein Benutzer eine Datei lesen möchte.

## 4.4. Implementierung von POET in FUSE

### Ausgangslage: Ein Dateisystem mit POET-Verschlüsselung

In dieser Implementierung wird ein FUSE-basiertes Dateisystem verwendet, welches die POET-Verschlüsselung integriert, um die Sicherheit der auf dem Dateisystem gespeicherten Daten zu gewährleisten. Ausgangspunkt für diese Implementierung ist die FUSE-Architektur, die es ermöglicht, Dateisystemoperationen im Userspace zu handhaben.

Dieser Ansatz ermöglicht die Integration der Verschlüsselungsschicht auf transparente Weise, sodass Benutzer das Dateisystem wie jedes andere verwenden können, während ihre Daten im Hintergrund verschlüsselt werden.

Nach dem Mounten des FUSE-basierten Dateisystems können Benutzer auf die Verzeichnisstruktur zugreifen, als wäre es ein gewöhnliches Dateisystem. Es erscheint wie jedes andere Verzeichnis im Dateisystem, aber im Hintergrund werden alle Schreib- und Lesevorgänge über die POET-Verschlüsselungsschicht verarbeitet.

Die Folgenden zwei Abbildungen zeigen einen cat Befehl vor dem Mounten und danach

```

uni@uni-ThinkPad-T490s: ~/CLionProjects/untitled5/rootDir
(base) uni@uni-ThinkPad-T490s: ~/CLionProjects/untitled5/rootDir$ cd rootDir/
(base) uni@uni-ThinkPad-T490s: ~/CLionProjects/untitled5/rootDir$ cat jetzt.txt
#####e2e0fhe#
e3egFae]e4CPfKeeYKÜ#####7ee#
V3X(*NSeeV6geeeHece6a'xh:eIeee6eEa\W#####14e-ee2e8eye\eeDeei2[]zeeeeXeeBee7Dee
e6NeeXQe-eeZleeI#####Bee3ee(
eLeek3o3?;eeVeeee/eca>ceee(zbe!ee4#eeeeeSe-ce#eeee;(eeHreS'
Ie\
e#e5[(HreReee-GeeHeeXl>eeHHeDeBeeq2
1e@IEeckgT.#####be2ee'n'Peese3R'1VG5eeereeeqeeHeeNgGee]#####Jee#
ee
e{eeZ(qZee_yeV*!>>e
eBeeet
te9Ve[]ee2?(VF@'lueleeEe:G]ceeehe.reeee,#6Keeee]T
de'eSeme3Nvet].ee]eev3e)ee2eeKeteL]U5exZD+ee(#####\[-e_uee eF#####eA)x]eeUNq(ebAee#
eeIeeeeEeeeelexUjKeeeeFze(e7egee1Ze75eDeDege3?eLe
:ee#^2ejeb#
3ee#8e->Ie2eeeeHS/eUeGe,
Iee3b7eeCNe)eeHee5Y^be{^ee$9#Hee
xxe-\ AezeZ\jee/CIeeeeIee;eeS#eeS)he%PeeeLeek^" eNTHueN-eeecese'neYeDeef#eeejee+\ eeRee
Qe53HT ]eueeeEyle)eHe(eee[eeHeqeceee4e Teeeeeee_5]S5eJSee!>eeee[edQ['ee]cnBeeAee4h
yePeGeFe_

```

Abbildung 4.1.: Terminalausgabe vor dem Mounten

```

uni@uni-ThinkPad-T490s: ~/CLionProjects/untitled5/mountDir
(base) uni@uni-ThinkPad-T490s: ~/CLionProjects/untitled5/mountDir$ cat jetzt.txt
Hallo, nach dem Mounten sind die Dateien lesbar
(base) uni@uni-ThinkPad-T490s: ~/CLionProjects/untitled5/mountDir$

```

Abbildung 4.2.: Terminalausgabe nach dem Mounten

Beim Mounten wird die FUSE-Logik initialisiert und das Dateisystem wird an einem bestimmten Mountpunkt SafeBayFS verbunden. Dabei registriert FUSE eine Reihe von Funktionen, die auf Systemaufrufe wie `read()`, `write()`, `open()`, `release()` usw. reagieren. In dieser wissenschaftlichen Arbeit werden die Funktionen so implementiert, dass sie POET zur Verschlüsselung und Entschlüsselung der Daten nutzen. Wenn das Dateisystem gemountet ist und der Benutzer beginnt, Dateien hinzuzufügen oder zu bearbeiten, tritt die Verschlüsselung sofort in Kraft.

Die Integration der POET-Verschlüsselung in FUSE stellt sicher, dass die Daten sowohl beim Lesen als auch beim Schreiben verschlüsselt sind.

1. Bei jedem Schreibvorgang in `bb_write` wird der Block, der in die Datei geschrieben wird, verschlüsselt. Jeder Block hat eine eigene Nonce und einen Authentifizierungstag. Dies entspricht den Anforderungen von POET und sorgt für die Sicherheit und Integrität der Daten.
2. In `bb_read` erfolgt die Entschlüsselung der Blöcke. Hierbei werden die Nonce und der Authentifizierungstag überprüft, um sicherzustellen, dass die Daten korrekt und unverändert sind.
3. Da POET für jeden 4096-Byte-Block 32 zusätzliche Bytes (16 Bytes für die Nonce und 16 Bytes für den Tag) hinzufügt, ist die verschlüsselte Datei größer als die entschlüsselte Datei. Dies erfordert spezielle Berechnungen in der FUSE-Implementierung, um die richtigen Offsets zu berechnen und sicherzustellen, dass die Daten korrekt ausgelesen werden.



Indem für jeden Block eine Nonce und ein Authentifizierungstag verwendet werden, wird die Datensicherheit erhöht. Durch die korrekte Berechnung der Blockgrößen und Offsets wird sichergestellt, dass der Client immer die erwarteten Daten erhält, unabhängig davon, ob es sich um einen sequenziellen oder nicht-sequenziellen Zugriff handelt.

Diese Art der Implementierung stellt sicher, dass das Dateisystem nicht nur effizient arbeitet, sondern auch den höchsten Sicherheitsanforderungen genügt.

Der Client ist die Anwendung oder das Programm, das auf die verschlüsselten Dateien zugreifen möchte. Dies kann ein einfacher Befehl wie `cat` sein, der den Inhalt der Datei im Terminal anzeigt, aber auch komplexere Anwendungen wie ein PDF-Viewer oder ein Mediaplayer, die auf die Daten zugreifen und sie in einem lesbaren oder abspielbaren Format darstellen.

#### 4.4.1. POET-Verschlüsselung in der `bb_write`-Funktion

Die Implementierung von POET in ein FUSE-basiertes Dateisystem erfordert eine klare und strukturierte Herangehensweise an die Verschlüsselung und Entschlüsselung von Daten. Dabei gibt es vier zentrale Funktionen in POET, die die Basis des Verschlüsselungsprozesses bilden: `keysetup`, `header_processing`, `encrypt_final` und `decrypt_final`. Diese Funktionen sind verantwortlich für die Schlüsselverwaltung, das Verarbeiten von Metadaten (Header), das Verschlüsseln von Datenblöcken und das anschließende Entschlüsseln, um die ursprünglichen Daten wiederherzustellen. Im Folgenden wird detailliert beschrieben, wie diese Verschlüsselungsmechanismen funktionieren und wie sie speziell in FUSE integriert wurden. Das folgende Listing zeigt eine allgemeine Verwendung von POET.

---

```
1 // Verschlüsselung mit POET
2 keysetup(&ctx, key); // 1. Schlüssel einrichten
3 generate_nonce(&nonce); // 2. Nonce generieren
4 process_header(&ctx, nonce); // 3. Header mit Nonce verarbeiten
5 encrypt_final(&ctx, plaintext, ciphertext, tag); // 4. Verschlüsseln und
    Tag generieren
6
7 // Entschlüsselung mit POET
8 keysetup(&ctx, key); // 1. Schlüssel einrichten
9 process_header(&ctx, nonce); // 2. Header mit Nonce verarbeiten
10 decrypt_final(&ctx, ciphertext, tag, plaintext); // 3. Entschlüsseln und
    Tag prüfen
```

---

*Listing 4.6: POET-Verschlüsselungs- und Entschlüsselungsprozess*

## Die Funktionsweise der `bb_write`-Funktion

Die Hauptaufgabe von `bb_write` besteht darin, Daten, die vom Client an das FUSE-Dateisystem übergeben werden, in Blöcken von 4096 Bytes zu verarbeiten, diese zu verschlüsseln und zusammen mit der zugehörigen Nonce und dem Tag in die Datei zu schreiben. Jede dieser Operationen erfordert mehrere Schritte, die sicherstellen, dass die Integrität und Sicherheit der Daten gewahrt bleibt.

### Schlüssel-Setup(`keysetup`)

Zu Beginn der Funktion wird die Verschlüsselungskontextstruktur (`poet_ctx_t`) initialisiert. Der Schlüssel, der in der `BB_DATA`-Struct gespeichert ist, wird über die Funktion `keysetup` bereitgestellt. Diese Funktion erweitert den 16-Byte-Schlüssel zu einer Reihe von Runden-Schlüsseln, die später für die Verschlüsselung der Datenblöcke verwendet werden.

### Blockweise Verarbeitung

Die Daten werden in Blöcken von 4096 Bytes verarbeitet. Das bedeutet, dass, wenn der Client Daten in die Datei schreibt, die Funktion zuerst den entsprechenden Block ermittelt, in dem der Schreibvorgang beginnen soll. Hierbei wird der Offset in der Datei verwendet, um zu bestimmen, wo der aktuelle Block beginnt und wie viele Daten in diesem Block geschrieben werden müssen.

### Berechnung des Block-Offsets

Der Block wird durch die Berechnung der Blocknummer und des Offsets innerhalb des Blocks bestimmt:

- Diese wird berechnet, indem der Dateioffset durch die Blockgröße (4096 Bytes) geteilt wird. Dies gibt die Position des Blocks in der Datei an.

Die Berechnung des Offsets innerhalb der Datei für die verschlüsselten Daten erfordert jedoch zusätzliche Schritte, da die verschlüsselte Datei aufgrund der Nonce und des Tags größer ist als die unverschlüsselte Datei. Jede Einheit eines verschlüsselten Blocks besteht aus:

- Nonce (16 Bytes)
- Daten (4096 Bytes)
- Tag (16 Bytes)

Das bedeutet, dass jeder verschlüsselte Block eine Größe von 4128 Bytes hat, was bei der Berechnung des verschlüsselten Offsets berücksichtigt werden muss. Das folgende Listing zeigt die oben genannten Funktionsaufrufe.

---

```

1 // Schluessel einrichten
2 keysetup(&ctx, BB_DATA->key);
3
4 // Berechnung der Blocknummer und des Offsets innerhalb des Blocks
5 block_number = offset / BUFFER_SIZE;
6 size_t offset_in_block = offset % BUFFER_SIZE;
7
8 // Berechnung des verschluesselten Offsets
9 encrypted_offset = block_number * (NONCE_LEN + BUFFER_SIZE + TAGLEN);
10
11 log_msg("Blocknummer: %d, Encrypted Offset: %lld\n", block_number, (long
    long)encrypted_offset);
12
13 // Bereite den Puffer fuer die Verschluesselung vor
14 memset(buffer, 0, BUFFER_SIZE);
15 }

```

---

*Listing 4.7: Funktion bb\_write keysetup und Blocknummerberechnung der Daten*

## Nonce-Generierung und Header-Verarbeitung

Für jeden Block, der verschlüsselt wird, muss eine neue Nonce (ein zufälliger Wert) generiert werden, die dann als Teil des Headers in die Datei geschrieben wird. Diese Nonce ist entscheidend für die Sicherheit, da sie garantiert, dass selbst identische Datenblöcke bei jedem Schreibvorgang unterschiedlich verschlüsselt werden.

Die Funktion `process_header` wird verwendet, um die Nonce in den Verschlüsselungskontext einzubinden, sodass sie in den Verschlüsselungsprozess integriert wird. Dies ist ein wichtiger Schritt, da die Nonce die Sicherheit des gesamten Blockes gewährleistet.

---

```

1 // Generiere eine neue Nonce
2 if (getrandom(nonce, NONCE_LEN, 0) != NONCE_LEN) {
3     log_msg("Fehler bei der Nonce-Generierung\n");
4     return -EIO;
5 }
6
7 // Verarbeite die Nonce als Header
8 process_header(&ctx, nonce, NONCE_LEN);

```

---

*Listing 4.8: Generierung und Verarbeitung einer neuen Nonce*

## Padding

In der Implementierung von POET in FUSE war das Hinzufügen von Padding erforderlich, um sicherzustellen, dass jeder Block auf die richtige Größe gebracht wird, insbesondere bei Blöcken, die kleiner als 4096 Bytes sind. Dies hilft dabei, die Daten konsistent zu verschlüsseln und sicherzustellen, dass der letzte Block immer die erwartete Blockgröße hat.

---

```

1 // Wenn die Gesamtgroesse des Puffers kleiner als BUFFER_SIZE ist, fuege
  Padding hinzu
2 if (offset_in_block + size < BUFFER_SIZE) {
3     buffer[offset_in_block + size] = 0x80;
4     memset(buffer + offset_in_block + size + 1, 0, BUFFER_SIZE - (
        offset_in_block + size + 1));
5 }

```

---

*Listing 4.9: Padding-Implementierung*

## Vermeidung des Überschreibens bereits verschlüsselter Daten

Ein kritischer Punkt bei der Implementierung der `bb_write`-Funktion ist die Vermeidung des Überschreibens bereits verschlüsselter Daten. Um dies zu gewährleisten, wird beim Schreiben in einen bestehenden Block der Block zuerst vollständig gelesen, entschlüsselt, mit den neuen Daten aktualisiert und anschließend wieder verschlüsselt. Dieser Prozess stellt sicher, dass keine ungewollten Datenüberschreibungen stattfinden.

---

```

1 // Wenn der Offset innerhalb des Blocks nicht null ist oder die Groesse
  kleiner als BUFFER_SIZE ist,
2 // muessen der bestehende Block gelesen und aktualisieren werden
3 if (offset_in_block != 0 || size < BUFFER_SIZE) {
4     // Es muessen den vorhandenen Block lesen
5     unsigned char existing_nonce[NONCE_LEN];
6     unsigned char existing_tag[TAGLEN];
7     unsigned char existing_encrypted_buffer[BUFFER_SIZE];
8     unsigned char existing_decrypted_buffer[BUFFER_SIZE];
9
10    // Setze den Dateizeiger auf den verschluesselten Offset
11    if (lseek(fi->fh, encrypted_offset, SEEK_SET) == -1) {
12        log_msg("Fehler beim Setzen des Dateizeigers\n");
13        return -EIO;
14    }
15
16    // Lese die Nonce
17    if (read(fi->fh, existing_nonce, NONCE_LEN) == NONCE_LEN) {
18        // Verarbeite die Nonce als Header
19        process_header(&ctx, existing_nonce, NONCE_LEN);
20
21        // Lese den verschluesselten Block
22        ssize_t bytes_read = read(fi->fh, existing_encrypted_buffer,
            BUFFER_SIZE);
23        if (bytes_read != BUFFER_SIZE) {
24            log_msg("Fehler beim Lesen des Ciphertexts\n");
25            return -EIO;
26        }
27

```

---

```

28     // Lese den Tag
29     if (read(fi->fh, existing_tag, TAGLEN) != TAGLEN) {
30         log_msg("Fehler beim Lesen des Tags\n");
31         return -EIO;
32     }
33
34     // Entschlüssele den Block
35     if (decrypt_final(&ctx, existing_encrypted_buffer, bytes_read,
36                     existing_tag, existing_decrypted_buffer) != 0) {
37         log_msg("Fehler beim Entschlüsseln des bestehenden Blocks\n");
38         return -EIO;
39     }
40
41     // Kopiere die vorhandenen Daten in den Puffer
42     memcpy(buffer, existing_decrypted_buffer, BUFFER_SIZE);
43
44     // Verschlüsselungskontext zurücksetzen für den neuen Block
45     keysetup(&ctx, BB_DATA->key);
46 } else {
47     // Wenn kein bestehender Block vorhanden ist, setzen wir den Kontext
48     // zur ck
49     keysetup(&ctx, BB_DATA->key);
50 } else {
51     keysetup(&ctx, BB_DATA->key);
52 }

```

---

*Listing 4.10:* Kein überschreiben, bereits verschlüsselter Blöcke

### Verschlüsselung der Daten (encrypt\_final)

Nachdem die Nonce verarbeitet wurde, werden die tatsächlichen Daten des Blocks verschlüsselt. Dabei werden die Daten mit AES128 im POET-Modus verschlüsselt, und es wird ein 16-Byte-Tag generiert, der später für die Integritätsprüfung verwendet wird. Der verschlüsselte Block sowie der Tag werden dann in die Datei geschrieben.

Wichtig ist hierbei, dass, wenn die zu schreibenden Daten kleiner als 4096 Bytes sind (beispielsweise am Ende der Datei), Padding hinzugefügt wird. Das Padding nach dem ISO/IEC 7816-4 Standard verwendet das Byte 0x80 als erstes Padding-Byte, gefolgt von Nullen. Dies stellt sicher, dass der letzte Block immer die volle Blockgröße von 4096 Bytes erreicht.

---

```

1 // Verschlüssele den Block und generiere den Tag
2 encrypt_final(&ctx, buffer, BUFFER_SIZE, encrypted_buffer, tag);

```

---

*Listing 4.11:* Blockverschlüsselung und Tag-Generierung

## Sequenzielles Schreiben

Die `bb_write`-Funktion verarbeitet die Daten in einer sequenziellen Art und Weise. FUSE arbeitet standardmäßig in 4KB-Blöcken, was bedeutet, dass die Schreibvorgänge immer aufeinanderfolgend in Blöcken von 4096 Bytes stattfinden. Dies ist effizient, da die Blockgröße direkt der natürlichen Blockgröße von FUSE entspricht.

Beim Schreiben eines Blocks wird zuerst der entsprechende Block gelesen, entschlüsselt, dann werden die neuen Daten hinzugefügt, und der gesamte Block wird wieder verschlüsselt und gespeichert.

### 4.4.2. POET-Verschlüsselung in der `bb_read`-Funktion

Wie in `bb_write` wird zu Beginn der `bb_read`-Funktion der POET-Verschlüsselungskontext initialisiert, indem der in der `BB_DATA`-Struktur gespeicherte Schlüssel über `keysetup` verwendet wird. Dies stellt sicher, dass der Entschlüsselungsvorgang mit dem gleichen Schlüssel erfolgt, der ursprünglich zum Verschlüsseln der Daten verwendet wurde. Nach der Schlüsselinitialisierung folgt der Aufruf von `process_header`, der eine vorbereitende Verarbeitung der Nonce durchführt.

In `bb_read` ist es daher entscheidend, die Nonce an der richtigen Stelle einzulesen, da eine inkorrekte Nonce zum Fehlschlagen der Entschlüsselung und damit zu einer unbrauchbaren Ausgabe führen würde, indem die Nonce korrekt verarbeitet wird.

#### Berechnung der Blocknummer und des Offsets

Der Zusammenhang zwischen `size` und `offset` spielt eine entscheidende Rolle in der `bb_read`-Funktion. Der Parameter `offset` gibt an, ab welcher Position innerhalb der Datei der Lesevorgang beginnen soll. Das Dateisystem verarbeitet Daten in 4096-Byte-Blöcken (`BUFFER_SIZE`). Daher berechnet die Funktion zuerst, in welchem Block der Lesevorgang beginnt und wie viele Blöcke gelesen werden müssen. Diese Berechnung erfolgt auf der Basis von `size`, welches angibt, wie viele Bytes der Client lesen möchte und dem `offset`, dass die Startposition markiert.

Da jeder verschlüsselte Block jedoch 4128 Bytes (4096 Bytes Daten + 16 Bytes Nonce + 16 Bytes Tag) umfasst, müssen der tatsächliche Start und die zu lesende Blockanzahl unter Berücksichtigung der 32 zusätzlichen Bytes pro Block berechnet werden.

---

```
1 //  überprüfen , ob der Offset innerhalb der Datei liegt
2 if (offset >= decrypted_file_size) {
3     return 0;
4 }
5
6 // Anpassung der Größe , falls sie über das Dateiende hinausgeht
7 if (offset + size > decrypted_file_size) {
8     size = decrypted_file_size - offset;
9 }
```

```

10
11 // Berechnung der Start- und Endblöcke
12 start_block = offset / BUFFER_SIZE;
13 end_block = (offset + size - 1) / BUFFER_SIZE;
14
15 // Offset im ersten Block
16 offset_in_first_block = offset % BUFFER_SIZE;

```

---

*Listing 4.12:* Überprüfung und Anpassung, der Dateigröße sowie Berechnung der Start- und Endblöcke

### Zusammenhang zwischen size, offset und current\_pos

current\_pos stellt in der bb\_read-Funktion den tatsächlichen Offset in der verschlüsselten Datei dar. Das liegt daran, dass jeder Block in der verschlüsselten Datei größer ist (4128 Bytes) als der Block in der entschlüsselten Datei (4096 Bytes). Das bedeutet, dass der Offset des Clients (bezogen auf die entschlüsselte Datei) in der verschlüsselten Datei angepasst werden muss. Diese Anpassung erfolgt durch die Berechnung von current\_pos, welches den tatsächlichen Leseoffset in der verschlüsselten Datei bestimmt.

---

```

1 // Berechnung von current_pos (verschlüsselter Dateioffset)
2 current_pos = start_block * (NONCE_LEN + BUFFER_SIZE + TAGLEN);

```

---

*Listing 4.13:* Berechnung des verschlüsselten Dateioffsets current\_pos

### Beispiel:

- Angenommen, der Client möchte 8192 Bytes lesen, beginnend bei einem Offset von 4096 Bytes. Dies entspricht zwei 4096-Byte-Blöcken.
- In der verschlüsselten Datei sind diese zwei Blöcke jedoch 4128 Bytes groß. Daher muss current\_pos so berechnet werden, dass der richtige Bereich in der verschlüsselten Datei gelesen wird, inklusive der Nonce und des Tags für jeden Block.

### Problematik der unterschiedlichen Dateigrößen

Ein zentrales Problem bei der Verwendung von POET in FUSE ist, dass die verschlüsselte Datei größer ist als die entschlüsselte Datei. Das liegt daran, dass jeder Block zusätzliche 32 Bytes für die Nonce und den Tag enthält. Das bedeutet:

- Entschlüsselte Datei: Jeder Block ist 4096 Bytes groß.
- Verschlüsselte Datei: Jeder Block ist 4128 Bytes groß (4096 Bytes Daten + 16 Bytes Nonce + 16 Bytes Tag).

Der Client erwartet jedoch, dass die Datei die Größe der verschlüsselten Datei hat und gibt daher size entsprechend der verschlüsselten Dateigröße an. Das Dateisystem muss also sicherstellen, dass trotz der größeren verschlüsselten Datei nur die entschlüsselten Daten an den Client weitergegeben werden.

Durch die Verwendung von `current_pos` wird der tatsächliche Offset in der verschlüsselten Datei berechnet. Die `bb_read`-Funktion liest den verschlüsselten Block (4128 Bytes), entschlüsselt ihn und gibt nur die 4096 Bytes entschlüsselten Daten an den Client zurück. So wird sichergestellt, dass der Client die erwartete Datenmenge erhält, ohne die zusätzlichen Nonce- und Tag-Bytes zu sehen.

### Entschlüsselung der Daten (`decrypt_final`)

Jeder Block wird nun entschlüsselt und dabei der zugehörige Authentifizierungstag (Tag) überprüft. Die Entschlüsselung erfolgt blockweise, wobei der Tag am Ende jedes Blocks validiert wird, um sicherzustellen, dass der Block nicht manipuliert wurde und die Datenintegrität gewährleistet bleibt. Falls die Tag-Überprüfung fehlschlägt, wird dies im Log vermerkt und der Prozess wird mit einer Fehlermeldung abgebrochen. Dieses Verfahren stellt sicher, dass nur authentische und unveränderte Daten entschlüsselt und weiterverarbeitet werden.

---

```
1 // Block entschluesseln und Tag ueberpruefen
2 if (decrypt_final(&ctx, buffer, bytes_read, tag, decrypted_buffer) != 0) {
3     log_msg("Tag-Ueberpruefung_␣fehlgeschlagen\n");
4     return -EIO;
5 }
```

---

*Listing 4.14:* Block entschlüsseln und Tag überprüfen

### Padding entfernen

Um das im `bb_write`-Vorgang hinzugefügte Padding beim Lesen korrekt zu entfernen, wird ein spezielles Muster im entschlüsselten Puffer gesucht. Die Funktion durchsucht die entschlüsselten Daten nach dem Padding. Ein einzelnes `0x80`-Byte, gefolgt von einer Folge von `0x00`-Bytes. Sobald dieses Muster am Ende des Datenblocks gefunden wurde, wird das Padding entfernt, um die Daten in ihrer ursprünglichen, unverfälschten Form auszugeben.

---

```
1 for (ssize_t j = bytes_read - 7; j >= 0; j--) {
2     if (decrypted_buffer[j] == 0x80 &&
3         decrypted_buffer[j + 1] == 0x00 &&
4         decrypted_buffer[j + 2] == 0x00 &&
5         decrypted_buffer[j + 3] == 0x00 &&
6         decrypted_buffer[j + 4] == 0x00 &&
7         decrypted_buffer[j + 5] == 0x00 &&
8         decrypted_buffer[j + 6] == 0x00) {
9
10        log_msg("Padding_␣gefunden_␣an_␣Position_␣%zd,␣entferne_␣Padding...\\n",
11                j);
12        padding_position = j;
13        padding_found = 1;
14        break;
15    }
```

---



### Vermeidung von fehlerhaften Datenzugriffen

Wie bei der `bb_write`-Funktion wird auch bei `bb_read` darauf geachtet, dass keine fehlerhaften oder überschriebenen Daten verarbeitet werden. Das Schreiben und Lesen von Daten in Blöcken mit separaten Nonces und Tags sorgt dafür, dass jeder Block unabhängig verarbeitet werden kann, ohne dass bereits gespeicherte verschlüsselte Daten beeinflusst werden.

Das Sequenzdiagramm soll die Zusammenarbeit zwischen FUSE und POET noch mal verdeutlichen

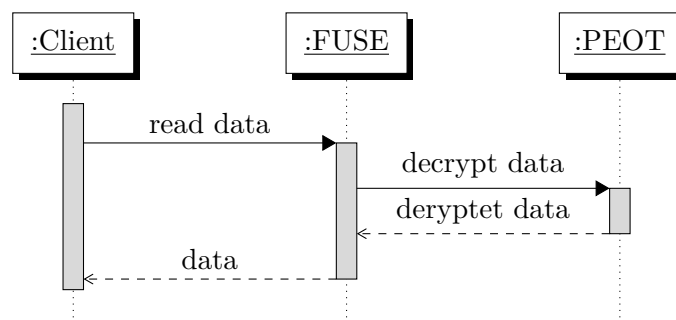


Abbildung 4.3.: FUSE und POET

### 4.4.3. Implementierung der main mit Schutzmassnahmen

Die Hauptfunktion hat mehrere Abschnitte, die spezifische Sicherheitsüberprüfungen und initialisierende Maßnahmen umfassen:

#### 1. Argumentprüfung und Eingabebeschränkungen:

- a) `argc < 3` und `safe_strlen` werden verwendet, um sicherzustellen, dass eine ausreichende Anzahl an Argumenten übergeben wird und dass deren Länge sicher bleibt. Das Ziel ist, Buffer Overflows und ungültige Argumente zu verhindern, die sonst dazu führen könnten, dass das Programm unerwartete Operationen durchführt oder abstürzt.
- b) `safe_strlen` ist eine sichere Implementierung, die sicherstellt, dass die Länge der Argumente niemals den definierten Puffer überschreitet (`MAX_ARG_LENGTH`). Falls die Länge zu groß ist, wird das Programm mit einer Fehlermeldung abgebrochen.

#### 2. Sicheres Kopieren der Argumente:

`strncpy` kopiert die Argumente `SafeBayFS-ENC` und `SafeBayFS` den mount Point in lokale Puffer. Die Verwendung von `strncpy` und das Hinzufügen eines Nullterminators (`\0`) stellen sicher, dass die Zielpuffer nicht überschrieben werden.

### 3. Überprüfung des Mountpoints:

Die Funktion `is_directory_empty` stellt sicher, dass das angegebene Verzeichnis für das Mounten leer ist. Diese Überprüfung verhindert das Mounten über existierenden Dateien, was sonst zu Datenverlusten führen könnte und das Verhalten des Systems beeinträchtigen kann.

### 4. Speicherallokierung und -initialisierung für `bb_data`:

Der `calloc`-Aufruf für `bb_data` initialisiert den Speicher auf Null, um unerwartetes Verhalten aufgrund nicht initialisierter Daten zu verhindern. Uninitialisierter Speicher kann dazu führen, dass zufällige Daten oder sogar sensible Informationen im Speicher verbleiben, die Angreifer möglicherweise auslesen könnten.

### 5. Überprüfung der Benutzerberechtigungen:

- a) Die Überprüfung von `getuid` und `geteuid` stellt sicher, dass das Programm nicht als root-Benutzer gestartet wird.

### 6. Verwaltung und Schutz des Schlüssels:

- a) `sodium_init()` initialisiert die Libsodium-Bibliothek und stellt sicher, dass alle kryptografischen Funktionen korrekt vorbereitet sind. Dieser Schritt ist notwendig, um die weiteren Funktionen für sichere Speicherverwaltung und Schlüsselhandhabung nutzen zu können.
- b) `sodium_malloc` dient zum sicheren Schutz von Speicherbereichen für sensible Daten.
- c) `(sodium_mlock)` und die Einstellung von nur Lesezugriff `sodium_mlock` sperrt den angegebenen Speicherbereich im RAM und verhindert dessen Auslagerung auf die Festplatte.
- d) `sodium_mprotect_readonly` setzt einen Speicherbereich auf nur-Lesezugriff[17].

#### 7. Eingabe des Schlüssels und Längenüberprüfung:

Die Eingabe des Schlüssels erfolgt zeichenweise, um sicherzustellen, dass die Länge 32 Zeichen beträgt (`KEY_LEN_HEX`). Eine zusätzliche Überprüfung entfernt das Zeilenumbruchzeichen, falls es vorhanden ist und überprüft, ob nur hexadezimale Zeichen eingegeben wurden. Dadurch wird, vor Formatierungsangriffen und unerwarteten Eingaben, die das Programmverhalten beeinflussen könnten, geschützt.

#### 8. Eingabe des Schlüssels und Längenüberprüfung:

Die Eingabe des Schlüssels erfolgt zeichenweise, um sicherzustellen, dass die Länge 32 Zeichen beträgt (`KEY_LEN_HEX`). Eine zusätzliche Überprüfung entfernt das Zeilenumbruchzeichen, falls es vorhanden ist, und überprüft, ob nur hexadezimale Zeichen eingegeben wurden. Dies schützt vor Formatierungsangriffen und unerwarteten Eingaben, die das Programmverhalten beeinflussen könnten.

#### 9. Speicherbereinigung und Schutz nach der Eingabe:

Nach der Verarbeitung wird der Speicher mit `sodium_memzero` gelöscht und der Zugriff auf `noaccess` gesetzt (`sodium_mprotect_noaccess`). Dadurch wird verhindert, dass unautorisierte Prozesse auf den Speicher zugreifen oder dass der Speicher für zukünftige Operationen wiederverwendet wird.

### 4.4.4. Hilfsfunktionen

#### 1. `is_directory_empty`

- a) Prüft, ob das Verzeichnis leer ist. Es wird verwendet, um sicherzustellen, dass der Mountpoint für das FUSE-Dateisystem frei von Dateien oder anderen Inhalten ist.
- b) **Sicherheitsvorteil:** Verhindert das Überschreiben vorhandener Dateien, was Datenverlust und potenziellen unautorisierten als weiteren Schutz gegen ungewollten Zugriff auf das Dateisystem.

#### 2. `is_valid_string`

- a) Überprüft, ob eine Zeichenkette nur alphabetische Zeichen enthält, um sicherzustellen, dass bestimmte Eingaben keine unerwünschten Sonderzeichen oder Zahlen enthalten, was für den Schutz vor Code Injection und Fehlformatierungen nützlich ist.

#### 3. `safe_strlen`

Ermittelt die Länge einer Zeichenkette bis zu einer bestimmten maximalen Länge (`max_len`). Diese Implementierung schützt gegen nicht terminierte Zeichenketten und verhindert, dass der Speicher außerhalb der erwarteten Grenzen durchsucht wird. Durch die Einschränkung der maximalen Länge schützt `safe_strlen` vor Buffer Overflows und Out-of-Bounds-Lesezugriffen.

#### 4. `is_hex_string`

Diese Funktion prüft, ob ein gegebener Schlüssel nur hexadezimale Zeichen enthält. Der Schlüssel muss die genaue Länge (`KEY_LEN_HEX`) und das passende Format aufweisen. Durch die Überprüfung der Eingabe wird sichergestellt, dass nur gültige Schlüssel verarbeitet werden

#### 5. `cleanup_and_exit`

- a) stellt eine sichere und geordnete Freigabe aller Ressourcen sicher, die während der Programmausführung reserviert wurden, darunter der Speicher für den Schlüssel und die Logdatei.
- b) Durch die Verwendung von `sodium_memzero`, `sodium_mprotect_readwrite`, und `sodium_free` verhindert, dass sensible Daten wie der Schlüssel im Speicher verbleiben und ausgelesen werden könnten. Damit wird verhindert, dass potenziell sensible Informationen nach Programmende im Speicher verfügbar bleiben.

### 4.4.5. Kompilierung, Flags und Debugging

#### Kompilierung und Flags

##### 1. `-fstack-protector-all`

Aktiviert Stack-Canaries, um Pufferüberläufe auf dem Stack zu erkennen. Wenn ein Überlauf den Stack-Canary überschreibt, beendet das Programm sich selbst mit einem Fehler. Dies ist eine wichtige Schutzmaßnahme gegen Stack-basierte Buffer Overflow-Angriffe.

Stack-basierte Überläufe sind eine häufige Schwachstelle. Stack-Canaries bieten einen zuverlässigen Schutz, indem sie Manipulationen im Stack erkennen

##### 2. `-fPIC (Position Independent Code)`

Erzwingt die Erstellung von positionsunabhängigem Code. Damit können die Binaries in zufällige Speicherbereiche geladen werden, ohne dass Adressen fest in den Code eingebaut werden müssen.

Positionsunabhängiger Code ist ein Kernelement von Address Space Layout Randomization (ASLR), das Angreifen erschwert, die Speicheradressen im Programm vorherzusagen. Dies reduziert die Gefahr von Return-Oriented Programming (ROP)-Angriffen

##### 3. `-fPIE (Position Independent Executable)`

Diese Erweiterung von `-fPIC` ermöglicht es, Executables zu erstellen, die ASLR vollständig nutzen können. Hiermit wird sichergestellt, dass das gesamte Programm in zufälligen Speicherbereichen geladen wird.

In Kombination mit `-fPIC` und ASLR schützt dies vor Memory Corruption Exploits, da

die Speicheradressen von Programmkomponenten für Angreifer unvorhersehbar sind.

#### 4. -O0 -ggdb3 (Optimierungen deaktivieren und Debugging-Informationen hinzufügen)

Das Flag -O0 deaktiviert alle Compiler-Optimierungen, um sicherzustellen, dass der generierte Code so nah wie möglich am Quellcode bleibt. -ggdb3 fügt umfassende Debugging-Informationen hinzu, die GDB im Debugging-Prozess helfen.

Dieses Sequenzdiagramm zeigt den gesamten Ablauf und das Starten von SafeBayFS

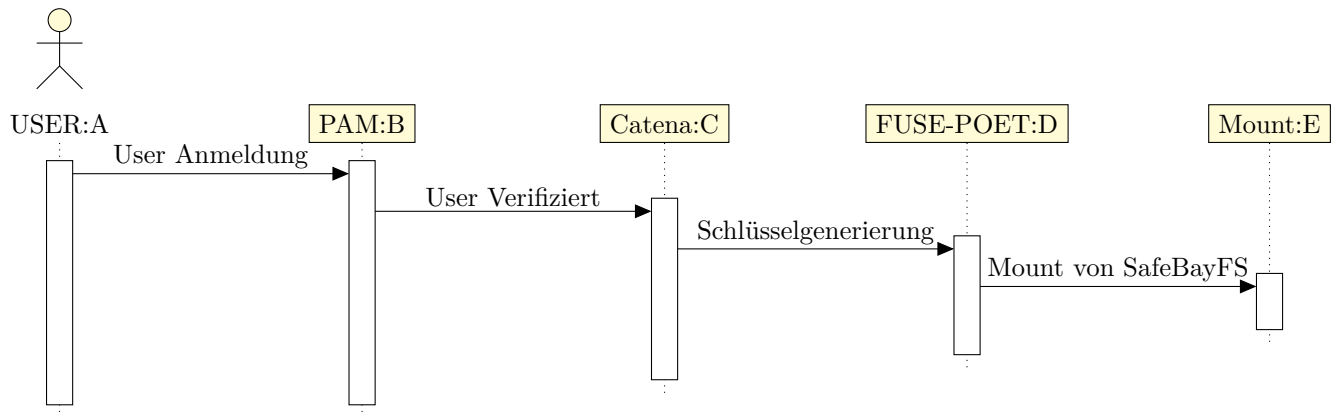


Abbildung 4.4.: Ablauf von SafeBayFS

## **AddressSanitizer (ASan) für Debugging und Sicherheit**

AddressSanitizer (ASan) ist ein leistungsfähiges Werkzeug, das zur Laufzeit Speicherfehler erkennt und in diesem Projekt für den Debugging-Prozess aktiviert war. ASan bietet mehrere wesentliche Vorteile. Erstens ermöglicht es die Erkennung von Speicherzugriffsfehlern, indem es Buffer Overflows, Use-After-Free, Stack-Buffer-Overflows und Heap-Buffer-Overflows identifiziert und meldet. Diese Fehler werden sofort entdeckt, wodurch das Programm abstürzt und eine detaillierte Fehlermeldung ausgibt. Ohne ASan könnten viele Speicherfehler unentdeckt bleiben und nur sporadisch auftreten, was die Fehlerbehebung erheblich erschwert. ASan identifiziert solche Fehler jedoch sofort, zeigt die betroffene Speicheradresse an und markiert den fehlerhaften Codeabschnitt.

Ein weiterer Vorteil von ASan liegt in der Prävention von Use-After-Free und Double-Free-Fehlern. ASan markiert den Speicher nach einem free-Aufruf als ungültig und stellt sicher, dass kein erneuter Zugriff erfolgt. Auch mehrfaches Freigeben desselben Speicherblocks wird erkannt. Dadurch wird unvorhersehbares Verhalten und das Entstehen potenzieller Sicherheitslücken verhindert, die bei mehrfacher Freigabe oder Zugriffen auf bereits freigegebenen Speicher entstehen könnten.

ASan bietet zudem erweiterte Fehlermeldungen und einen Stack-Trace, wodurch die genaue Speicheradresse und eine Backtrace-Anzeige des Fehlers bereitgestellt werden. Diese detaillierten Meldungen sind äußerst wertvoll, um die Fehlerursache zu identifizieren und den betroffenen Speicherbereich ausfindig zu machen. Sie unterstützen Entwickler dabei, den fehlerhaften Code schnell zu lokalisieren und zu beheben.

Schließlich verbessert ASan die Code-Sicherheit während der Entwicklung erheblich. Es erfasst Fehler, die möglicherweise in regulären Tests nicht auffallen und macht den Code robuster gegenüber potenziellen Sicherheitslücken. Besonders für Programme, die sicherheitskritische Daten verarbeiten, wie in diesem Projekt beim Umgang mit kryptografischen Schlüsseln, ist ASan eine wertvolle Hilfe. Durch das Aufdecken versteckter Speicherprobleme trägt ASan entscheidend zur Erhöhung der Codequalität und -sicherheit bei, da potenziell sicherheitskritische Schwachstellen sichtbar werden[19].

## 5. Ergebnisse

### 5.1. Analyse und Evaluation

Die Sicherheit des Systems wurde umfassend gegen verschiedene Bedrohungsszenarien bewertet, insbesondere gegen brute-force-Angriffe und die Integrität der verschlüsselten Daten. Es wurde auch auf Sicherheitsmaßnahmen gegen Exploitation unternommen.

#### Sicherheitsmaßnahmen gegen Exploitation

1. Buffer Overflow-Schutz:

Die Kombination aus `safe_strlen`, `strncpy` und festgelegten Puffergrößen (`MAX_ARG_LENGTH`, `KEY_LEN_HEX`) verhindert, dass Eingaben den Speicher überschreiten und dadurch unerwartete Verhaltensweisen verursachen.

2. Speicherschutz und Bereinigung: Die Nutzung von `sodium_malloc` und Libsodium-Funktionen für die Schlüsselverwaltung stellt sicher, dass der kryptografische Schlüssel gegen unerwünschten Zugriff geschützt ist. Speichersperren und Berechtigungen (`sodium_mprotect_noaccess`, `sodium_mprotect_readonly`) garantieren, dass der Schlüssel nur dann zugänglich ist, wenn es unbedingt erforderlich ist.

3. Keine Ausführung als Root:

Die Überprüfung der Benutzerberechtigung stellt sicher, dass das Programm nicht mit Root-Rechten ausgeführt wird, was das Risiko von schwerwiegenden Sicherheitslücken reduziert.

4. Format- und Längenprüfung:

Jede Benutzereingabe, insbesondere der Schlüssel, wird auf Länge und Format geprüft. Dies schützt vor Format-String-Schwachstellen und manipulierten Eingaben, die den Programmfluss ändern oder interne Datenstrukturen beschädigen könnten.

5. Starke Schlüssel und Schutz gegen brute-force-Angriffe

- a) Durch die Verwendung der Catena-Butterfly-Variante in Verbindung mit den Parametern `Garlic` und `Lambda` ist das System gegen brute-force-Angriffe geschützt. Die Parameter `Garlic = 22` und `Lambda = 2` sorgen für einen stark speicherintensiven und zeitaufwendigen Schlüsselableitungsprozess, der auf einem durchschnittlichen Rechner mit normaler Hardware (z. B. einem typischen Intel i7 und 16GB RAM) etwa 15

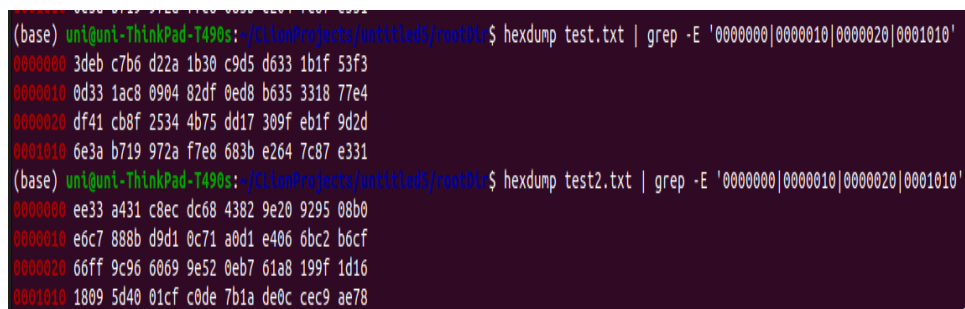
Sekunden für die Generierung eines einzigen Passwort-Hashes benötigt.

Beispielberechnung für brute-force:

Um die Effektivität dieses Schutzes zu veranschaulichen, betrachten wir eine brute-force-Attacke, bei der ein Angreifer 1.000.000 Passwortkombinationen testen möchte:

- i. Wenn die Ableitung eines Passwort-Hashes 15 Sekunden dauert, würde die Berechnung von 1.000.000 Passwörtern auf derselben Hardware ca. 173 Tage benötigen.
  - ii. Selbst bei leistungsstärkeren Maschinen würde die enorme Rechenzeit das Durchführen einer solchen Attacke stark erschweren, wenn nicht sogar völlig ineffektiv machen.
6. Die hohe Berechnungslast und die speicherintensive Struktur der Catena-Butterfly-Funktion stellen sicher, dass brute-force-Angriffe fast aussichtslos sind. Zusätzlich bewirkt der hohe Speicherbedarf, dass selbst moderne Hochleistungsrechner, die parallel arbeiten, nur begrenzt effektiver sind, da die gesamte Funktion für jeden Versuch Speicher intensiv nachbilden muss.
7. Einzigartige Verschlüsselung jedes Blocks:

Eine weitere Stärke des Systems liegt in der Struktur der Blockverschlüsselung. Jeder Datenblock wird mithilfe eines einzigartigen Nonce-Werts und einem Schlüssel verschlüsselt, sodass jeder Block eine einzigartige, nicht wiederverwendbare Verschlüsselung erhält. Dies schützt die Daten vor sogenannten Replay-Angriffen und verhindert, dass ein Angreifer bekannte Muster im verschlüsselten Text identifizieren kann. Dies kann im Hexdump der verschlüsselten Dateien nachvollzogen werden, in dem jeder Block unterschiedliche verschlüsselte Werte aufweist.



```
(base) uni@uni-ThinkPad-T490s:~/CiliumProjects/untitled5/rootDir$ hexdump test.txt | grep -E '00000000|0000010|0000020|0001010'
00000000 3deb c7b6 d22a 1b30 c9d5 d633 1b1f 53f3
00000100 0d33 1ac8 0904 02df 0ed8 b635 3318 77e4
00000200 df41 cb8f 2534 4b75 dd17 309f eb1f 9d2d
00010100 6e3a b719 972a f7e8 683b e264 7c87 e331
(base) uni@uni-ThinkPad-T490s:~/CiliumProjects/untitled5/rootDir$ hexdump test2.txt | grep -E '00000000|0000010|0000020|0001010'
00000000 ee33 a431 c8ec dc68 4382 9e20 9295 08b0
00000100 e6c7 888b d9d1 0c71 a0d1 e406 6bc2 b6cf
00000200 66ff 9c96 6069 9e52 0eb7 61a8 199f 1d16
00010100 1809 5d40 01cf c0de 7b1a de0c cec9 ae78
```

Abbildung 5.1.: Hexdump von zwei Dateien mit identischen Inhalt

Die Hexdumps von verschlüsselten Dateien zeigen den Anfang und das Ende der zwei Dateien. Es ist gut erkennbar, dass jeder Block durch den Einsatz des Nonce eine eigene Verschlüsselung erhält. Jeder Block wird durch den Nonce, den durch POET generierten Tag und die verschlüsselten Daten einzigartig gemacht. Der Screenshot bestätigt, dass keine zwei Blöcke die gleichen Werte haben, selbst wenn der Inhalt identisch ist.



Die Analyse zeigt, dass die Kombination von Catena für die sichere Schlüsselableitung und die POET-Verschlüsselung für eine blockweise, einzigartige Verschlüsselung der Daten erhebliche Sicherheitsvorteile bietet. Die Implementierung stellt sicher, dass Angriffe auf das Dateisystem aufgrund des starken Schlüssels und der individuellen Blockverschlüsselung aussichtslos und ineffektiv sind.

### **5.1.1. Tests und Validierung der korrekten Funktionsweise**

Um die Stabilität und korrekte Funktionalität des FUSE-basierten, verschlüsselten Dateisystems zu gewährleisten, wurde das System umfassend mit verschiedenen Dateitypen und Größen getestet. Die Tests zielten darauf ab, sicherzustellen, dass alle typischen Dateien korrekt verarbeitet werden, die Verschlüsselung und Entschlüsselung wie vorgesehen funktionieren und dass die Benutzerfreundlichkeit auch bei größeren Dateien erhalten bleibt.

#### **1. Dateitypen**

Das System wurde mit einer Vielzahl an Dateiformaten getestet, die in verschiedenen Anwendungsbereichen gängig sind. Hierzu zählen:

- a) Textdateien: .txt, .md, .docx und .odt
- b) Bilddateien: .jpg, .png, .bmp, .gif
- c) Videoformate: .mp4, .avi, .mkv
- d) PDF-Dateien: .pdf
- e) Weitere Dokumenttypen: .xlsx, .pptx, .odg

Alle Dateitypen konnten erfolgreich gespeichert und wiederhergestellt werden, wobei die Verschlüsselung und Entschlüsselung im Hintergrund korrekt und nahtlos abliefen.

#### **2. Dateigrößen**

Die Tests wurden mit Dateien unterschiedlicher Größe durchgeführt, um die Auswirkungen der Verschlüsselung auf die Lade- und Speichergeschwindigkeit bei kleinen und großen Dateien zu untersuchen:

- a) Kleine Dateien (< 10 MB): Textdateien und kleinere Bilder öffneten sich nahezu sofort ohne spürbare Verzögerungen. Sowohl die Lade- als auch die Speichergeschwindigkeit blieb für den Benutzer unmerklich schnell.
- b) Mittlere Dateien (10 - 500 MB): PDF-Dokumente, Präsentationen und einige Videodateien in mittlerer Größe zeigten eine minimale Verzögerung beim Öffnen und Speichern, die jedoch für den Benutzer leicht wahrnehmbar war.
- c) Große Dateien (> 500 MB): Besonders große Dateien wie Videos oder umfangreiche PDF- und Office-Dokumente (z. B. umfangreiche Reports und Präsentationen) erforderten beim Öffnen und Speichern eine merkbare Verzögerung. Beispielsweise

mussten Benutzer bei Filmen (z.B. .mp4 oder .mkv mit mehr als 1 GB) einige Sekunden warten, bis das Video geladen ist. In manchen Fällen müsste der Benutzer erneut auf „Play“ drücken, um das Video zu starten, da die Initialisierung etwas Zeit in Anspruch nimmt.

### **3. Benutzererfahrung bei typischen Dateigrößen und -typen**

Insgesamt verliefen die Tests erfolgreich. Das System konnte mit gängigen Dateiformaten und -größen zuverlässig umgehen. Während kleinere Dateien nahezu ohne Verzögerung bearbeitet werden konnten, kam es bei sehr großen Dateien zu einer Wartezeit, die jedoch angesichts der Sicherheitsvorteile und des speicherintensiven Verschlüsselungsprozesses akzeptabel ist.

Die Tests bestätigen, dass das FUSE-basierte Verschlüsselungssystem eine breite Palette an Dateitypen und -größen unterstützt und dabei eine stabile und sichere Funktionalität für die Benutzer bietet. Die POET-Verschlüsselung bewährte sich als zuverlässig und bewältigte sowohl kleinere Alltagsdateien als auch größere Medien- und Office-Dateien mit geringen Einschränkungen in der Benutzerfreundlichkeit.

Im Folgenden werden Tabellen präsentiert, die die Schreib-Performanceunterschiede zwischen einem normalen System ohne FUSE und einem System mit FUSE verdeutlichen. In der ersten Tabelle werden die Messergebnisse des Systems ohne FUSE dargestellt, die als Referenz dienen. Die zweite Tabelle zeigt die Ergebnisse eines Systems mit FUSE und verdeutlicht bereits einen signifikanten Performanceverlust im Vergleich zur ersten Tabelle. Ein Blick auf die dritte Tabelle verdeutlicht zudem, wie stark die Performance insbesondere beim Schreiben der Daten leidet, wenn FUSE und POET eingesetzt werden. Dieser Vergleich zeigt klar die Auswirkungen der FUSE-Integration auf die Systemleistung und stellt dar, in welchen Bereichen der Overhead besonders spürbar ist.

<b>Performance der Schreibgeschwindigkeit ohne FUSE</b>				
<b>Typ</b>	<b>Format</b>	<b>Größe</b>	<b>Geschwindigkeit</b>	<b>Zeit</b>
Bild	JPG	1,8 MB	434 MB/s	0,004s
Text	TXT	5 MB	775 MB/S	0,006s
PDF	PDF	26 MB	1,4 GB/s	0,018s
Video	MP4	1,8 GB	2,4 GB/s	0,829s

*Tabelle 5.1.:* Performance der Schreibgeschwindigkeit Ubuntu ohne FUSE

<b>Performance der Schreibgeschwindigkeit mit FUSE</b>				
<b>Typ</b>	<b>Format</b>	<b>Größe</b>	<b>Geschwindigkeit</b>	<b>Zeit</b>
Bild	JPG	1,8 MB	19,4 MB/s	0,01s
Text	TXT	5 MB	49.9 MB/S	0,10s
PDF	PDF	26 MB	146 MB/s	0,20s
Video	MP4	1,8 GB	157 MB/s	12,40s

*Tabelle 5.2.:* Performance der Schreibgeschwindigkeit Ubuntu mit FUSE

<b>Performance der Schreibgeschwindigkeit mit SafeBayFS</b>				
<b>Typ</b>	<b>Format</b>	<b>Größe</b>	<b>Geschwindigkeit</b>	<b>Zeit</b>
Bild	JPG	1,8 MB	2,2 MB/s	0,86s
Text	TXT	5 MB	6 MB/S	0,9s
PDF	PDF	26 MB	5,8 MB	4,6s
Video	MP4	1,8 GB	7.7 MB/s	253,04s

*Tabelle 5.3.:* Performance der Schreibgeschwindigkeit Ubuntu mit SafeBayFS

Ein Blick auf die Tabellen der Lese-Performance verdeutlicht, dass FUSE im Leseverhalten eine bessere Performance bietet als beim Schreiben. Dennoch zeigt sich, dass es auch bei der read-Funktion von SafeBay noch Verbesserungspotenzial gibt. Insbesondere bei großen Dateien und Leseoperationen könnten Optimierungen die Effizienz weiter steigern und den bestehenden Overhead reduzieren

<b>Performance der Lesegeschwindigkeit ohne FUSE</b>				
<b>Typ</b>	<b>Format</b>	<b>Größe</b>	<b>Geschwindigkeit</b>	<b>Zeit</b>
Bild	JPG	1,8 MB	793 MB/s	0,002s
Text	TXT	5 MB	299 MB/S	0,017s
PDF	PDF	26 MB	554 GB/s	0,048s
Video	MP4	1,8 GB	7,5 GB/s	0,259s

*Tabelle 5.4.:* Performance Lesegeschwindigkeit Ubuntu ohne FUSE

<b>Performance der Lesegeschwindigkeit mit FUSE</b>				
<b>Typ</b>	<b>Format</b>	<b>Größe</b>	<b>Geschwindigkeit</b>	<b>Zeit</b>
Bild	JPG	1,8 MB	326 MB/s	0,005s
Text	TXT	5 MB	424 MB/S	0,012s
PDF	PDF	26 MB	367 MB/s	0,073s
Video	MP4	1,8 GB	2,9 GB MB/s	0,671s

*Tabelle 5.5.:* Performance Lesegeschwindigkeit Ubuntu mit FUSE

<b>Performance der Lesegeschwindigkeit mit SafeBayFS</b>				
<b>Typ</b>	<b>Format</b>	<b>Größe</b>	<b>Geschwindigkeit</b>	<b>Zeit</b>
Bild	JPG	1,8 MB	13,8 MB/s	0,13s
Text	TXT	5 MB	10,8 MB/S	0,470s
PDF	PDF	26 MB	11,2 MB	2,399s
Video	MP4	1,8 GB	11,2 MB/s	174,075s

*Tabelle 5.6.:* Performance Lesegeschwindigkeit Ubuntu mit SafeBayFS

## 6. Zusammenfassung und Ausblick

### 6.1. Zusammenfassung der Ergebnisse

In diesem Projekt wurde ein FUSE-basiertes Dateisystem erfolgreich entwickelt und um eine sichere POET-Verschlüsselung erweitert. Ziel war es, eine Lösung zu schaffen, die eine transparente, benutzerfreundliche und gleichzeitig sichere Verschlüsselung und Speicherung von Dateien ermöglicht, ohne dass der Benutzer manuelle Verschlüsselungsschritte ausführen muss.

#### **Erreichte Ziele und Ergebnisse:**

##### **1. Sicherheitsziele**

Durch die Integration der Catena-Butterfly-KDF und POET konnte ein robuster Schutz gegen brute-force-Angriffe realisiert werden. Die starke Schlüsselableitung sorgt dafür, dass die Ableitung eines einzigen Passwort-Hashes etwa 15 Sekunden benötigt und brute-force-Versuche praktisch unwirksam macht. Die Verwendung eines einzigartigen Nonce pro Block gewährleistet, dass jeder verschlüsselte Block individuell und manipulationssicher ist. Die Blockverschlüsselung und der Authentifizierungs-Tag schützen die Daten zusätzlich vor Replay-Angriffen und Manipulationen, was für die Integrität der gespeicherten Dateien entscheidend ist.

##### **2. Benutzerfreundlichkeit und Performance**

Die Implementierung ermöglicht es Benutzern, wie gewohnt mit Dateien zu arbeiten, ohne die Verschlüsselung direkt wahrzunehmen. Das System bietet fast verzögerungsfreie Lese- und Schreibvorgänge für Dateien bis 500 MB und sorgt dafür, dass auch größere Dateien zuverlässig gespeichert und entschlüsselt werden. Typische Dateigrößen in Formaten wie .txt, .jpg, .pdf, .mp4 und .docx konnten ohne spürbare Leistungseinbußen verarbeitet werden. Bei sehr großen Mediendateien wie Filmen kam es zu einer leichten Verzögerung, die angesichts des Umfangs der Verschlüsselung jedoch im Rahmen bleibt.

### 3. Kompatibilität und Dateisystemintegration

Die erfolgreiche Umsetzung eines benutzerfreundlichen FUSE-Dateisystems, das mit einer Vielzahl an Dateitypen umgehen kann, verdeutlicht, dass das System eine solide Grundlage für den alltäglichen Einsatz in unterschiedlichen Anwendungen bietet. Der Aufbau des FUSE-Dateisystems in Verbindung mit SafebayFS-ENC und SafebayFS ermöglicht eine klare Trennung zwischen verschlüsselten und entschlüsselten Daten und unterstützt die Sicherheit sowie die Benutzerfreundlichkeit.

Fast alle definierten Ziele aus der Einleitung wurden erreicht und das System bietet eine sichere und transparente Verschlüsselung im Userspace, ist gegen typische Angriffsszenarien geschützt und für den Benutzer intuitiv bedienbar.

Diese Ergebnisse belegen die Robustheit und Praxistauglichkeit des entwickelten FUSE-basierten Verschlüsselungssystems und legen eine solide Basis für den Einsatz in sicherheitskritischen Umgebungen.

Ein Ziel wurde nicht erreicht, nämlich die Bereitstellung eines fertigen Debian-Pakets, um es dem Benutzer noch einfacher zu machen, das System zu installieren und zu starten.

## 6.2. Wesentliche Erkenntnisse

Die Entwicklung und Implementierung des FUSE-basierten, verschlüsselten Dateisystems in Verbindung mit der POET-Verschlüsselung und Catena als KDF hat entscheidende Erkenntnisse sowohl über die Sicherheitsvorteile als auch die praktischen Herausforderungen des Systems geliefert. Eine strenge, objektive Bewertung der Ergebnisse zeigt einige Stärken, aber auch Bereiche, die potenziell verbessert werden könnten.

### **Sicherheitsstärke und Beständigkeit gegen brute-force-Angriffe**

Die Wahl von Catena-Butterfly als KDF zur sicheren Schlüsselableitung hat sich als äußerst robust erwiesen. Die Kombination aus speicherintensiven Berechnungen und den Werten für Garlic und Lambda macht brute-force-Angriffe praktisch undurchführbar, selbst für leistungsstarke Rechner. Dies zeigt, dass das System eine hervorragende Sicherheit bietet und gegen Passwort-basierte Angriffe resistent ist. Allerdings könnte die komplexe Berechnung der KDF bei schwächerer Hardware zu leichten Einschränkungen bei der Benutzerfreundlichkeit führen, was den breiten Einsatz des Systems im Blick behalten muss.

### **Leistungsanalyse: Auswirkung der Verschlüsselung auf die Systemperformance**

Ein klarer Kompromiss wurde zwischen Sicherheit und Performance erzielt. Für typische Dateigrößen (bis 500 MB) ermöglicht das System eine flüssige und benutzerfreundliche Erfahrung. Bei größeren Dateien, wie sie häufig in Medienanwendungen auftreten, kam es jedoch zu spürbaren Verzögerungen. Das zeigt, dass das System zwar robust ist, aber bei hohem Datenvolumen die Performance-Einbußen spürbar werden. Die Lese- und Schreibgeschwindigkeiten könnten

für den praktischen Einsatz optimiert werden, ohne dabei die Sicherheit zu gefährden, indem beispielsweise die Algorithmen in Zukunft für bestimmte Anwendungsfälle und Dateigrößen optimiert werden.

### **Architektur der FUSE-Integration: Trennung von SafeBayFS-ENC und SafeBayFS**

Die Trennung zwischen SafeBayFS-ENC und SafeBayFS hat sich als zentraler Baustein des Systems erwiesen und zeigt, dass die sichere und physische Speicherung der verschlüsselten Daten funktioniert. Die Entkopplung von verschlüsseltem und entschlüsseltem Speicherzugriff stellt sicher, dass sensible Daten niemals im Klartext auf dem Datenträger abgelegt werden. Allerdings bringt die Wahl von FUSE als Framework gewisse Einschränkungen in der Performance und Flexibilität, insbesondere bei der Verarbeitung größerer Dateien, was in zukünftigen Versionen möglicherweise durch andere Lösungen im Kernel-space adressiert werden könnte.

### **Kompatibilität und Testbarkeit: Vielfalt an Dateitypen und Anwendungsfällen**

Die Tests zeigten, dass das System mit einer Vielzahl an Dateitypen umgehen kann. Es hat sich jedoch herausgestellt, dass bei Dateiformaten mit sehr großen Strukturen (z.B. große Videos oder sehr umfangreiche PDF-Dateien) die Speicher- und Ladezeiten beeinträchtigt werden. Ein zusätzliches Augenmerk auf die Optimierung der Speicherverwaltung und eine verstärkte Anpassung der Blockgrößen an den jeweiligen Dateityp könnten in Zukunft sinnvoll sein, um die Ladezeiten weiter zu verringern und die Benutzerfreundlichkeit zu steigern.

### **Verbesserungspotenzial in der Benutzerfreundlichkeit**

Obwohl das System fast verzögerungsfreie Operationen für kleinere und mittlere Dateien ermöglicht, kann die Nutzererfahrung bei größeren Dateien verbessert werden. Ein optimiertes Puffermanagement oder eine adaptive Blockgröße wären mögliche Ansätze, um die Lade- und Speicherzeiten zu verringern, ohne Abstriche bei der Sicherheit zu machen. Zusätzlich wäre eine benutzerseitige Einstellung der Performance-Parameter wie *Garlic* und *Lambda* je nach Leistungsanforderungen und Sicherheit eine wertvolle Ergänzung, um dem Benutzer die Kontrolle über die Balance zwischen Sicherheit und Performance zu geben.

Die Analyse zeigt, dass das entwickelte FUSE-basierte Verschlüsselungssystem ein hohes Maß an Sicherheit und eine breite Unterstützung für verschiedene Dateitypen bietet. Die Performance ist für typische Anwendungen akzeptabel, jedoch ist eine weitere Optimierung für speicherintensive Anwendungen empfehlenswert. In zukünftigen Versionen könnten gezielte Verbesserungen in der Performance und Benutzerfreundlichkeit die Anwendungsmöglichkeiten dieses Systems erweitern und dessen Effektivität noch weiter steigern.

### 6.3. Fazit

Die entwickelte Lösung stellt ein sicheres und funktionales FUSE-basiertes Dateisystem dar, dass Dank der POET-Verschlüsselung und der Catena-KDF hohe Sicherheit gegen brute-force- und Replay-Angriffe bietet. Durch die Implementierung dieser kryptografischen Maßnahmen und der transparenten Integration in das Dateisystem wird ein stabiles System für die verschlüsselte Datenspeicherung bereitgestellt, welches dem Benutzer eine intuitive Bedienung ermöglicht. Eine kritische Betrachtung zeigt jedoch, dass trotz der erzielten Erfolge bestimmte Schwächen vorhanden sind und Optimierungspotenzial besteht.

Die Lösung erreicht ein hohes Maß an Sicherheit durch die Verwendung starker Schlüssel und individueller Blockverschlüsselungen mit einem einzigartigen Nonce pro Block. Die Catena-KDF schützt das System gegen brute-force-Angriffe, indem der hohe Garlic- und Lambda-Wert die Berechnung jedes Passwort-Hashes erheblich verlangsamt. Auch die Implementierung der FUSE-Architektur, die eine klare Trennung zwischen SafeBayFS-ENC (verschlüsselte Daten) und SafeBayFS (entschlüsselte Daten) ermöglicht, hat sich bewährt.

Dadurch wird gewährleistet, dass vertrauliche Informationen nur verschlüsselt auf dem Datenträger liegen und im Klartext nur für den authentifizierten Benutzer in SafeBayFS verfügbar sind. Die Kombination aus Sicherheit und Benutzerfreundlichkeit macht das System für Alltagsanwendungen geeignet, bei denen der Schutz sensibler Daten im Vordergrund steht.

Eine bemerkbare Schwäche ist die Performance bei größeren Dateien, die in Form von längeren Ladezeiten bei sehr großen Medien- oder Dokumentdateien sichtbar wird. Die aktuelle Blockstruktur, die Nonce und Authentifizierungs-Tag in ihrer bisherigen Form integriert, führt zu einem Blockgrößen-Overhead, der bei jeder Lese- und Schreiboperation Performance kostet.

Eine vielversprechende Optimierung wäre die Anpassung der Blockstruktur auf exakt 4096 Byte, wobei die verschlüsselten Daten, Nonce und Tag inklusive, auf diese Blockgröße optimiert werden. Eine solche Anpassung würde die Performance verbessern, da die Lese- und Schreibzugriffe besser an die Speichereinheiten der Hardware angepasst wären. Ein Blockformat, dass die zusätzlichen Sicherheitsmerkmale in einer festen Größe integriert, könnte eine spürbare Reduzierung der Lade- und Speichergeschwindigkeit zur Folge haben, da die Ein- und Ausgabeoperationen effizienter verarbeitet würden.

Die Lösung bietet eine starke Sicherheit und gute Benutzerfreundlichkeit für typische Anwendungen, weist jedoch Performance-Einschränkungen auf, die insbesondere bei speicherintensiven Dateien bemerkbar werden. Eine strukturierte Anpassung der Blockgröße, die Nonce und Tag in einer festen Größe integriert, wäre ein nächster logischer Schritt, um die Performance zu verbessern, ohne die Sicherheit zu gefährden. Die entwickelte Lösung zeigt sich damit als robuste Basis.



### 6.3.1. Ausblick

#### Mögliche Erweiterungen

##### 1. Unterstützung weiterer Verschlüsselungsalgorithmen

Die aktuelle Implementierung basiert vollständig auf POET als AEAD. Zukünftige Arbeiten könnten eine flexible Architektur entwickeln, die es ermöglicht, verschiedene Verschlüsselungsalgorithmen zu unterstützen, wie AES-GCM, ChaCha20-Poly1305 oder XChaCha20. Durch eine modulare Struktur könnte der Benutzer den Verschlüsselungsalgorithmus je nach Anwendungsfall und Sicherheitsanforderungen auswählen.

Das würde die Flexibilität des Systems erhöhen und es an eine breitere Palette von Einsatzszenarien anpassen.

##### 2. Performance-Optimierungen durch Blockstruktur-Anpassungen

Eine wesentliche Möglichkeit zur Verbesserung der Performance liegt in der Anpassung der Blockstruktur. Aktuell werden Nonce und Tag zu den verschlüsselten Daten hinzugefügt, was zu einem leichten Overhead führt. Eine Anpassung, die die Blockgröße einschließlich Nonce und Tag exakt auf 4096 Byte festlegt, könnte die Performance bei großen Dateien erheblich steigern. Das würde die Lese- und Schreibgeschwindigkeit verbessern und das System für speicherintensive Anwendungen optimieren. Außerdem könnten fortgeschrittene Speicherstrategien, wie Verschlüsselungsprozesse, erforscht werden, um die Wartezeiten beim Schreiben großer Dateien zu reduzieren.

##### 3. Erweiterung auf andere Betriebssysteme

Derzeit ist die Implementierung auf Linux beschränkt, da FUSE stark im Linux-Ökosystem verwurzelt ist. Zukünftige Arbeiten könnten untersuchen, wie das System auf andere Betriebssysteme, wie macOS oder Windows, übertragen werden kann. Für macOS existiert ein FUSE-kompatibles System (macFUSE), das eine Implementierung vergleichbarer Dateisysteme ermöglicht. Für Windows könnte eine ähnliche Lösung entwickelt werden, um die Verfügbarkeit der verschlüsselten Speicherlösung über Betriebssystemgrenzen hinweg zu erweitern.

##### 4. Integration von Schlüsselverwaltungssystemen

Um das System in professionellen Umgebungen noch sicherer und einfacher in der Handhabung zu gestalten, könnte eine Integration mit Schlüsselverwaltungssystemen (KMS) wie HashiCorp Vault, AWS KMS oder Azure Key Vault erfolgen. Das würde es ermöglichen, kryptografische Schlüssel zentral zu speichern und zu verwalten, was den administrativen Aufwand für Unternehmen reduzieren würde. Durch die Kombination der FUSE-Architektur mit einem KMS könnten zentrale Richtlinien zur Schlüsselrotation und -sicherung umgesetzt werden.

## 5. Erweiterung auf Cloud-Umgebungen

Eine spannende Erweiterung könnte die Unterstützung für Cloud-basierte Speicherlösungen sein, bei denen die verschlüsselten Daten nicht lokal, sondern in einer Cloud-Umgebung abgelegt werden. Durch die Kombination von FUSE und POET mit einer Cloud-Anbindung könnten Daten sicher in Cloud-Diensten wie Amazon S3, Google Cloud Storage oder Azure Blob Storage gespeichert werden. Diese Erweiterung würde den Anwendungsbereich des Systems deutlich erweitern und die Sicherheit vertraulicher Daten in Cloud-Umgebungen verbessern.

Diese Vorschläge eröffnen vielversprechende Ansätze für zukünftige Arbeiten, die das FUSE-POET-basierte Verschlüsselungssystem in puncto Performance, Flexibilität und Benutzerfreundlichkeit weiter verbessern könnten. Die Implementierung zusätzlicher Verschlüsselungsoptionen, die Optimierung der Blockstruktur und eine erweiterte Kompatibilität auf mehreren Plattformen und Umgebungen würden den praktischen Nutzen und die Leistungsfähigkeit des Systems erheblich steigern und das Einsatzspektrum in sicherheitskritischen Anwendungen erweitern.

### 6.3.2. Integration und Zusammenarbeit

Die Implementierung des FUSE-basierten, verschlüsselten Dateisystems steht als Open-Source-Projekt auf GitLab zur Verfügung, um es der breiten Entwickler- und Sicherheits-Community zugänglich zu machen. Die Bereitstellung des Quellcodes ermöglicht es Entwicklern und IT-Sicherheitsforschern, das System frei zu nutzen, zu untersuchen und an ihre spezifischen Anforderungen anzupassen. Die offene Struktur bietet eine ideale Grundlage für die Zusammenarbeit und Integration mit anderen Sicherheitslösungen und fördert den Wissensaustausch in der Open-Source-Community.

#### **Zusammenarbeit und Open-Source-Community**

Alle Bestandteile dieses Projekts sind vollständig quelloffen und stehen der Open-Source-Community zur Verfügung. Die Veröffentlichung auf GitLab fördert die Zusammenarbeit und ermöglicht es anderen Entwicklern, den Code zu erweitern, Fehler zu melden und Verbesserungsvorschläge einzubringen. Es ist ein zentrales Anliegen dieses Projekts, neue Ideen und Anregungen von Entwicklern und Sicherheitsforschern weltweit aufzunehmen. Durch die Open-Source-Bereitstellung sind alle Interessierten eingeladen, das System weiterzuentwickeln und Sicherheitslösungen beizusteuern, die den Funktionsumfang und die Effizienz des Projekts erhöhen.

#### **Offenheit für Verbesserungen und Vorschläge**

Das Projekt ist sehr offen für konstruktives Feedback und Verbesserungsvorschläge aus der Community. Jeder Vorschlag, der dazu beitragen kann, die Sicherheit, die Benutzerfreundlichkeit oder die Performance des Systems zu steigern, ist herzlich willkommen. Die Bereitschaft zur Zusammenarbeit schafft eine dynamische Grundlage für die Weiterentwicklung und den fortlaufenden Fortschritt des Projekts.

Der Austausch mit anderen Entwicklern und IT-Sicherheitsfachleuten bietet dabei eine wertvolle Gelegenheit, das System kontinuierlich zu optimieren und den steigenden Sicherheitsanforderungen anzupassen.

Durch die Veröffentlichung als Open-Source und die aktive Zusammenarbeit mit der Community leistet dieses Projekt einen Beitrag zur Weiterentwicklung sicherer Dateisysteme und stärkt den offenen Austausch in der IT-Sicherheitsbranche. Die Bereitschaft zur Integration und Kooperation schafft eine ideale Plattform, um das Projekt gemeinsam mit der Community weiterzuentwickeln und langfristig zu verbessern.

# Literaturverzeichnis

- [1] Jean-Philippe Aumasson. *Serious Cryptography: A Practical Introduction to Modern Encryption*. No Starch Press, San Francisco, 2017. <https://nostarch.com/seriouscrypto>.
- [2] Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications, September 2021. <https://datatracker.ietf.org/doc/rfc9106/>, Letzter Zugriff: 10.11.2024.
- [3] Bundesamt für Sicherheit in der Informationstechnik (BSI). BSI Technische Richtlinie TR-02102-1: Kryptographische Verfahren: Empfehlungen und Schlüssellängen, Version 2024-01. Technical report, Bundesamt für Sicherheit in der Informationstechnik, 2024. [https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?\\_\\_blob=publicationFile&v=10](https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile&v=10), Letzter Zugriff: 16.10.2024.
- [4] Libfuse Community. Sshfs: Ssh filesystem, 2024. <https://github.com/libfuse/sshfs>, Letzter Zugriff: 09.11.2024.
- [5] S3FS-FUSE Community. S3fs: Fuse-based file system backed by amazon s3, 2024. <https://github.com/s3fs-fuse/s3fs-fuse>, Letzter Zugriff: 11.11.2024.
- [6] Richard Davis Elaine Barker, Lily Chen. NIST Special Publication 800-56C Revision 2: Recommendation for Key-Derivation Methods in Key-Establishment Schemes. Technical report, National Institute of Standards and Technology (NIST), 2020. <https://doi.org/10.6028/NIST.SP.800-56Cr2>, Letzter Zugriff: 21.09.2024.
- [7] John Foley Christian Forler Eik List Stefan Lucks David McGrew Jakob Wenzel Farzaneh Abed, Scott Fluhrer. The poet family of on-line authenticated encryption schemes, 2014. <http://https://competitions.cr.yp.to/round2/poetv20.pdf>, Letzter Zugriff: 15.9.2024.
- [8] Christian Forler, Stefan Lucks, and Jakob Wenzel. The Catena Password-Scrambling Framework, 2015. <http://www.uni-weimar.de/fileadmin/user/fak/medien/professuren/Mediensicherheit/Research/Publications/catena-v3.3.pdf>, Letzter Zugriff: 28.09.2024.
- [9] Gluster Community. Gluster documentation, 2024. <https://docs.gluster.org/en/latest/>, Letzter Zugriff: 19.9.2024.

- [10] Valient Gough. Encfs: Encrypted file system, 2024. <https://github.com/vgough/encfs>, Letzter Zugriff: 06.11.2024.
- [11] IBM. PAM authentication on UNIX and Linux, 2021. <https://www.ibm.com/docs/de/netcoolomnibus/8.1?topic=authentication-pam-unix-linux>, Letzter Zugriff: 2024-09-13.
- [12] Mohamud Ahmed Jimale, Muhammad Reza Z'aba, Miss Laiha Binti Mat Kiah, Mohd Yamani Idna Idris, Norziana Jamil, Moesfa Soheila Mohamad, and Mohd Saufy Rohmad. Authenticated encryption schemes: A systematic review. *IEEE Access*, 10:14739–14766, 2022. <https://ieeexplore.ieee.org/document/9695453>, Letzter Zugriff: 09.10.2024.
- [13] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, Boca Raton, FL, USA, 3rd edition edition, 2021. <https://www.crcpress.com/Chapman--HallCRC-Cryptography-and-Network-Security-Series/book-series/CHCRYNETSEC>.
- [14] Mustafa Khairallah. *Hardware Oriented Authenticated Encryption Based on Tweakeable Block Ciphers*. Springer, 2022. <https://link.springer.com/book/10.1007/978-981-16-6344-4>.
- [15] James Lembke, Pierre-Louis Roman, and Patrick Eugster. Defuse: An interface for fast and correct user space file system access. *ACM Transactions on Storage (TOS)*, 18(3):1–29, 2022. <https://doi.org/10.1145/3558792>, Letzter Zugriff: 11.10.2024.
- [16] libfuse Community. python-fuse: Python bindings for fuse, 2023. <https://github.com/libfuse/python-fuse>, Letzter Zugriff: 09.10.2024.
- [17] Libsodium Project. Libsodium Documentation: Memory Management, 2024. [https://doc.libsodium.org/memory\\_management](https://doc.libsodium.org/memory_management), Letzter Zugriff: 07.11.2024.
- [18] David A. McGrew. An interface and algorithms for authenticated encryption. Request for Comments (RFC) 5116, January 2008. <https://datatracker.ietf.org/doc/html/rfc5116>, Letzter Zugriff: 09.10.2020.
- [19] Microsoft. *AddressSanitizer (ASan) für C++*, 2024. <https://learn.microsoft.com/de-de/cpp/sanitizers/asan?view=msvc-170>, Letzter Zugriff: 17.10.2024.
- [20] Han-Wen Nienhuys. go-fuse: Fuse bindings for go, 2023. <https://github.com/hanwen/go-fuse>, Letzter Zugriff: 19.10.2024.
- [21] Christof Paar, Jan Pelzl, and Tim Güneysu. *Understanding Cryptography: From Established Symmetric and Asymmetric Ciphers to Post-Quantum Algorithms*. Springer-Verlag GmbH, Berlin, Germany, 2nd edition edition, 2024. <https://doi.org/10.1007/978-3-662-69007-9>.

- [22] David Pfeiffer. *Writing a FUSE Filesystem*, 2007. <https://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/>, Letzter Zugriff: 27.10.2024.
- [23] Rogério Pontes, Dorian Burihabwa, Francisco Maia, João Paulo, Valerio Schiavoni, Pascal Felber, Hugues Mercier, and Rui Oliveira. SafeFS: A Modular Architecture for Secure User-Space File Systems. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–12. ACM, 2017. <https://repositorio.inesctec.pt/server/api/core/bitstreams/85db7293-c524-40cc-8fc8-fdb0701ce377/content>, Letzter Zugriff: 04.09.2024.
- [24] puniverse Community. `javafs`: Java bindings for fuse, 2023. <https://github.com/puniverse/javafs>, Letzter Zugriff: 05.10.2024.
- [25] PyPI. `python-pam`: Python module for PAM authentication, 2024. <https://pypi.org/project/python-pam/>, Letzter Zugriff: 13.10.2024.
- [26] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213, 2010. <http://www.csl.sri.com/users/gehani/papers/SAC-2010.FUSE.pdf>, Letzter Zugriff: 02.10.2024.
- [27] William Stallings. *Cryptography and Network Security: Principles and Practice*. Pearson Education Limited, Harlow, UK, 8th global edition edition, 2023. <https://www.pearsonglobaleditions.com>.
- [28] ubuntuusers.de. PAM - Pluggable Authentication Modules, 2024. <https://wiki.ubuntuusers.de/PAM/>, Letzter Zugriff: 17.10.2024.
- [29] Rob van der Wijngaart and Fred Oh. Improving gpu performance by reducing instruction cache misses. *NVIDIA Technical Blog*, August 2024. Verfügbar unter: <https://developer.nvidia.com/blog/improving-gpu-performance-by-reducing-instruction-cache-misses-2/>, Letzter Zugriff: 29.10.2024.
- [30] Bharath Kumar Reddy Vangoor. To fuse or not to fuse? analysis and performance characterization of the fuse user-space file system framework. Master’s thesis, Stony Brook University, 2016. URL: [https://repo.library.stonybrook.edu/xmlui/bitstream/handle/11401/77254/Vangoor\\_grad.sunysb\\_0771M\\_13181.pdf?sequence=1](https://repo.library.stonybrook.edu/xmlui/bitstream/handle/11401/77254/Vangoor_grad.sunysb_0771M_13181.pdf?sequence=1), Letzter Zugriff: 24.10.2024.
- [31] Bert Wesarg, Niels Vangoor, and Hubertus Franke. Filesystem in userspace (fuse): A performance analysis and optimizations. Technical report, LibFUSE Project, 2017. <https://github.com/libfuse/libfuse/blob/master/doc/fast17-vangoor.pdf>, Letzter Zugriff: 01.11.2024.

## A. Sourcecode von bb\_write und bb\_read

---

```
1 int bb_write(const char *path, const char *buf, size_t size, off_t offset,
2             struct fuse_file_info *fi) {
3     poet_ctx_t ctx;
4     unsigned char buffer[BUFFER_SIZE];
5     unsigned char encrypted_buffer[BUFFER_SIZE];
6     unsigned char tag[TAGLEN];
7     unsigned char nonce[NONCE_LEN];
8     off_t encrypted_offset;
9     int block_number;
10
11     log_msg("\nbb_write(path=\"%s\", buf=0x%08x, size=%zu, offset=%lld, fi=0x
12             %08x)\n",
13             path, buf, size, offset, fi);
14
15     // Schlüssel einrichten
16     keysetup(&ctx, BB_DATA->key);
17
18     // Berechnung der Blocknummer und des Offsets innerhalb des Blocks
19     block_number = offset / BUFFER_SIZE;
20     size_t offset_in_block = offset % BUFFER_SIZE;
21
22     // Berechnung des verschlüsselten Offsets
23     encrypted_offset = block_number * (NONCE_LEN + BUFFER_SIZE + TAGLEN);
24
25     log_msg("Blocknummer: %d, Encrypted Offset: %lld\n", block_number, (long
26             long)encrypted_offset);
27
28     // Berechne den Puffer für die Verschlüsselung vor
29     memset(buffer, 0, BUFFER_SIZE);
30
31     // Wenn der Offset innerhalb des Blocks nicht null ist oder die Größe
32     // kleiner als BUFFER_SIZE ist,
33     // müssen der bestehende Block gelesen und aktualisiert werden
34     if (offset_in_block != 0 || size < BUFFER_SIZE) {
35         // Es müssen den vorhandenen Block lesen
36         unsigned char existing_nonce[NONCE_LEN];
37         unsigned char existing_tag[TAGLEN];
38         unsigned char existing_encrypted_buffer[BUFFER_SIZE];
39         unsigned char existing_decrypted_buffer[BUFFER_SIZE];
```

```

36
37 // Setze den Dateizeiger auf den verschlüsselten Offset
38 if (lseek(fi->fh, encrypted_offset, SEEK_SET) == -1) {
39     log_msg("Fehler beim Setzen des Dateizeigers\n");
40     return -EIO;
41 }
42
43 // Lese die Nonce
44 if (read(fi->fh, existing_nonce, NONCE_LEN) == NONCE_LEN) {
45     // Verarbeite die Nonce als Header
46     process_header(&ctx, existing_nonce, NONCE_LEN);
47
48     // Lese den verschlüsselten Block
49     ssize_t bytes_read = read(fi->fh, existing_encrypted_buffer,
50                               BUFFER_SIZE);
51     if (bytes_read != BUFFER_SIZE) {
52         log_msg("Fehler beim Lesen des Ciphertexts\n");
53         return -EIO;
54     }
55
56     // Lese den Tag
57     if (read(fi->fh, existing_tag, TAGLEN) != TAGLEN) {
58         log_msg("Fehler beim Lesen des Tags\n");
59         return -EIO;
60     }
61
62     // Entschlüssele den Block
63     if (decrypt_final(&ctx, existing_encrypted_buffer, bytes_read,
64                      existing_tag, existing_decrypted_buffer) != 0) {
65         log_msg("Fehler beim Entschlüsseln des bestehenden Blocks\n");
66         return -EIO;
67     }
68
69     // Kopiere die vorhandenen Daten in den Puffer
70     memcpy(buffer, existing_decrypted_buffer, BUFFER_SIZE);
71
72     // Verschlüsselungskontext zur cksetzen für den neuen Block
73     keysetup(&ctx, BB_DATA->key);
74 } else {
75     // Wenn kein bestehender Block vorhanden ist, setzen wir den
76     // Kontext zur ck
77     keysetup(&ctx, BB_DATA->key);
78 } else {
79     // Verschlüsselungskontext zur cksetzen für den neuen Block
80     keysetup(&ctx, BB_DATA->key);
81 }

```



```

81 // Kopiere die neuen Daten in den Puffer
82 memcpy(buffer + offset_in_block, buf, size);
83
84 // Wenn die Gesamtgröße des Puffers kleiner als BUFFER_SIZE ist, füge
   Padding hinzu
85 if (offset_in_block + size < BUFFER_SIZE) {
86     buffer[offset_in_block + size] = 0x80; // ISO/IEC 7816-4 Padding
87     memset(buffer + offset_in_block + size + 1, 0, BUFFER_SIZE - (
        offset_in_block + size + 1));
88 }
89
90 // Generiere eine neue Nonce
91 if (getrandom(nonce, NONCE_LEN, 0) != NONCE_LEN) {
92     log_msg("Fehler bei der Nonce-Generierung\n");
93     return -EIO;
94 }
95
96 // Verarbeite die Nonce als Header
97 process_header(&ctx, nonce, NONCE_LEN);
98
99 // Verschlüssele den Block und generiere den Tag
100 encrypt_final(&ctx, buffer, BUFFER_SIZE, encrypted_buffer, tag);
101
102 // Setze den Dateizeiger erneut auf den verschlüsselten Offset
103 if (lseek(fi->fh, encrypted_offset, SEEK_SET) == -1) {
104     log_msg("Fehler beim Setzen des Dateizeigers vor dem Schreiben\n");
105     return -EIO;
106 }
107
108 // Schreibe die Nonce
109 if (write(fi->fh, nonce, NONCE_LEN) != NONCE_LEN) {
110     log_msg("Fehler beim Schreiben der Nonce\n");
111     return -EIO;
112 }
113
114 // Schreibe den verschlüsselten Block
115 if (write(fi->fh, encrypted_buffer, BUFFER_SIZE) != BUFFER_SIZE) {
116     log_msg("Fehler beim Schreiben des verschlüsselten Blocks\n");
117     return -EIO;
118 }
119
120 // Schreibe den Tag
121 if (write(fi->fh, tag, TAGLEN) != TAGLEN) {
122     log_msg("Fehler beim Schreiben des Tags\n");
123     return -EIO;
124 }
125
126 // Gebe die Anzahl der geschriebenen Bytes zurück
127 return size;

```

Listing A.1: Die vollständige bb\_write Funktion

---

```

1 int bb_read(const char *path, char *buf, size_t size, off_t offset, struct
  fuse_file_info *fi) {
2     poet_ctx_t ctx;
3     unsigned char buffer[BUFFER_SIZE];
4     unsigned char decrypted_buffer[BUFFER_SIZE];
5     unsigned char tag[TAGLEN];
6     unsigned char nonce[NONCE_LEN];
7     size_t total_bytes_read = 0;
8     off_t encrypted_file_size;
9     off_t decrypted_file_size;
10    int start_block, end_block;
11    size_t offset_in_first_block;
12    off_t current_pos;
13    int block;
14
15    log_msg("\nbb_read(path=\"%s\", buf=0x%08x, size=%zu, offset=%lld, fi=0x
      %08x)\n",
16            path, buf, size, offset, fi);
17
18    keysetup(&ctx, BB_DATA->key);
19
20    // verschlüsselte Dateigröße
21    encrypted_file_size = lseek(fi->fh, 0, SEEK_END);
22    if (encrypted_file_size == -1) {
23        log_msg("Fehler beim Ermitteln der verschlüsselten Dateigröße\n");
24        return -EIO;
25    }
26
27    //Anzahl der Blöcke
28    int num_blocks = encrypted_file_size / (NONCE_LEN + BUFFER_SIZE + TAGLEN);
29
30    // Berechnung der maximalen entschlüsselten Dateigröße (ohne
      Berücksichtigung des Paddings)
31    decrypted_file_size = num_blocks * BUFFER_SIZE;
32
33    log_msg("Anzahl der Blöcke: %d, Maximale entschlüsselte Dateigröße: %
      lld\n", num_blocks, (long long)decrypted_file_size);
34
35    // überprüfen, ob der Offset innerhalb der Datei liegt
36    if (offset >= decrypted_file_size) {
37        return 0;
38    }
39
40    // Anpassung der Größe, falls sie über das Dateiende hinausgeht

```

```

41     if (offset + size > decrypted_file_size) {
42         size = decrypted_file_size - offset;
43     }
44
45     // Berechnung der Start- und Endblocke
46     start_block = offset / BUFFER_SIZE;
47     end_block = (offset + size - 1) / BUFFER_SIZE;
48
49     // Offset im ersten Block
50     offset_in_first_block = offset % BUFFER_SIZE;
51
52     // Berechnung von current_pos (verschlüsselter Dateioffset)
53     current_pos = start_block * (NONCE_LEN + BUFFER_SIZE + TAGLEN);
54
55     log_msg("Start_Block: %d, End_Block: %d\n", start_block, end_block);
56     log_msg("Current_Position(encrypted_file_offset): %lld\n", (long long)
        current_pos);
57
58     // Setze den Dateizeiger auf current_pos
59     if (lseek(fi->fh, current_pos, SEEK_SET) == -1) {
60         log_msg("Fehler beim Setzen des Dateizeigers\n");
61         return -EIO;
62     }
63
64     // Variable zur Anpassung der entschlüsselten Dateigröße nach
        Entfernung des Paddings
65     int last_block_adjusted = 0;
66
67     // Schleife über die Blöcke
68     for (block = start_block; block <= end_block && total_bytes_read < size;
        block++) {
69         // Lese die Nonce
70         if (read(fi->fh, nonce, NONCE_LEN) != NONCE_LEN) {
71             log_msg("Fehler beim Lesen der Nonce\n");
72             return -EIO;
73         }
74
75         // Verarbeite die Nonce als Header
76         process_header(&ctx, nonce, NONCE_LEN);
77
78         // Lese den verschlüsselten Block (Ciphertext)
79         ssize_t bytes_read = read(fi->fh, buffer, BUFFER_SIZE);
80         if (bytes_read != BUFFER_SIZE) {
81             log_msg("Fehler beim Lesen des Ciphertexts\n");
82             return -EIO;
83         }
84
85         // Lese den Tag
86         if (read(fi->fh, tag, TAGLEN) != TAGLEN) {

```

```

87         log_msg("Fehler beim Lesen des Tags\n");
88         return -EIO;
89     }
90
91     // Block entschlüsseln und Tag überprüfen
92     if (decrypt_final(&ctx, buffer, bytes_read, tag, decrypted_buffer) !=
93         0) {
94         log_msg("Tag-überprüfung fehlgeschlagen\n");
95         return -EIO;
96     }
97
98     // Wenn es der letzte Block der verschlüsselten Datei ist, prüfen
99     // auf Padding
100
101     if (block == num_blocks - 1 && !last_block_adjusted) {
102         log_msg("Letzter Block der Datei erreicht, prüfe auf Padding...\n");
103
104         int padding_found = 0;
105         ssize_t padding_position = -1;
106
107         // Suche nach der Padding
108         for (ssize_t j = bytes_read - 7; j >= 0; j--) {
109             if (decrypted_buffer[j] == 0x80 &&
110                 decrypted_buffer[j + 1] == 0x00 &&
111                 decrypted_buffer[j + 2] == 0x00 &&
112                 decrypted_buffer[j + 3] == 0x00 &&
113                 decrypted_buffer[j + 4] == 0x00 &&
114                 decrypted_buffer[j + 5] == 0x00 &&
115                 decrypted_buffer[j + 6] == 0x00) {
116
117                 log_msg("Padding gefunden an Position %zd, entferne\n",
118                     j);
119                 padding_position = j;
120                 padding_found = 1;
121                 break;
122             }
123         }
124
125         // Wenn Padding gefunden wurde, passe bytes_read und
126         // decrypted_file_size an
127         if (padding_found) {
128             bytes_read = padding_position;
129             decrypted_file_size -= (BUFFER_SIZE - bytes_read);
130             log_msg("Angepasste entschlüsselte Dateigröße: %lld\n", (
131                 long long)decrypted_file_size);
132         }
133
134         last_block_adjusted = 1; // Dateigröße nur einmal anpassen
135     }

```

```

130
131 // Berechne die Anzahl der zu kopierenden Bytes
132 size_t data_offset = 0;
133 size_t data_length = bytes_read;
134
135 // Anpassung f r den ersten Block
136 if (block == start_block) {
137     data_offset = offset_in_first_block;
138     if (data_offset >= bytes_read) {
139         // Keine Daten in diesem Block zu kopieren
140         data_length = 0;
141     } else {
142         data_length -= data_offset;
143     }
144 }
145
146 // Anpassung f r den letzten Lesevorgang
147 if (total_bytes_read + data_length > size) {
148     data_length = size - total_bytes_read;
149 }
150
151 if (data_length > 0) {
152     // Kopiere die entschl sselten Daten in den Puffer
153     memcpy(buf + total_bytes_read, decrypted_buffer + data_offset,
154           data_length);
155     total_bytes_read += data_length;
156 }
157
158 // Aktualisiere current_pos f r den n chsten Block
159 current_pos += NONCE_LEN + BUFFER_SIZE + TAGLEN;
160
161 // Setze den Dateizeiger f r den n chsten Block, falls es weitere
162 // Bl cke gibt
163 if (block < end_block) {
164     if (lseek(fi->fh, current_pos, SEEK_SET) == -1) {
165         log_msg("Fehler beim Setzen des Dateizeigers f r den
166               n chsten Block\n");
167         return -EIO;
168     }
169 }
170
171 // Gebe die Anzahl der gelesenen Bytes zur ck
172 return total_bytes_read;
173 }

```

---

*Listing A.2:* Die vollst ndige bb\_read Funktion

## B. Schreibgeschwindigkeit Screenshots

```
(base) uni@uni-ThinkPad-T490: $ time dd if=~/Dokumente/video-test.mp4 of=~/CLionProje
1887426800 Bytes (1,9 GB, 1,8 GiB) kopiert, 12 s, 157 MB/s
464+1 Datensätze ein
464+1 Datensätze aus
1948880479 Bytes (1,9 GB, 1,8 GiB) kopiert, 12,4056 s, 157 MB/s

real    0m12,407s
user    0m0,004s
(base) uni@uni-ThinkPad-T490: $ time dd if=~/Dokumente/loIvER.txt of=~/CLionProject
1+1 Datensätze ein
1+1 Datensätze aus
5110464 Bytes (5,1 MB, 4,9 MiB) kopiert, 0,00659083 s, 775 MB/s

real    0m0,010s
user    0m0,001s
sys     0m0,009s
(base) uni@uni-ThinkPad-T490: $ time dd if=~/Dokumente/buch-test.pdf of=~/CLionProj
6+1 Datensätze ein
6+1 Datensätze aus
26782802 Bytes (27 MB, 26 MiB) kopiert, 0,0185971 s, 1,4 GB/s

real    0m0,022s
user    0m0,002s
sys     0m0,020s
(base) uni@uni-ThinkPad-T490: $ time dd if=~/Dokumente/bild-test.jpg of=~/CLionProj
8+1 Datensätze ein
8+1 Datensätze aus
```

Abbildung B.1.: Ubuntu Schreibgeschwindigkeit ohne Fuse

```
sys     0m0,020s
(base) uni@uni-ThinkPad-T490: $ time dd if=~/Dokumente/bild-test.jpg of=~/CLionProj
8+1 Datensätze ein
8+1 Datensätze aus
1847928 Bytes (1,8 MB, 1,8 MiB) kopiert, 0,00425709 s, 434 MB/s

real    0m0,007s
user    0m0,000s
sys     0m0,008s
(base) uni@uni-ThinkPad-T490: $ time dd if=~/Dokumente/bild-test.jpg of=~/CLionProj
8+1 Datensätze ein
8+1 Datensätze aus
1847928 Bytes (1,8 MB, 1,8 MiB) kopiert, 0,0950547 s, 19,4 MB/s

real    0m0,100s
user    0m0,002s
sys     0m0,018s
(base) uni@uni-ThinkPad-T490: $ time dd if=~/Dokumente/buch-test.pdf of=~/CLionProj
6+1 Datensätze ein
6+1 Datensätze aus
26782802 Bytes (27 MB, 26 MiB) kopiert, 0,183603 s, 146 MB/s

real    0m0,188s
user    0m0,001s
sys     0m0,034s
(base) uni@uni-ThinkPad-T490: $ time dd if=~/Dokumente/loIvER.txt of=~/CLionProject
1+1 Datensätze ein
1+1 Datensätze aus
```

Abbildung B.2.: Ubuntu Schreibgeschwindigkeit mit FUSE

```

(base) uni@uni-ThinkPad-T490s: $ time dd if=~/Dokumente/btld-test.jpg of=~/CLionProj
0+1 Datensätze ein
0+1 Datensätze aus
1847928 Bytes (1,8 MB, 1,8 MiB) kopiert, 0,856303 s, 2,2 MB/s

real    0m0,861s
user    0m0,002s
sys     0m0,020s
(base) uni@uni-ThinkPad-T490s: $ time dd if=~/Dokumente/buch-test.pdf of=~/CLionProj
25165824 Bytes (25 MB, 24 MiB) kopiert, 4 s, 5,7 MB/s
6+1 Datensätze ein
6+1 Datensätze aus
26782802 Bytes (27 MB, 26 MiB) kopiert, 4,63261 s, 5,8 MB/s

real    0m4,638s
user    0m0,001s
sys     0m0,076s
(base) uni@uni-ThinkPad-T490s: $ time dd if=~/Dokumente/loIVER.txt of=~/CLionProject
1+1 Datensätze ein
1+1 Datensätze aus
5110464 Bytes (5,1 MB, 4,9 MiB) kopiert, 0,856587 s, 6,0 MB/s

real    0m0,861s
user    0m0,002s
sys     0m0,021s
(base) uni@uni-ThinkPad-T490s: $ 

```

Abbildung B.3.: Ubuntu Schreibgeschwindigkeit mit SafeBay

## C. Lesegeschwindigkeit Screenshots

```
unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/bid-test.jpg of=/dev/null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
1048576 Bytes (1,0 MB) kopiert, 0,0023807 s, 793 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.001s  
user 0m0.001s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/tst008-test-of-dev-null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
1048576 Bytes (1,0 MB) kopiert, 0,017886 s, 299 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.021s  
user 0m0.020s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/buch-test.pdf of=/dev/null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
2076382 Bytes (2,0 MB) kopiert, 0,046234 s, 554 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.022s  
user 0m0.021s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/video-test.mpg of=/dev/null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
1048576 Bytes (1,0 MB) kopiert, 0,239974 s, 7,3 GB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.240s  
user 0m0.240s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$
```

Abbildung C.1.: Ubuntu Lesegeschwindigkeit ohne FUSE

```
unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/bid-test.jpg of=/dev/null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
1048576 Bytes (1,0 MB) kopiert, 0,013803 s, 138 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.014s  
user 0m0.013s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/tst008-test-of-dev-null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
1048576 Bytes (1,0 MB) kopiert, 0,024686 s, 267 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.025s  
user 0m0.024s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/buch-test.pdf of=/dev/null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
2076382 Bytes (2,0 MB) kopiert, 0,063302 s, 423 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.063s  
user 0m0.062s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/video-test.mpg of=/dev/null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
1048576 Bytes (1,0 MB) kopiert, 0,472745 s, 2,9 GB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.473s  
user 0m0.472s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$
```

Abbildung C.2.: Ubuntu Lesegeschwindigkeit mit FUSE

```
unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/bid-test.jpg of=/dev/null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
1048576 Bytes (1,0 MB) kopiert, 0,133747 s, 13,8 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.134s  
user 0m0.133s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/tst008-test-of-dev-null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
1048576 Bytes (1,0 MB) kopiert, 0,479868 s, 10,8 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 0m0.480s  
user 0m0.479s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/buch-test.pdf of=/dev/null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
2076382 Bytes (2,0 MB) kopiert, 2,38893 s, 11,2 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 2m04.132s  
user 2m04.131s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$ time dd if=/CLionProjects/untitled5_random_mmcu/video-test.mpg of=/dev/null bs=1M status=progress  
1+1 Datensätze ein  
1+1 Datensätze aus  
1048576 Bytes (1,0 MB) kopiert, 174,875 s, 11,2 MB/s  
  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$  
(base) real 2m54.175s  
user 2m54.174s  
sys 0m0.000s  
(base) unigine@unibz1400:~/CLionProjects/untitled5_random_mmcu$
```

Abbildung C.3.: Ubuntu Lesegeschwindigkeit mit SafeBayFS