Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Σχεδίαση και Υλοποίηση Μονοπατιού για Κρυπτογραφημένη Ε/Ε στο Χώρο Χρήστη, με Χρήση του Πλαισίου Ublk και του Μηχανισμού io_uring

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτρης Γ. Χαρίσης-Πούλος

Αθήνα, Μάρτιος 2024

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Σχεδίαση και Υλοποίηση Μονοπατιού για Κρυπτογραφημένη Ε/Ε στο Χώρο Χρήστη, με Χρήση του Πλαισίου Ublk και του Μηχανισμού io_uring

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Δημήτρης Γ. Χαρίσης-Πούλος

Επιβλέπων: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Μαρτίου 2024.

....	....	....
Νεκτάριος Κοζύρης	Γεώργιος Γκούμας	Διονύσιος Πνευματικάτος
Καθηγητής ΕΜΠ	Αν. Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Μάρτιος 2024

NATIONAL TECHINCAL UNIVERSITY OF ATHENS

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

DIVISION OF COMPUTER SCIENCE

Design and Implementation of an Encrypted I/O Path in Userspace, Using the Ublk Framework and the io_uring Mechanism

DIPLOMA THESIS

Dimitris G. Charisis-Poulos

Athens, March 2024

NATIONAL TECHINCAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Design and Implementation of an Encrypted I/O Path in Userspace, Using the Ublk Framework and the io_uring Mechanism

DIPLOMA THESIS

Dimitris G. Charisis-Poulos

Supervisor: Nectarios Koziris
Professor NTUA

Approved by the three-member examination committee on the 8th of March 2024.

....	....	....
Nectarios Koziris Professor NTUA	Georgios Goumas Assoc. Professor NTUA	Dionisios Pnevmatikatos Professor NTUA

Athens, March 2024

. . . .

Δημήτρης Γ. Χαρίσης-Πούλος
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Copyright © Δημήτρης Γ. Χαρίσης-Πούλος, 2024
Με επιφύλαξη παντός δικαιώματος. All rights reserved.
Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Η στοιχειοθεσία του κειμένου έγινε με το XƎTEX 0.999994.

Χρησιμοποιήθηκαν οι γραμματοσειρές Minion Pro, Myriad Pro και `Consolas`.

Περίληψη

Το Linux είναι ένα λειτουργικό σύστημα που τρέχει στην πλειοψηφία των εξυπηρετητών παγκοσμίως και σε πολλούς οικιακούς υπολογιστές. Μία από τις κύριες λειτουργίες ενός λειτουργικού συστήματος είναι να διαχειρίζεται αιτήματα προγραμμάτων που τρέχουν στο χώρο χρήστη και θέλουν πρόσβαση στο υλικό. Παραδοσιακά, το Linux υποστηρίζει αυτές τις κλήσεις συστήματος με σύγχρονο τρόπο, δηλαδή ολοκληρώνει πρώτα την απαιτούμενη λειτουργία και στη συνέχεια επιστρέφει τον έλεγχο στο πρόγραμμα. Παρόλο που το Linux προσφέρει τρόπους για ασύγχρονη επικοινωνία, αυτοί έχουν σημαντικά μειονεκτήματα.

Σε αυτό το πλαίσιο και για να ικανοποιήσει αυτήν την έλλειψη αξιόπιστου μηχανισμού για ασύγχρονη επικοινωνία, ο μηχανισμός io_uring ενσωματώθηκε στον πυρήνα του Linux το 2019. Το io_uring επιτρέπει στα προγράμματα να επικοινωνούν με τον πυρήνα με ασύγχρονο, γρήγορο και αποδοτικό τρόπο. Αυτό οδήγησε στην ανάπτυξη νέων πλαισίων που εκμεταλλεύονται τις δυνατότητες του io_uring. Ένα τέτοιο πλαίσιο είναι το ublk, το οποίο επιτρέπει την υλοποίηση οδηγών μπλοκ συσκευών στο χώρο χρήστη. Αυτό επιτυγχάνεται με την ύπαρξη ενός περιορισμένου module του ublk στον πυρήνα που διαβιβάζει τα αιτήματα των εφαρμογών σε έναν εξυπηρετητή (server) στο χώρο χρήστη για επεξεργασία.

Σε αυτή τη διπλωματική εργασία, επεκτείναμε το πλαίσιο ublk, ενσωματώνοντας ένα μονοπάτι κρυπτογράφησης απευθείας στον εξυπηρετητή του ublk που τρέχει στο χώρο χρήστη. Έτσι, τα δεδομένα που στέλνουν οι εφαρμογές που χρησιμοποιούν το δίσκο που υποστηρίζει το ublk αποθηκεύονται κρυπτογραφημένα. Υλοποιήσαμε την κρυπτογράφηση με τρεις διαφορετικούς τρόπους - ένα σειριακό και δύο παράλληλους - και συγκρίναμε τις υλοποιήσεις αποκομίζοντας χρήσιμα συμπεράσματα σχετικά με τις δυνατότητες των υλοποιήσεων μας και τις πιθανές επεκτάσεις τους με στόχο την upstream συνεισφορά μας στο ublk.

Λέξεις-Κλειδιά

Λειτουργικά συστήματα, Linux, κρυπτογραφία, οδηγός συσκευής, εικονικές συσκευές, AES, XTS, io_uring, ublk, GPGME, OpenSSL, LUKS

Abstract

Linux is an operating system that runs on the majority of servers worldwide and on many home computers. One of the main functions of an operating system is to manage requests from userspace programs that require access to hardware. Traditionally, Linux supports these system calls in a synchronous manner, meaning it completes the requested operation first and then returns control to the program. Although Linux offers methods for asynchronous communication, they have significant drawbacks.

To address the lack of a reliable mechanism for asynchronous communication, the io_uring mechanism, which was integrated into the Linux kernel in 2019, allows programs to communicate with the kernel in an asynchronous, fast, and efficient manner. This led to the development of new frameworks that take advantage of io_uring's capabilities. One such framework is ublk, which enables the implementation of block device drivers in userspace. This is achieved via a limited ublk driver module in the kernel that forwards application requests to a server in userspace for processing.

In this thesis, we expanded the ublk framework by integrating a cryptographic path directly into the ublk server running in userspace. Thus, data sent by applications using the ublk-supported disk are stored encrypted. We implemented the encryption in three different ways - one serial and two parallel - and compared the implementations, drawing useful conclusions about the capabilities of our implementations and their potential extensions with the aim of contributing our work upstream to ublk.

Keywords

Operating systems, Linux, cryptography, AES, XTS, io_uring, ublk, GPGME, OpenSSL, block device drivers, character device drivers, virtual block devices

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εργαστήριο Υπολογιστικών Συστημάτων της σχολής ΗΜΜΥ του ΕΜΠ, υπό την επίβλεψη του καθηγητή κ. Νεκτάριου Κοζύρη, τον οποίο ευχαριστώ για την ευκαιρία που μου έδωσε να ασχοληθώ με το συγκεκριμένο θέμα.

Ιδιαίτερη μνεία, θα ήθελα να κάνω στον Δρ. Βαγγέλη Κούκη, η βοήθεια του οποίου υπήρξε πολύτιμη καθ΄ όλη τη διάρκεια αυτής της διπλωματικής εργασίας. Οι παρατηρήσεις του, τα σχόλια του, αλλά και ο γενικότερος τρόπος με τον οποίο προσέγγιζε τα προβλήματα υπήρξαν για εμένα ένα πολύτιμο σχολείο. Είναι σίγουρο ότι χωρίς τη βοήθεια του δεν θα ήταν εφικτή η εκπόνηση της παρούσας διπλωματικής και γι΄ αυτό τον ευχαριστώ θερμά.

Θα ήθελα επίσης να ευχαριστήσω την Δρ. Χλόη Αλβέρτη για τη συνεργασία μας στην αρχή αυτής της διπλωματικής εργασίας.

Τέλος, θέλω να ευχαριστήσω την οικογένεια μου, τους γονείς μου και τον αδερφό μου, καθώς και τους φίλους μου για την υποστήριξή και την υπομονή τους κατά τη συγγραφή αυτής της διπλωματικής αλλά και γενικότερα την παρουσία τους στη ζωή μου.

Δημήτρης Χαρίσης-Πούλος
Μάρτιος 2024

Contents

Περίληψη **vii**

Abstract **ix**

Ευχαριστίες **xi**

Εκτενής Περίληψη **xvii**

0.1 Εισαγωγή xvii

0.1.1 Διατύπωση Προβλήματος xviii

0.1.2 Προτεινόμενη Λύση xix

0.2 Υπόβαθρο xxi

0.2.1 Linux OS xxi

0.2.2 Στοίβα Εισόδου/Εξόδου (I/O) xxiii

0.2.3 Κρυπτογραφία xxvi

0.3 Σχεδίαση xxvii

0.3.1 Τρόποι Επικοινωνίας xxvii

0.3.2 io_uring xxix

0.3.3 Ublk xxx

0.3.4 Κρυπτογραφημένο Ublk xxxii

0.4 Υλοποίηση xxxvi

0.5 Αξιολόγηση xxxvii

0.6 Επίλογος xxxviii

0.6.1 Μελλοντικό Έργο xxxix

1 Introduction 1

1.1 Motivation 1

1.2 Problem Statement 3

1.3 Proposed Solution 5

1.4 Outline 7

2 Background 9

2.1 Linux OS 9

2.1.1 Operating System vs Kernel 10

2.1.2 And...what a kernel does? 11

2.1.3 user mode vs kernel mode 11

2.1.4 userspace vs kernelspace 12

2.2 Kernel Architecture 14

2.2.1 Monolithic vs Microkernel 14

2.2.2 Device Drivers 17

2.2.3 A dive into /dev directory 19

2.2.4 Character Device Drivers 21

2.2.5 Miscellaneous Device Drivers 24

2.3 Disks 24

2.3.1 Time related concepts 26

2.3.2 Disk Types 28

2.4 I/O Stack 30

2.4.1 Application 31

2.4.2 Virtual File System (VFS) 31

2.4.3 Filesystem (FS) 34

2.4.4 Block Layer 35

2.4.5 Block Device Drivers 40

2.4.6 Storage Device 42

2.5 Cryptography 42

2.5.1 Introduction to cryptography 43

2.5.2 Symmetric vs Asymmetric Cryptography 43

2.5.3 Introduction to AES 45

2.5.4 Mathematical Background 47

3 Design **53**

3.1 Synchronous vs Asynchronous vs Blocking vs Non-Blocking 53

3.1.1 Synchronous API 53

3.1.2 Asynchronous API 54

3.1.3 Blocking 55

3.1.4 Non-Blocking 55

3.2 io_uring 57

3.2.1 io_uring overview 58

3.2.2 io_uring system calls 61

3.2.3 Thankfully...liburing! 64

3.2.4 Advanced modes of operation 66

3.3 Coroutines 68

3.3.1 And...what are coroutines? 71

3.3.2 Coroutines in C++ 72

3.4 Ublk 76

3.4.1 Ublk overview 78

3.4.2 Initial Phase: Setting up the Environment 82

3.4.3 Second Phase: Ublk Server Internal Setup 88

3.4.4 Third phase: The Data Path 94

3.5 Encrypted Ublk 97

3.5.1 Overview of Encrypted Ublk 98

3.5.2 Key Setup design 100

3.5.3 Single-Thread Encryption 106

3.5.4 Intra-Block Encryption 112

3.5.5 Inter-Block Encryption 116

3.6 AES 123

3.6.1 Structure of AES 123

3.6.2 Modes of Operation 130

3.6.3 XEX Tweakable Block Ciphertext Stealing (XTS) 134

3.7 Linux Unified Key Setup (LUKS) 138

3.7.1 Key hierarchy 139

3.7.2 Secret Splitting 141

3.7.3 Key Derivation Functions (KDF) 142

3.7.4 LUKS internal structure 143

3.7.5 LUKS semantics 143

4 Implementation **149**

4.1 Overview 149

4.2 Key Setup Implementation 151

4.3 Single-Thread Encryption Implementation 156

4.4 Intra-Block Encryption Implementation 161

4.5 Inter-Block Encryption Implementation 167

5 Evaluation **179**

5.1 Machine Specification 179

5.2 fio 179

5.3 Experimental Evaluation 183

5.3.1 Metrics Without AES-NI Support 185

5.3.2 Comments on Results (no AES-NI support) 189

5.3.3 Metrics With AES-NI Support 192

5.3.4 Comments on Results (with AES-NI support) 196

6 Conclusion **197**

6.1 Concluding Remarks 197

6.2 Future Work 198

Bibliography **199**

Εκτενής Περίληψη

0.1 Εισαγωγή

Το Linux είναι ένα Λειτουργικό Σύστημα (ΛΣ) που χρησιμοποιείται ευρέως τόσο για εταιρικούς όσο και για οικιακούς σκοπούς. Η πλειονότητα των servers που λειτουργούν παγκοσμίως, των smartphones με λειτουργικό Android και των 500 κορυφαίων υπερυπολογιστών χρησιμοποιούν Linux. Πολλά από τα χαρακτηριστικά του Linux το έχουν καταστήσει το επιθυμητό ΛΣ για πολλές εταιρείες και προμηθευτές, κυρίως επειδή είναι ανοιχτού κώδικα, μη ιδιοκτησιακό και επεκτάσιμο.

Ο κύριος σκοπός ενός ΛΣ είναι να λειτουργεί ως γέφυρα μεταξύ των εφαρμογών χώρου χρήστη και του υλικού. Οι εφαρμογές που λειτουργούν στο χώρο χρήστη δεν επιτρέπεται να επικοινωνούν απευθείας με το υλικό. Εάν μια εφαρμογή θέλει να έχει πρόσβαση σε αυτό (π.χ. να διαβάσει από έναν δίσκο), πρέπει να ζητήσει από το ΛΣ να εκτελέσει την αντίστοιχη ενέργεια εκ μέρους της. Αυτές οι αιτήσεις είναι γνωστές ως **κλήσεις συστήματος**.

Οι κλήσεις συστήματος είναι ένα σύνολο από διεπαφές (API) που προσφέρει το ΛΣ σε διεργασίες χώρου χρήστη, για να τους επιτρέψει να επικοινωνήσουν με το υλικό.

Η μετάβαση από την κατάσταση χρήστη (user mode) στην κατάσταση πυρήνα (kernel mode) όμως έρχεται με ένα κόστος. Πέραν αυτού του κόστους, μία διεργασία που εκτελεί μία κλήση συστήματος, μπλοκάρει στον πυρήνα περιμένοντας να εκτελεστεί η αίτηση της, τουλάχιστον για την περίπτωση του blocking I/O (περισσότερα στην ενότητα 3.1). Αυτό σημαίνει ότι ακόμη και αν αυτή η διεργασία έχει άλλες εργασίες για να εκτελέσει που δεν εξαρτώνται από τα αποτελέσματα της κλήσης συστήματος, δεν μπορεί να τις εκτελέσει. Αυτά τα ζητήματα είναι ιδιαίτερα κρίσιμα σε εφαρμογές με υ-

ψηλές προδιαγραφές, που απαιτούν πολύ γρήγορη πρόσβαση στα δεδομένα με χαμηλή καθυστέρηση.

Το **io_uring** είναι ένας μηχανισμός που επιτρέπει σε διεργασίες να εκτελούν κλήσεις συστήματος με ασύγχρονο τρόπο, και ενσωματώθηκε στην version 5.1 του πυρήνα. Σε γενικές γραμμές, το io_uring παρέχει δύο buffers μνήμης που είναι κοινοί μεταξύ του χώρου χρήστη και του χώρου πυρήνα. Ολόκληρη η επικοινωνία λαμβάνει χώρα εκεί. Η αρχιτεκτονική του υποστηρίζει τη φυσική ομαδοποίηση των αιτημάτων (κλήσεων συστήματος), οδηγώντας σε σημαντική μείωση των εναλλαγών μεταξύ χώρου χρήστη και χώρου πυρήνα. Γενικά είναι μια διεπαφή σχεδιασμένη με τέτοιον τρόπο ώστε να παρέχει αποδοτικότητα, κλιμάκωση και επεκτασιμότητα.

Ως φυσική συνέπεια, εμφανίζονται νέα frameworks, με στόχο να εκμεταλλευτούν τη νέα, ταχύτερη διεπαφή επικοινωνίας που παρέχει το io_uring. Η ιδέα είναι ότι αν ξοδεύουμε λιγότερο χρόνο επικοινωνώντας με τον πυρήνα, τότε μπορούμε να εξάγουμε λειτουργικότητα από αυτόν και να την υλοποιήσουμε στο χώρο χρήστη, που γενικά παρέχει αυξημένη ασφάλεια, ευελιξία και ευκολία στο debugging.

Ένα τέτοιο framework είναι το **ublk**, που ενσωματώθηκε από τον Ming Lei στον πυρήνα του Linux v.6.0 [Lei]. Το ublk είναι ένα πλαίσιο για την υλοποίηση οδηγών μπλοκ συσκευών (block device drivers) στο χώρο χρήστη. Αποτελείται από δύο "στοιχεία". Έναν οδηγό εντός του πυρήνα και έναν εξυπηρετητή στο χώρο χρήστη.

0.1.1 Διατύπωση Προβλήματος

Στην τρέχουσα έκδοσή του, το ublk δεν υποστηρίζει κρυπτογραφημένες λειτουργίες. Η κρυπτογράφηση του αρχείου που λειτουργεί ως δίσκος, εξαρτάται αποκλειστικά από τις δυνατότητες κρυπτογράφησης του συστήματος. Το ublk αποτελείται από δύο επικοινωνούντα μέρη: έναν οδηγό μέσα στον πυρήνα, τον οποίο θα αναφέρουμε ως "**ublk driver**", και ένα εξυπηρετητή στο χώρο χρήστη, τον οποίο θα αναφέρουμε ως "**ublk server**". Η επικοινωνία τους πραγματοποιείται μέσω του io_uring. Το ublk μπορεί να υποστηρίξει πολλούς στόχους (δηλαδή να υλοποιήσει διαφορετικές εικονικές μπλοκ συσκευές). Σε αυτή τη διπλωματική εργασία, επικεντρωνόμαστε στον **στόχο συσκευής loop**. Μια συσκευή loop είναι μια συσκευή μπλοκ που αντιστοιχίζει τα δεδομένα της όχι σε μια φυσική συσκευή όπως ένας σκληρός δίσκος ή ένας οπτικός δίσκος, αλλά στα

μπλοκ ενός κανονικού αρχείου σε ένα σύστημα αρχείων ή σε μια άλλη συσκευή μπλοκ [mpd].

Όταν μια εφαρμογή που χρησιμοποιεί το ublk επιθυμεί να στείλει ένα αίτημα I/O, αυτό το αίτημα διασχίζει τα υποσυστήματα του πυρήνα, φτάνοντας τελικά στον ublk driver. Στη συνέχεια, ο driver προωθεί το αίτημα στον ublk server χρησιμοποιώντας το io_-uring, παρέχοντας στο χώρο χρήστη την ευκαιρία να χειριστεί το αίτημα πριν αυτό σταλεί στο loop target.

Συνεπώς, τα δεδομένα αποθηκεύονται στο αρχείο με την ίδια μορφή που είχαν όταν αρχικά στάλθηκαν από την εφαρμογή. Αυτό το γεγονός, επιβάλλει στην εφαρμογή να είναι είτε ενήμερη για το περιβάλλον στο οποίο λειτουργεί το ublk (π.χ. ενδεχομένως να υλοποιεί κρυπτογράφηση το σύστημα αρχείων), είτε να χειρίζεται την κρυπτογράφηση μόνη της αν ο επιθυμητός βαθμός ιδιωτικότητας δεν εγγυάται διαφορετικά. Για παράδειγμα, η εφαρμογή μπορεί να χρειαστεί να κρυπτογραφήσει τα δεδομένα πριν τα στείλει στο ublk.

0.1.2 Προτεινόμενη Λύση

Σχεδιάσαμε και υλοποιήσαμε ένα σχήμα κρυπτογράφησης που επιτρέπει στο ublk να αποθηκεύει τα δεδομένα μιας εφαρμογής κρυπτογραφημένα στο δίσκο. Με αυτόν τον τρόπο, εφαρμογές με απαιτήσεις ασφαλείας μπορούν να χρησιμοποιήσουν τη συσκευή που υποστηρίζει το ublk χωρίς να χρειάζεται να διαχειρίζονται ανεξάρτητα την ασφάλεια των δεδομένων τους. Η υλοποίησή μας πραγματοποιείται εξ ολοκλήρου στο χώρο χρήστη, ως μέρος του ublk server.

Χωρίσαμε τη διαδικασία κρυπτογράφησης σε δύο διαφορετικές φάσεις:

- **Φάση 1:** Κατά την εκκίνηση του ublk ορίζουμε ένα αρχείο .gpg που περιέχει ένα κρυπτογραφικό κλειδί. Το ίδιο το αρχείο είναι κρυπτογραφημένο, εξασφαλίζοντας την προστασία του. Σε αυτή τη φάση αποκρυπτογραφούμε αυτό το αρχείο μεταδεδομένων και εξάγουμε το κλειδί.

- **Φάση 2:** Ο ublk server χρησιμοποιεί το παραγόμενο κλειδί από τη φάση 1, για να εκτελέσει κρυπτογραφικές λειτουργίες I/O στη συσκευή. Χρησιμοποιούμε

συμμετρική κρυπτογράφηση, συγκεκριμένα το AES-256 σε XTS operation mode για αυτόν τον σκοπό.

Έτσι, όταν μια εφαρμογή γράφει δεδομένα στη συσκευή, ο ublk server κρυπτογραφεί τα δεδομένα και τα αποθηκεύει στον δίσκο. Ομοίως, όταν ένας χρήστης ζητά δεδομένα από τη συσκευή, ο ublk server ανακτά και αποκρυπτογραφεί τα δεδομένα πριν επικοινωνήσει την απάντηση του στον ublk driver.

Ο στόχος μας ήταν να εξασφαλίσουμε την ασφάλεια των δεδομένων που αποθηκεύονται στο δίσκο και να παρέχουμε στις εφαρμογές τη δυνατότητα να χρησιμοποιήσουν το ublk χωρίς να χρειάζεται να διαχειριστούν μόνες τους τη διαδικασία κρυπτογράφησης. Η λύση μας εγγυάται ότι καμία πληροφορία δεν θα αποθηκευτεί ποτέ στη συσκευή σε ακρυπτογράφητη μορφή. Ως αποτέλεσμα, ακόμη και αν ο δίσκος κλαπεί ή κατασχεθεί, δεν μπορούν να εξαχθούν δεδομένα χωρίς γνώση του κλειδιού κρυπτογράφησης.

Για κάθε φάση, χρησιμοποιήσαμε μια διαφορετική κρυπτογραφική βιβλιοθήκη για να επιτύχουμε το στόχο μας. Στη φάση 1, χρησιμοποιήσαμε τη βιβλιοθήκη *GnuPG Made Easy* (GPGME) [GNUa], για να κρυπτογραφήσουμε και να αποκρυπτογραφήσουμε το σχετικό αρχείο και να αποκτήσουμε το κλειδί δεδομένων, ενώ στη φάση 2, χρησιμοποιήσαμε τη βιβλιοθήκη *OpenSSL* [Ope] για την κρυπτογράφηση και αποκρυπτογράφηση δεδομένων κατά τη διάρκεια του I/O.

Επιπλέον, υλοποιήσαμε τη δεύτερη φάση της κρυπτογράφησης με τρεις διαφορετικούς τρόπους:

- **Single-thread:** Το κύριο νήμα του ublk server, που είναι υπεύθυνο για την επικοινωνία με τον ublk driver, χειρίζεται επίσης την κρυπτογράφηση και αποκρυπτογράφηση.

- **Intra-block parallelism:** Δημιουργήσαμε μια δεξαμενή νημάτων που αποτελείται από νήματα-εργάτες. Αντί να εκτελεί προσωπικά την κρυπτογράφηση/αποκρυπτογράφηση σε κάθε buffer το κύριο νήμα, διαιρεί τον buffer και διανέμει κάθε τμήμα στα νήματα-εργάτες.

- **Inter-block parallelism:** Παρόμοια με τον intra-block παραλληλισμό, δημιουργήσαμε μια δεξαμενή νημάτων με νήματα-εργάτες. Ωστόσο, σε αυτήν την περί-

πτωση, το κύριο νήμα παραδίδει ολόκληρο το buffer σε κάποιο νήμα-εργάτη και συνεχίζει.

Με αυτόν τον τρόπο, μπορέσαμε να πειραματιστούμε με την αποδοτικότητα και τις προκλήσεις του παράλληλου προγραμματισμού σε δύο διακριτές μορφές. Συγκρίναμε αυτές τις προσεγγίσεις με την πρώτη περίπτωση της σειριακής κρυπτογράφησης, και τελικά ανακαλύψαμε τους περιορισμούς και τα πιθανά οφέλη του παραλληλισμού. Η δυνατότητά για παραλληλισμό στο κρυπτογραφικό μας σχήμα, ήταν δυνατή λόγω της χρήσης του αλγορίθμου AES [Wika] σε λειτουργία XTS [NIS].

Ο παραλληλισμός της λειτουργίας XTS οφείλεται στο γεγονός ότι κάθε μπλοκ δεδομένων μπορεί να κρυπτογραφηθεί ανεξάρτητα χωρίς να χρειάζεται πληροφορία από κάποιο άλλο μπλοκ. Αυτό το χαρακτηριστικό επιτρέπει στα κρυπτογραφημένα δεδομένα να διαιρεθούν σε τμήματα με τέτοιο τρόπο ώστε διαφορετικά μπλοκ δεδομένων να μπορούν να επεξεργαστούν ταυτόχρονα σε διαφορετικές μονάδες επεξεργασίας [MAAR].

0.2 Υπόβαθρο

0.2.1 Linux OS

Το Linux, ξεκίνησε από τον Linus Torvalds το 1991 ως ένα προσωπικό έργο, και έχει εξελιχθεί σε ένα αξιόπιστο, ανοιχτού κώδικα, λειτουργικό σύστημα που έχει γίνει ευρέως αποδεκτό σε μια ποικιλία πλατφορμών, περιλαμβάνοντας προσωπικούς υπολογιστές, κινητές συσκευές, ενσωματωμένα συστήματα και υπερυπολογιστές.

Ένα από τα βασικά πλεονεκτήματα του, είναι η φύση του ως λογισμικό ανοιχτού κώδικα, που διέπεται από την Άδεια Γενικής Δημόσιας Χρήσης GNU έκδοση 2 (GPLv2), η οποία επιτρέπει σε οποιονδήποτε ενδιαφερόμενο να μελετήσει, να τροποποιήσει και να διανείμει το λογισμικό. Αυτή η διαφάνεια έχει συμβάλει στην ασφάλεια του Linux, καθώς προβλήματα ασφαλείας μπορούν να εντοπιστούν και να αντιμετωπιστούν γρήγορα από την κοινότητα.

Το Linux υλοποιεί ένα μοντέλο ασφαλείας βασισμένο σε δαχτυλίδια προνομίων, με τον πυρήνα να λειτουργεί σε ένα προνομιακό "kernel mode" (Ring 0) και τις εφαρμογές χρήστη να τρέχουν σε ένα μη προνομιακό "user mode" (Ring 3). Αυτός ο διαχωρισμός

διασφαλίζει ότι οι εφαρμογές χρήστη δεν μπορούν να παρέμβουν απευθείας στις λειτουργίες του πυρήνα, ενισχύοντας την ασφάλεια και τη σταθερότητα του συστήματος. Όταν μια εφαρμογή χρήστη απαιτεί πρόσβαση σε hardware, πρέπει να κάνει μία κλήση συστήματος, προκαλώντας μια μετάβαση από το user mode στο kernel mode. Αυτός ο μηχανισμός επιτρέπει στον πυρήνα να διαμεσολαβεί με ασφάλεια στην πρόσβαση στους πόρους του συστήματος, αποτρέποντας τις μη εξουσιοδοτημένες λειτουργίες.

Ο αρχιτεκτονική του πυρήνα του Linux χαρακτηρίζεται ως **μονολιθική**, πράγμα που σημαίνει ότι όλα τα υποσυστήματα του είναι ενσωματωμένα στον πυρήνα και λειτουργούν σε kernel mode, διευκολύνοντας τις απευθείας κλήσεις μεταξύ των υποσυστημάτων. Αυτό αντιπαρατίθεται με την προσέγγιση του **μικροπυρήνα** (microkernels), η οποία δίνει προτεραιότητα στο μινιμαλισμό διατηρώντας τον πυρήνα μικρό και αναθέτοντας πολλές λειτουργίες του σε διεργασίες που τρέχουν στο χώρο χρήστη. Έτσι ενισχύεται η ανθεκτικότητα του συστήματος μέσω της απομόνωσης των υποσυστημάτων του, πληρώνοντας όμως το κόστος της διαδιεργασιακής επικοινωνίας (IPC) μεταξύ των υποσυστημάτων του που τρέχουν με διαφορετικά προνόμια.

Για να εξισορροπήσει την αποδοτικότητα των μονολιθικών πυρήνων με την αρθρωτή δομή των μικροπυρήνων, το Linux χρησιμοποιεί kernel modules. Αυτά είναι δυναμικά αρχεία αντικειμενικού κώδικα που επιτρέπουν την επέκταση της λειτουργικότητας του πυρήνα κατά παραγγελία χωρίς την ανάγκη επανεκκίνησης ή επαναμεταγλώττισης του. Τα modules λειτουργούν σε kernel mode, προσφέροντας έναν συμβιβασμό που διατηρεί την απόδοση ενώ προωθεί έναν ενιαίο σχεδιασμό, επιτρέποντας την οργανωμένη διαχείριση υποσυστημάτων και την οικονομία μνήμης.

Το Linux διακρίνει τις συσκευές σε τρεις βασικές κατηγορίες: συσκευές χαρακτήρων (character devices), συσκευές μπλοκ (block devices) και συσκευές δικτύου (network devices). Ενώ οι συσκευές χαρακτήρων προσφέρουν διαδοχική πρόσβαση στα δεδομένα, επιτρέποντας λειτουργίες σε μεμονωμένα bytes (π.χ. πληκτρολόγια, σειριακές θύρες), οι συσκευές μπλοκ λειτουργούν με δεδομένα σε μπλοκ σταθερού μεγέθους και υποστηρίζουν τυχαία πρόσβαση (π.χ. σκληροί δίσκοι). Οι δικτυακές συσκευές διευκολύνουν τη μετάδοση δεδομένων μεταξύ των υπολογιστών.

Η υλοποίηση ενός οδηγού συσκευής χαρακτήρων, περιστρέφεται γύρω από τη δομή `struct file_operations`, η οποία περιέχει δείκτες σε συναρτήσεις που εκτελούν διάφορες "εργασίες" στη συσκευή. Αυτές οι συναρτήσεις αντιστοιχούν σε κλήσεις συστή-

ματος όπως `open()`, `close()`, `read()` και `write()`, επιτρέποντας στον οδηγό να ορίσει πώς η συσκευή αλληλεπιδρά με αυτές τις κλήσεις.

0.2.2 Στοίβα Εισόδου/Εξόδου (I/O)

Η στοίβα I/O στο Linux είναι ένα δομημένο μονοπάτι που ακολουθεί μια κλήση συστήματος `read()` ή `write()` από μια εφαρμογή και περνάει μέσα από διάφορα υποσυστήματα του πυρήνα μέχρι να φτάσει στο φυσικό δίσκο. Αυτή η διαδρομή περιλαμβάνει αρκετά επίπεδα, συμπεριλαμβανομένων μεταξύ άλλων του Εικονικού Συστήματος Αρχείων (VFS), του Συστήματος Αρχείων (FS), του Block Layer και των οδηγών συσκευών που αλληλεπιδρούν τελικά με τη φυσική συσκευή αποθήκευσης.

Επίπεδο Εφαρμογής

Στην κορυφή της στοίβας Ε/Ε βρίσκεται το επίπεδο εφαρμογής, όπου οι εφαρμογές χρήστη αλληλεπιδρούν με τα αρχεία μέσω κλήσεων συστήματος όπως `open()`, `read()` και `write()`. Από την οπτική των εφαρμογών, τα αρχεία θεωρούνται ως γραμμικές ακολουθίες bytes, επιτρέποντας την απλή πρόσβαση και τροποποίηση των δεδομένων. Αυτή η αλληλεπίδραση υψηλού επιπέδου κρύβει τις υποκείμενες πολυπλοκότητες του συστήματος αρχείων και της διαχείρισης συσκευών.

Εικονικό Σύστημα Αρχείων (VFS)

Το Εικονικό Σύστημα Αρχείων (VFS) είναι ένα κρίσιμο υποσύστημα στο Linux που διαχειρίζεται τις λειτουργίες των συστημάτων αρχείων, προσφέροντας μια ενοποιημένη διεπαφή στις εφαρμογές χρήστη και επιβάλλοντας ένα κοινό μοντέλο αρχείου σε διαφορετικά συστήματα αρχείων. Το VFS αφαιρεί τις λεπτομέρειες των υποκείμενων filesystems, επιτρέποντας στις εφαρμογές να αλληλεπιδρούν με τα αρχεία χωρίς να χρειάζεται να γνωρίζουν τον τύπο του filesytem ή τις λεπτομέρειες της υλοποίησής του. Το επιτυγχάνει αυτό μέσω βασικών οντοτήτων εντός του πυρήνα όπως είναι τα `struct superblock`, `struct inode`, `struct file` και `struct dentry`. Αυτή η αρχιτεκτονική όχι μόνο απλοποιεί τις λειτουργίες αρχείων για τις εφαρμογές αλλά εξασφαλίζει επίσης συνέπεια και αποδοτικότητα στη διαχείριση των αρχείων σε διαφορετικά filesystems.

Σύστημα Αρχείων (FS)

Ένα σύστημα αρχείων στο Linux οργανώνει τα δεδομένα ιεραρχικά, προσφέροντας μια διεπαφή για πρόσβαση βασισμένη σε αρχεία. Ο στόχος των filesystems είναι να υλοποιούν λειτουργίες που ορίζονται από το VFS για να διασφαλίζεται η συμβατότητα. Ένα κύριο χαρακτηριστικό που διαχειρίζονται τα συστήματα αρχείων είναι η λειτουργία της **κρυφής μνήμης σελίδων (page cache)**, η οποία επιταχύνει τις λειτουργίες I/O κρατώντας στη μνήμη τις προσπελασμένες ή πιθανώς προσπελαστέες (στο μέλλον) σελίδες. Οι αιτήσεις ανάγνωσης γεμίζουν την page cache με δεδομένα από τον δίσκο (αν δεν είναι ήδη παρόντα), ενώ οι αιτήσεις εγγραφής μπορούν να χρησιμοποιήσουν την page cache ως προσωρινό αποθηκευτικό χώρο (write-back cache), επιτρέποντας ασύγχρονες εγγραφές στον δίσκο.

Ωστόσο, οι εφαρμογές μπορούν να επιλέξουν να παρακάμψουν την page cache χρησιμοποιώντας τη σημαία `O_DIRECT` κατά την κλήση συστήματος `open()`, οδηγώντας σε άμεσες μεταφορές δεδομένων μεταξύ των buffers του χώρου χρήστη και του αρχείου στο δίσκο. Για τη διατήρηση της συνέπειας, ο πυρήνας σημειώνει τις ενημερωμένες σελίδες της page cache ως "dirty" και τις αδειάζει στον δίσκο όποτε χρειάζεται, εξασφαλίζοντας την ακεραιότητα των δεδομένων σε διαφορετικές εφαρμογές που προσπελαύνουν τα ίδια δεδομένα. Στην περίπτωση που η λειτουργία Ε/Ε χρησιμοποιεί την page cache αναφέρεται ως **buffered** ενώ στην περίπτωση που την προσπερνάει αναφέρεται ως **direct**.

Generic Block Layer

To block layer στο Linux λειτουργεί ως ένας ενδιάμεσος στη στοίβα I/O, συνδέοντας υψηλότερα επίπεδα όπως το σύστημα αρχείων με τους οδηγούς συσκευών μπλοκ τυποποιώντας την πρόσβαση σε αυτούς. Με αυτόν τον τρόπο απλοποιεί τις αλληλεπιδράσεις με τις συσκευές μπλοκ παρέχοντας μια συνεπή διεπαφή για τα συστήματα αρχείων.

Single-Queue Block Layer

Ιστορικά, το block layer χρησιμοποιούσε έναν σχεδιασμό μονής ουράς (Single-Queue), όπου κάθε συσκευή μπλοκ "συνδεόταν" με μία δικιά της δομής (`struct request_queue`). Αυτή η δομή αποτελούνταν από request (`struct request`), τα οποία με τη σειρά τους περιελάμβαναν δομές `struct bio`, καθένα εκ΄ των οποίων περιγράφει μία λειτουρ-

γία I/O. Αυτός ο σχεδιασμός ήταν κυρίως επικεντρωμένος στους HDDs, έχοντας ως στόχο την ελαχιστοποίηση των χρόνων αναζήτησης της κεφαλής του δίσκου με τη συγκέντρωση διαδοχικών αιτημάτων (διαδοχικών `bio` σε ένα). Ο σχεδιασμός της Single-Queue περιελάμβανε μηχανισμούς όπως το plugging για την καθυστέρηση της επεξεργασίας αιτημάτων, επιτρέποντας τη συγχώνευση διαδοχικών `bio` έχοντας ως στόχο την ενίσχυση της αποτελεσματικότητας. Ωστόσο, αυτή η προσέγγιση άρχισε να εμφανίζει προβλήματα επίδοσης με την εμφάνιση των SSDs, καθώς η σύγκρουση κλειδωμάτων μεταξύ των CPUs για πρόσβαση στη δομή `request_queue` έγινε ένα σημαντικό εμπόδιο κατά την υλοποίηση ενός I/O.

Multi-Queue Block Layer

Για να αντιμετωπιστούν οι προκλήσεις απόδοσης που παρουσίασε ο σχεδιασμός Single-Queue στην εποχή των SSDs, το Linux ενσωμάτωσε μια νέα αρχιτεκτονική στο block layer γνωστή και ως blk-mq, ακολουθώντας ένα σχεδιασμό πολλαπλών ουρών (Multi-Queue). Αυτή η αρχιτεκτονική διαχωρίζει την επεξεργασία αιτημάτων σε δύο επίπεδα: σε software staging queues και σε hardware dispatch queues, με τις πρώτες να κατανέμονται ανά CPU ή κόμβο NUMA και τις δεύτερες να καθορίζονται από τις δυνατότητες της συσκευής. Αυτός ο διαχωρισμός μειώνει σημαντικά τη σύγκρουση κλειδωμάτων αναθέτοντας ξεχωριστές ουρές σε κάθε CPU, διευκολύνοντας έτσι την επεξεργασία αιτημάτων. Ο σχεδιασμός blk-mq κάνει ξεκάθαρο τον διαχωρισμό των ευθυνών, με το επίπεδο μπλοκ να διαχειρίζεται κυρίως τις software queues και τους οδηγούς συσκευών να εποπτεύουν τις hardware queues, διευκολύνοντας την ανάπτυξη οδηγών και βελτιώνοντας τη συνολική απόδοση της στοίβας I/O.

Οδηγοί Μπλοκ Συσκευών

Οι οδηγοί συσκευών μπλοκ στον πυρήνα του Linux είναι κρίσιμοι για την επικοινωνία μεταξύ του λειτουργικού συστήματος και των συσκευών αποθήκευσης. Χρησιμοποιούν τη δομή `struct gendisk` για να περιγράψουν έναν δίσκο, συμπεριλαμβάνοντας βασικές πληροφορίες του.

Η αρχικοποίηση των οδηγών μπλοκ συσκευών περιλαμβάνει την αρχικοποίηση και τη διαμόρφωση των δομών `gendisk` και `request_queue`, συνήθως εντός της ρουτίνας αρχικοποίησης του οδηγού.

Δίσκος

Οι δίσκοι λειτουργούν ως οι κύριες συσκευές αποθήκευσης στα συστήματα υπολογιστών, περιλαμβάνοντας μεταξύ άλλων από Hard Disk Drivers (HDDs) και Solid-State Drives (SSDs) έως οπτικούς δίσκους και δισκέτες. Για κάθε τύπο δίσκου υπάρχει ένα trade-off μεταξύ ταχύτητας, κόστους και ανθεκτικότητας, με τις ταχύτερες επιλογές αποθήκευσης να είναι γενικά πιο ακριβές ή λιγότερο ανθεκτικές.

0.2.3 Κρυπτογραφία

Για την υλοποίηση του κρυπτογραφικού μονοπατιού στον ublk server, έπρεπε, εκτός των άλλων, να εξοικειωθούμε με τα βασικά χαρακτηριστικά της κρυπτογραφίας, για να είμαστε σε θέση να κάνουμε τις κατάλληλες σχεδιαστικές κρυπτογραφικές επιλογές.

Η κρυπτογραφία είναι η επιστήμη της κωδικοποίησης πληροφοριών, και η μετατροπή των δεδομένων σε μία μορφή ακατανόητη και άχρηστη για κάθε μη εξουσιοδοτημένη οντότητα, με στόχο την εξασφάλιση μιας ασφαλούς επικοινωνίας μεταξύ των υποκειμένων που ανταλλάσσουν πληροφορία. Στην ψηφιακή εποχή, η κρυπτογραφία συχνά σχετίζεται με τη διαδικασία μετατροπής κανονικού κειμένου (plaintext) σε κρυπτογραφημένο κείμενο (ciphertext), εξασφαλίζοντας την εμπιστευτικότητα, την ακεραιότητα και την ταυτοποίηση των δεδομένων και των επικοινωνιών.

Η κρυπτογραφία χωρίζεται σε δύο βασικούς τύπους: τη συμμετρική και την ασύμμετρη. Η συμμετρική κρυπτογραφία χρησιμοποιεί ένα κλειδί κοινό και για την κρυπτογράφηση και για την αποκρυπτογράφηση, ενώ η ασύμμετρη χρησιμοποιεί ένα ζεύγος κλειδιών: ένα δημόσιο για την κρυπτογράφηση και ένα ιδιωτικό για την αποκρυπτογράφηση. Η συμμετρική κρυπτογράφηση είναι γενικά πιο γρήγορη και κατάλληλη για την κρυπτογράφηση μεγάλων όγκων δεδομένων, αλλά απαιτεί ασφαλή κατανομή του κοινού κλειδιού. Η ασύμμετρη κρυπτογράφηση απλοποιεί την κατανομή των κλειδιών και μπορεί να παρέχει ταυτοποίηση, καθιστώντας την έτσι κατάλληλη για ασφαλείς ανταλλαγές κλειδιών και ψηφιακές υπογραφές σε ανοικτά συστήματα. Οι δύο μέθοδοι μπορούν να συνδυαστούν για μια πιο ολοκληρωμένη λύση ασφαλείας, όπως στην κρυπτογράφηση SSL/TLS [F5], που χρησιμοποιεί ασύμμετρη κρυπτογράφηση για τη δημιουργία μιας ασφαλούς συνεδρίας και την ανταλλαγή του συμμετρικού κλειδιού και στη συνέχεια συμμετρική κρυπτογράφηση κατά την διάρκεια ανταλλαγής των δεδομέ-

νων.

Εισαγωγή στον AES

Το Advanced Encryption Standard (AES) είναι ο πιο διαδεδομένος αλγόριθμος συμμετρικής κρυπτογράφησης, αντικαθιστώντας το DES μετά από την επιλογή του από το NIST το 2001 [Wika]. Σχεδιάστηκε από τους Βέλγους κρυπτογράφους Vincent Rijmen και Joan Daemen και χρησιμοποιεί μεγέθη κλειδιών 128, 192 και 256 bits. Λειτουργεί ως ένα block cipher, χωρίζοντας τα δεδομένα σε blocks των 128 bits και εκτελείται σε γύρους, με τον αριθμό των γύρων να εξαρτάται από το μέγεθος του κλειδιού (10, 12 και 14 γύρους).

Τα 128 bits της εισόδου του AES διαιρούνται εσωτερικά σε chunks των 16 bytes, δημιουργώντας έναν 4x4 πίνακα, γνωστό και ως πίνακα κατάστασης, όπου πραγματοποιούνται όλες οι λειτουργίες του AES. Κάθε γύρος περιλαμβάνει τέσσερα στάδια: Byte Substitution, Shift Row, Mix Columns, και Key Addition.

Στο Byte Substitution, κάθε byte της εισόδου αντιστοιχεί με ένα άλλο byte.

Στο Shift Rows τα bytes κάθε σειράς του πίνακα κατάστασης, περιστρέφονται κυκλικά, προσθέτοντας οριζόντια διάχυση, ενώ στη φάση του Mix Columns κάθε στήλη του πίνακα κατάστασης πολλαπλασιάζεται με έναν προκαθορισμένο πίνακα, προσθέτοντας κάθετη διάχυση.

Τέλος, στην φάση του Key Addition πραγματοποιεί XOR του τρέχοντος πίνακα κατάστασης με ένα 128-bit υποκλειδί που προκύπτει από το κύριο κλειδί.

Η διαδικασία αποκρυπτογράφησης αντιστρέφει τα βήματα αυτά, χρησιμοποιώντας τις αντίστροφες λειτουργίες για κάθε στάδιο και εφαρμόζοντας τα υποκλειδιά σε αντίστροφη σειρά.

0.3 Σχεδίαση

0.3.1 Τρόποι Επικοινωνίας

Στον προγραμματισμό, συναντάμε συχνά τις έννοιες της **σύγχρονης**, **ασύγχρονης**, **blocking** και **non-blocking** επικοινωνίας. Η επιλογή ενός εξ΄ αυτών των τύπων επικοινωνίας ε-

ξαρτάται από τις απαιτήσεις και τα χαρακτηριστικά της κάθε εφαρμογής.

Σύγχρονη Επικοινωνία

Όταν μια διεργασία κάνει μια σύγχρονη κλήση προς το λειτουργικό σύστημα (όπως ένα σύγχρονο read), η ολοκλήρωση της εργασίας που θέλει η διεργασία θα έχει ολοκληρωθεί όταν θα εκτελέσει την επόμενη εντολή της. Ωστόσο, μπορεί να μπλοκάρει περιμένοντας μέσα στον πυρήνα αν το λειτουργικό σύστημα δεν είναι έτοιμο να απαντήσει αμέσως.

Ασύγχρονη επικοινωνία

Μια διεργασία κάνει μία ασύγχρονη κλήση στο λειτουργικό σύστημα όταν η εργασία που ζητάει από τον πυρήνα να εκτελέσει, πραγματοποιείται χωρίς η διεργασία να μπλοκάρει στον πυρήνα. Η διεργασία ξεκινά την εργασία και εάν ο πυρήνας δεν είναι έτοιμος, δεν "κοιμάται" εντός του αλλά επιστρέφει. Γι΄ αυτό το λόγο απαιτείται ένας τρόπος επικοινωνίας για να ειδοποιηθεί η διεργασία όταν ο πυρήνας ολοκληρώσει το αίτημα της.

Blocking επικοινωνία

Μια blocking κλήση θα μπλοκάρει τη διεργασία που την εκτελεί μέχρι να είναι έτοιμο το αποτέλεσμα. Αυτό συμβαδίζει με την έννοια της σύγχρονης κλήσης και γι' αυτό συχνά χρησιμοποιούνται ως ταυτόσημες έννοιες.

Non-blocking επικοινωνία

Μια non-blocking κλήση δεν θα μπλοκάρει τη διεργασία αν το αποτέλεσμα δεν είναι έτοιμο, αλλά δεν θα ξεκινήσει και την εργασία την οποία αιτήθηκε, σε αντίθεση με τη μία ασύγχρονη κλήση που επίσης δεν θα μπλοκάρει αλλά επιπλέον θα ενημερώσει τον πυρήνα να ξεκινήσει την εργασία.

0.3.2 io_uring

Το πιο συνηθισμένο μοντέλο επικοινωνίας που χρησιμοποιούν σήμερα οι εφαρμογές για I/O είναι το *σύγχρονο*, χρησιμοποιώντας τις βασικές κλήσεις συστήματος `read()/write()` ή τις νεότερες εκδοχές τους, `pread()/pwrite()` και `preadv()/pwritev()`. Ωστόσο, σε ορισμένες περιπτώσεις, ένα ασύγχρονο μοντέλο ταιριάζει καλύτερα. Το Linux, εκτός του io_uring, υποστηρίζει δύο ασύγχρονα APIs: το POSIX AIO και το Linux AIO (ή libaio).

Το POSIX AIO στο Linux υλοποιείται από τη βιβλιοθήκη glibc μέσω νημάτων. Κάθε ασύγχρονη κλήση I/O ανατίθενται σε ένα thread που δημιουργεί η glibc το οποίο μπλοκάρει περιμένοντας το αποτέλεσμα. Αυτό απαιτεί επικοινωνία μεταξύ των εμπλεκόμενων threads για να ενημερωθεί το κύριο thread ότι το αποτέλεσμα είναι έτοιμο, αλλά δεν κλιμακώνει καλά λόγω διότι δημιουργεί ένα thread ανά request. Αντίθετα, το Linux AIO χρησιμοποιεί κλήσεις συστήματος, με τον πυρήνα να χειρίζεται απευθείας τα αιτήματα χωρίς το βάρος του ενός thread ανά request που έχει το POSIX AIO. Ωστόσο, έχει άλλους σημαντικούς περιορισμούς όπως το ότι δεν υποστηρίζει buffered I/O, περιορίζοντας σημαντικά τις εφαρμογές που μπορούν να χρησιμοποιήσουν αυτό το API.

Οι περιορισμοί των Linux AIO και POSIX AIO οδήγησαν πολλές εφαρμογές που ήθελαν να βελτιώσουν την απόδοσή τους μέσω μιας ασύγχρονης διεπαφής να υλοποιήσουν τις δικές τους ασύγχρονες λύσεις, διαχειριζόμενες ένα pool νημάτων γι΄ αυτόν το σκοπό. Το **io_uring** έγινε merged στην έκδοση 5.1 του πυρήνα για να να καλύψει αυτό το κενό στο ασύγχρονο I/O. Αρχικά υποστήριζε κυρίως κλήσεις συστήματος για block I/O αλλά στη συνέχεια εξελίχθηκε υποστηρίζοντας περισσότερες κλήσεις συστήματος, έχοντας ως στόχο να γίνει ένα γενικό πλαίσιο για την εκτέλεση ασύγχρονων κλήσεων συστήματος.

Το io_uring διαθέτει δύο buffers μνήμης: τον Submission Queue (SQ) για την υποβολή αιτημάτων και τον Completion Queue (CQ) για την ανάκτηση απαντήσεων. Και οι δύο είναι μοιραζόμενοι μεταξύ της εφαρμογής και του πυρήνα. Κάθε αίτημα περιγράφεται από ένα Submission Queue Entry (SQE) και για κάθε SQE, ο πυρήνας τοποθετεί ένα Completion Queue Entry (CQE) στο CQ όταν το αποτέλεσμα είναι έτοιμο.

Το io_uring υποστηρίζει τρεις κλήσεις συστήματος για τη δημιουργία των memory

mapped buffers και την επικοινωνία με τον πυρήνα: την `io_uring_setup()`, την `io_uring_enter()` και την `io_uring_register()`.

Η `io_uring_setup()` είναι η πρώτη κλήση που πρέπει να κάνει μια εφαρμογή για να δημιουργήσει ένα instance του io_uring, δηλώνοντας τον αριθμό των entries και λαμβάνοντας πίσω έναν file descriptor για την αναφορά στο instance αυτό. Μετά την επιτυχή επιστροφή, η εφαρμογή πρέπει να εκτελέσει ένα `mmap()` για να "εμφανίσει " τους buffers στο χώρο διευθύνσεων της.

Η κλήση συστήματος `io_uring_enter()` ενημερώνει τον πυρήνα για την ύπαρξη των SQEs στο Submission Queue. Με αυτήν την κλήση συστήματος υπάρχει η δυνατότητα να υποβάλλει η εφαρμογή πολλαπλά αιτήματα (πολλά SQEs) με μία μόνο κλήση. Με άλλα λόγια μία εφαρμογή μπορεί να "ετοιμάσει" πολλά SQEs και να ενημερώσει τον πυρήνα με μία μόνο κλήση στην `io_uring_enter()`, πληρώνοντας έτσι την επιβάρυνση ενός μόνο context switch.

Τέλος, η κλήση συστήματος `io_uring_register()` επιτρέπει την προεγγραφή buffers και file descriptors της εφαρμογής στον πυρήνα, μειώνοντας έτσι το overhead ανά I/O.

Το io_uring παρέχει μια βιβλιοθήκη χώρου χρήστη, την liburing [Axbe], που διευκολύνει σε μεγάλο βαθμό την χρήση του από τις εφαρμογές, καθώς απλοποιεί την πολυπλοκότητα της διαχείρισης των κλήσεων συστήματος παρέχοντας τους μία διεπαφή υψηλού επιπέδου.

Επιπλέον, το io_uring υποστηρίζει δύο λειτουργίες polling, τις λεγόμενες SQPOLL και IOPOLL, για εφαρμογές με υψηλές απαιτήσεις σε χαμηλή καθυστέρηση. Το SQPOLL δημιουργεί ένα kernel thread που ελέγχει την SQ για διαθέσιμα αιτήματα, ενώ το IOPOLL επιτρέπει στη διεργασία που έκανε το αίτημα να ελέγχει ενεργά για ολοκληρώσεις τη συσκευή στόχο.

0.3.3 Ublk

Η εισαγωγή του μηχανισμού επικοινωνίας io_uring στο Linux, έχει ενθαρρύνει τη μεταφορά λειτουργιών από τον πυρήνα στο χώρο χρήστη όπου αυτό είναι εφικτό. Οι βασικοί λόγοι για αυτήν τη μετακίνηση περιλαμβάνουν (α) τη μεγαλύτερη ευελιξία στον προγραμματισμό, καθώς οι προγραμματιστές μπορούν να χρησιμοποιήσουν διάφορες γλώσσες προγραμματισμού και βιβλιοθήκες (β) την ευκολία στο debugging (γ) τη με-

γαλύτερη ασφάλεια, καθώς τα προγράμματα στο χώρο χρήστη μειώνουν τον κίνδυνο ολικών προβλημάτων ή κρασαρισμάτων του συστήματος σε αντίθεση με το ενδεχόμενο ύπαρξης κάποιου σφάλματος στον πυρήνα και (δ) επιτρέπουν ανεξάρτητη ανάπτυξη και συντήρηση από τον πυρήνα, του οποίου το περιβάλλον ανάπτυξης είναι αρκετά περιοριστικό.

Ωστόσο, υπάρχει ένας σημαντικός περιορισμός: η επίδοση. Οι κρίσιμες λειτουργίες που απαιτούν γρήγορους χρόνους απόκρισης μπορεί να μην είναι κατάλληλες για μεταφορά στο χώρο χρήστη λόγω της αυξημένης καθυστέρησης στην επικοινωνία του πυρήνα με το userspace.

Το **ublk framework** ακολουθεί την ίδια λογική με δύο επικοινωνούντα μέρη, ένα στο χώρο χρήστη, το οποίο θα ονομάζουμε "ublk server" ή απλώς "server", και ένα στον πυρήνα, που θα το αναφέρουμε ως "ublk driver" ή απλά "driver". Το ublk, όπως σχεδιάστηκε από τον Ming Lei, έχει ενσωματωθεί ως πειραματικό module στον πυρήνα του Linux στην έκδοση 6.0 και υποστηρίζει προς το παρόν συγκεκριμένους τύπους εικονικών μπλοκ συσκευών (τους Null, loop, NBD και qcow2).

Στην παρούσα διατριβή, η έμφαση δίνεται στο target "loop", για το οποίο υλοποιήθηκε ένα κρυπτογραφικό σύστημα εξ΄ ολοκλήρου στο χώρο χρήστη (στον ublk server), όπως αναφέραμε και στην εισαγωγή.

Γενική Σχεδίαση του ublk

Ο ublk driver ξεκινάει, αφότου φορτώσουμε το σχετικό module στον πυρήνα, δημιουργώντας μια συσκευή χαρακτήρων, την `/dev/ublk-control`, με την οποία ο ublk server θα αλληλεπιδρά. Ο ublk server ξεκινώντας, δημιουργεί ένα instance του io_uring για να "μιλήσει" με τον driver, και ανοίγει την συσκευή `ublk-control`.

Κάθε SQE που θα κάνει submit ο server στο io_uring θα είναι τύπου `IORING_OP_URING_CMD`. Το συγκεκριμένο opcode, είναι το io_uring command passthrough, που αντιστοιχεί σε ένα ασύγχρονο `ioctl()`. Με άλλα λόγια, ο server με SQEs τύπου `IORING_OP_URING_CMD` μπορεί να στέλνει αυθαίρετες εντολές στον driver μέσω του io_uring.

Ξεκινώντας ο server εκδίδει μια εντολή `UBLK_CMD_ADD_DEV`, καθορίζοντας τις δυνατότητές της συσκευής που θα υποστηρίζει το ublk, όπως ο αριθμός των ουρών που η συσκευή θα δέχεται requests, η χωρητικότητα αυτών των ουρών, το μέγεθος του μπλοκ

κοκ. Η επιτυχής εκτέλεση αυτής της εντολής έχει ως συνέπεια τη δημιουργία ενός ακόμα character device, του `/dev/ublkcN`, όπου το 'N' είναι το αναγνωριστικό της συσκευής. Αυτή είναι η συσκευή που θα λειτουργήσει ως βοηθός για την επικοινωνία του server με τον driver κατά την διάρκεια του data path.

Εν΄ συνεχεία ο server, κάνει `mmap()` μία περιοχή από descriptors, για να έχει πρόσβαση κατά την διάρκεια του data path. Κάθε ένας descriptor πρακτικά θα περιγράφει ένα `read/write` request, και χρησιμοποιώντας το device `/dev/ublkcN`, κάνει submit τόσα requests, όσα είναι και το μέγεθος της ουράς του ενημερώνοντας τον driver ότι το συγκεκριμένο slot είναι έτοιμο να δεχθεί κάποιο request.

Τέλος, ο server εκδίδει μία εντολή `UBLK_CMD_DEV_START`, που έχει ως αποτέλεσμα να εμφανιστεί η συσκευή μπλοκ `/dev/ublkbN`. Από αυτό το σημείο και έπειτα οι εφαρμογές μπορούν να χρησιμοποιούν την συσκευή αυτή. Τα requests που στέλνουν, φτάνουν στον ublk driver μέσα στον πυρήνα, και ο driver με τη σειρά του αφού ενημερώσει για το είδος του request τον αντίστοιχο descriptor στη μοιραζόμενη περιοχή με τον server, θα συμπληρώσει ένα CQE για να ικανοποιήσει το αντίστοιχο ανοιχτό SQE που είχε κάνει προηγουμένως ο server. Ο server ξυπνώντας, λαμβάνει την απάντηση του driver, και με βάση πληροφορία που αντλεί μέσα από το CQE, πηγαίνει στον descriptor της μοιραζόμενης μνήμης και εκτελεί το request που περιγράφει επικοινωνώντας το με τον target. Όταν η δουλειά ολοκληρωθεί και ο target απαντήσει ο server επικοινωνεί εκ΄ νέου με τον driver για να τον ενημερώσει για την ολοκλήρωση του request, και ο driver με τη σειρά του γνωστοποιεί στο block layer του πυρήνα την απάντηση για να διαδοθεί μέχρι την εφαρμογή.

0.3.4 Κρυπτογραφημένο Ublk

Σε αυτό το σημείο, θα αναφερθούμε περιληπτικά στις επεκτάσεις που κάναμε στο ublk framework σε αυτήν την διατριβή με στόχο την υποστήριξη κρυπτογραφικών λειτουργιών. Οι επεκτάσεις μας βεβαιώνουν ότι τα δεδομένα που επικοινωνούν οι εφαρμογές με το ublk θα φτάσουν στον target σε κρυπτογραφημένη μορφή.

Ξεκινώντας θα αναφέρουμε κάποιες σχεδιαστικές μας επιλογές. Αρχικά, όσον αφορά το είδος της κρυπτογράφησης, αποφασίσαμε τη χρήση συμμετρικής κρυπτογραφίας για την κρυπτογράφηση/αποκρυπτογράφηση δεδομένων, με την επιλογή του αλγορίθμου

Advanced Encryption Standard (AES) που χρησιμοποιείται ευρύτατα και είναι ασφαλής και γρήγορος. Επιλέξαμε τη λειτουργία XTS του AES, διότι πέραν του γεγονότος ότι είναι το default operation mode του AES σε πολλές κρυπτογραφικές λύσεις σε block devices, όπως το LUKS, το BitLocker και το VeraCrypt έχει και το πλεονέκτημα ότι επιτρέπει την παράλληλη κρυπτογράφηση/αποκρυπτοργάφηση, πράγμα που είναι αναγκαίο για τις παράλληλες υλοποιήσεις μας.

Όπως αναφέραμε και στην εισαγωγή 0.1.2, χωρίσαμε τη διαδικασία κρυπτογράφησης σε δύο διακριτά στάδια. Στο πρώτο στάδιο, διαχειριζόμαστε το κύριο κλειδί που θα χρησιμοποιήσουμε κατά την διάρκεια του data path για την κρυπτογράφηση/αποκρυπτοργάφηση των δεδομένων. Θα παρουσιάσουμε συνοπτικά το πρώτο αυτό κοινό στάδιο για όλες μας τις υλοποιήσεις, και εν΄ συνεχεία θα αναφέρουμε τις κύριες σχεδιαστικές επιλογές για κάθε μία υλοποίηση ξεχωριστά.

Διαχείριση Κύριου Κλειδιού

Θέλαμε να αποθηκεύουμε το κύριο κλειδί (master key) σε ένα αρχείο ώστε να μπορούμε να το χρησιμοποιήσουμε ξανά. Ωστόσο, δεν θα ήταν σοφή κίνηση να αποθηκεύσουμε το master key ακρυπτογράφητο σε ένα αρχείο. Επιπλέον, θέλαμε να μπορούμε να προσθέτουμε νέα κλειδιά και να αφαιρούμε κλειδιά χωρίς να αλλάζουμε το ίδιο το master key. Και αυτό διότι αν το master key άλλαζε θα απαιτούσε την επανακρυπτογράφηση ολόκληρου του δίσκου, πράγμα που δεν είναι πρακτικό για μέσα αποθήκευσης που μπορεί να αποθηκεύουν μεγάλο όγκο δεδομένων.

Αυτή η ιδέα μας οδήγησε να υιοθετήσουμε ένα σχήμα **ιεραρχίας κλειδιών**. Θέλαμε ένα άλλο key που θα χρησιμοποιείται για την κρυπτογράφηση του master key. Αυτή η έννοια μας επιτρέπει να χειριζόμαστε την κρυπτογράφηση με μεγαλύτερη ευελιξία.

Επίσης θέλαμε να υποστηρίζουμε τις ακόλουθες υψηλού επιπέδου λειτουργίες:

- Δημιουργία ενός νέου κρυπτογραφικού δίσκου. Αρχικοποίηση ενός master key, και αποθήκευσή του σε ένα αρχείο σε κρυπτογραφημένη μορφή.

- Άνοιγμα ενός κρυπτογραφημένου δίσκου ανακτώντας το master key, που παρέχεται σε κρυπτογραφημένη μορφή από το αρχείο.

- Προσθήκη ενός νέου κλειδιού.

- Αφαίρεση ενός κλειδιού.

Για να μπορέσουμε να υλοποιήσουμε κάποιες από τις παραπάνω λειτουργίες χρειαζόμασταν μία περιοχή του δίσκου όπου θα αποθηκεύουμε πληροφορίες μεταδεδομένων. Έτσι ακολουθώντας τη λογική του LUKS, καταλάβαμε μία περιοχή στην αρχή του αρχείου που χρησιμοποιείται ως δίσκος στο ublk (loop target), για τα μεταδεδομένα της κρυπτογράφησης. Ένα από τα στοιχεία που κρατάμε σε αυτά τα μεταδεδομένα είναι και το hash του master key έτσι ώστε να μπορούμε να βεβαιώνουμε την είσοδο σε χρήστες που παρέχουν το σωστό κρυπτογραφημένο κλειδί.

Σε αυτήν την πρώτη φάση, χρησιμοποιούμε τη βιβλιοθήκη GnuPG Made Easy (GPGME) [GNUa] για τη διαχείριση των κρυπτογραφικών λειτουργιών.

Κρυπτογράφηση Single-Thread

Η ενσωμάτωση της κρυπτογράφησης στον ublk server ξεκινά με την πρώτη μας λύση, την single-thread. Σε αυτήν κάθε νήμα του ublk server αναλαμβάνει τόσο την επικοινωνία με τον ublk driver και τον στόχο (το αρχείο που χρησιμοποιείται ως δίσκος) όσο και τις λειτουργίες κρυπτογράφησης και αποκρυπτογράφησης.

Για να διευκολύνουμε την κρυπτογράφηση, προσθέσαμε κάποιους προσωρινούς buffers, πέρα από τους αρχικούς στους οποίους ο server δεχόταν τα δεδομένα από τον driver σε περίπτωση ενός write request ή έδινε τα δεδομένα στον driver στην περίπτωση ενός read request. Οι προσωρινοί buffers χρησιμοποιούνται για την αποθήκευση των κρυπτογραφημένων δεδομένων πριν από την εγγραφή στο στόχο (στην περίπτωση ενός write request) και για την αποθήκευση των κρυπτογραφημένων δεδομένων που ανακτώνται από το στόχο πριν από την αποκρυπτογράφηση (στην περίπτωση ενός read request).

Στα write request, τα δεδομένα κρυπτογραφούνται πριν την αποστολή στο στόχο, ενώ στα read, τα κρυπτογραφημένα δεδομένα αποκρυπτογραφούνται μετά την ανάκτησή τους από το στόχο. Οι κρυπτογραφικές λειτουργίες χρησιμοποιούν τη βιβλιοθήκη OpenSSL για AES κρυπτογράφηση σε λειτουργία XTS.

Η κρυπτογράφηση και η αποκρυπτογράφηση πραγματοποιούνται ανά sector. Οι buffer που διαχειρίζεται ο server για να αποθηκεύει τα δεδομένα, τόσο οι αρχικοί όσο και οι "προσωρινοί" που καταλάβαμε εμείς έχουν μέγεθος 0.5Mb. Στην περίπτωση λοιπόν

που ο sector είναι 512 bytes, κάθε buffer μπορεί να αποθηκεύσει έως 1024 sector. Κάθε λειτουργία κρυπτογράφησης/αποκρυπτογράφησης λοιπόν πραγματοποιείται διαδοχικά για κάθε sector των 512-byte από τον ublk server.

Κρυπτογράφηση Intra-Block

Η δεύτερη κρυπτογραφική μας λύση, βασίζεται στην παράλληλη εκτέλεση της κρυπτογράφησης/αποκρυπτογράφησης AES-XTS. Σε αντίθεση με τη single-thread υλοποίηση, όπου η κρυπτογράφηση/αποκρυπτογράφηση κάθε τμήματος των 512 bytes γίνεται διαδοχικά, σε αυτή την υλοποίηση κάθε νήμα του ublk server δημιουργεί ένα pool νημάτων και διανέμει την εργασία στα νήματα-εργάτες.

Συνεπώς, οι εργασίες κρυπτογράφησης και αποκρυπτογράφησης ανατίθενται σε ξεχωριστά νήματα αντί να εκτελούνται απευθείας από το κύριο νήμα του ublk server. Αυτό επιτρέπει την παράλληλη επεξεργασία των sectors με τους εργάτες να κρυπτογραφούν παράλληλα από τους αρχικούς buffers στους προσωρινούς, στην περίπτωση ενός write request, και να αποκρυπτογραφούν από τους προσωρινούς στους αρχικούς buffers στην περίπτωση ενός read request.

Ο αριθμός των νημάτων είναι παραμετροποιημένος και μπορεί να τεθεί κατά την εκκίνηση του server. Ο συγχρονισμός μεταξύ του κύριου νήματος και των νημάτων-εργατών επιτυγχάνεται μέσω δύο barriers, επιτρέποντας στο κύριο νήμα να αναθέτει εργασίες και να περιμένει για την ολοκλήρωσή τους χωρίς την ανάγκη για επιπλέον κλειδώματα.

Κρυπτογράφηση Inter-Block

Στην τρίτη κρυπτογραφική μας λύση, που ονομάσαμε inter-block, το κύριο νήμα του ublk server αναθέτει ολόκληρο το buffer για κρυπτογράφηση ή αποκρυπτογράφηση σε ένα νήμα-εργάτη, και συνεχίζει με την επεξεργασία των επόμενων αιτημάτων. Και σε αυτήν την υλοποίηση λοιπόν έχουμε ένα pool από νήματα-εργάτες, μόνο που το κύριο νήμα του ublk server δεν μοιράζει τον buffer για κρυπτογράφηση/αποκρυπτογράφηση στους εργάτες όπως έκανε στη δεύτερη υλοποίηση, αλλά αναθέτει ολόκληρο το αίτημα σε κάποιον από αυτούς χωρίς το ίδιο να μπλοκάρει περιμένοντας την απάντηση.

Σε αντίθεση λοιπόν με τις προηγούμενες υλοποιήσεις, η inter-block προσέγγιση διατηρεί την I/O-bound φύση του ublk server, αφαιρώντας την ανάγκη από το κύριο νήμα να καταναλώνει CPU εκτελώντας την κρυπτογράφηση (στην περίπτωση της single-thread υλοποίησης) ή να περιμένει την ολοκλήρωση των κρυπτογραφικών λειτουργιών (στην περίπτωση της intra-block υλοποίησης). Αυτό επιτρέπει στον ublk server να χειρίζεται τα αιτήματα I/O διατηρώντας την αρχιτεκτονική του προσανατολισμένη στην γρήγορη επεξεργασία αυτών των αιτημάτων και την αποστολή τους στον κατάλληλο παραλήπτη.

Η επικοινωνία μεταξύ του κύριου νήματος και των νημάτων-εργατών γίνεται μέσω ενός eventfd και ενός condition variable. Πιο συγκεκριμένα, το κύριο νήμα, χρησιμοποιεί το condition variable για να ενημερώσει τα νήματα-εργάτες για νέες εργασίες. Αυτό επιτρέπει στο κύριο νήμα να συνεχίζει την υποβολή και αναμονή για απαντήσεις αιτημάτων χωρίς να καθυστερεί από τις κρυπτογραφικές λειτουργίες. Από την άλλη η επικοινωνία από έναν εργάτη προς το κύριο νήμα για την γνωστοποίηση της ολοκλήρωσης της κρυπτογράφησης/αποκρυπτογράφησης μίας εργασίας γίνεται μέσω του eventfd, για το οποίο το κύριο νήμα έχει υποβάλει ένα SQE στον πυρήνα για poll request. Με άλλα λόγια, το main thread έχει ζητήσει από τον πυρήνα να το ξυπνήσει σε περίπτωση που κάποιος γράψει στο eventfd. Συνεπώς με αυτόν τον τρόπο, ενημερώνεται για την ολοκλήρωση κάποιας κρυπτογραφικής εργασίας.

Η διαχείριση των αιτημάτων μεταξύ του κύριου νήματος και των νημάτων-εργατών γίνεται μέσω ενός κοινού μεταξύ τους αντικειμένου που περιλαμβάνει ουρές υποβολής και ολοκλήρωσης, καθώς και έναν πίνακα από δομές αιτημάτων (`struct request`) κάθε μία εκ΄ των οποίων περιγράφει μία κρυπτογραφική εργασία.

0.4 Υλοποίηση

Καθώς στο συγκεκριμένο κεφάλαιο παρουσιάζουμε μια συνοπτική περιγραφή των λύσεων μας, τις υλοποιήσεις μας μπορείτε να τις βρείτε στο αντίστοιχο αγγλικό κεφάλαιο 4.

0.5 Αξιολόγηση

Σε αυτό το κεφάλαιο, παρουσιάζουμε συνοπτικά την αξιολόγηση των υλοποιήσεών μας. Συγκρίνουμε τις τρεις κρυπτογραφικές μας λύσεις για να εξάγουμε συμπεράσματα σχετικά με τα πιθανά οφέλη ή παγίδες των παράλληλων λύσεών μας, με στόχο να βρούμε τρόπους να τις βελτιώσουμε στο μέλλον. Οι μετρήσεις που παρουσιάζουμε έγιναν χρησιμοποιώντας το fio [Axbc], ένα παραμετροποιήσιμο πρόγραμμα που δοκιμάζει φορτία εργασίας και μετρά την απόδοση τους.

Πραγματοποιήσαμε τις μετρήσεις μας σε ένα μηχάνημα της Amazon Web Services (AWS), συγκεκριμένα ένα μοντέλο c5d.2xlarge. Αυτό το μηχάνημα χρησιμοποιεί την αρχιτεκτονική x86_64.

Εξαιτίας της συχνής χρήσης του AES σε πλήθος εφαρμογών, η Intel αρχικά σχεδίασε και ενσωμάτωσε στους επεξεργαστές τις κάποιες νέες εντολές με στόχο τη βελτίωση της απόδοσης του αλγορίθμου κρυπτογράφησης AES. Το Advanced Encryption Standard New Instructions (AES-NI) εισήγαγε 6 νέες εντολές για εκτέλεση του AES σε επίπεδο υλικού.

Ο επεξεργαστής που χρησιμοποιήσαμε για τις μετρήσεις υποστηρίζει το AES-NI, όπως και η βιβλιοθήκη OpenSSL. Οι δοκιμές πραγματοποιήθηκαν με τις νέες εντολές ενεργοποιημένες και απενεργοποιημένες. Για να τις απενεργοποιήσουμε, χρησιμοποιήσαμε τη μεταβλητή περιβάλλοντος `OPENSSL_ia32cap`. Κάναμε μετρήσεις με δύο είδη I/O: σύγχρονο και ασύγχρονο. Τις ασύγχρονες μετρήσεις μας τις κάναμε με το io_uring. Κάθε μέτρηση πραγματοποιήθηκε για block size 4k, 8k, 16k, 32k, 64k και 1m με στόχο να έχουμε μία πιο σφαιρική άποψη για την απόδοση των υλοποιήσεων μας σε διαφορετικού μεγέθους requests.

Για όλες τις μετρήσεις, ξεκινήσαμε το ublk με τις default παραμέτρους (μία ουρά για να δέχεται request, μεγέθους 128), και οι παράλληλες υλοποιήσεις (ublk-intra και ublk-inter) είχαν 4 νήματα-εργάτες.

Γενικά Σχόλια

Μετρήσεις χωρίς AES-NI

Οι μετρήσεις που λάβαμε χωρίς τις νέες εντολές, συμβαδίζουν με αυτά που περιμέναμε

να δούμε από τις υλοποιήσεις μας. Πιο αναλυτικά, στην περίπτωση των σύγχρονων λειτουργιών, η ublk-single υλοποίηση είναι καλύτερη για μικρού μεγέθους request, ενώ καθώς το request μεγαλώνει σε μέγεθος η ublk-intra κλιμακώνει καλύτερα και την ξεπερνάει. Η ublk-inter σε αυτή την περίπτωση δεν μπορεί να επωφεληθεί από την παραλληλία, διότι υπάρχει ένα μόνο request on the fly. Από την άλλη στην περίπτωση όπου έχουμε ασύγχρονα requests με το io_uring, παρατηρούμε ότι η ublk-inter υλοποίηση είναι καλύτερη ακόμη και αν αυτά είναι μικρού μεγέθους. Αντίθετα η ublk-single υλοποίηση δεν κλιμακώνει καλά, κάτι που είναι λογικό καθώς δεν μπορεί να εκμεταλλευτεί στο έπακρο το γεγονός ότι υπάρχουν περισσότερα του ενός requests on the fly.

Μετρήσεις με AES-NI

Οι μετρήσεις που λάβαμε τρέχοντας τα workloads με τις νέες εντολές, δεν συμβαδίζουν με αυτά που περιμέναμε να δούμε από τις υλοποιήσεις μας. Η ublk-single υλοποίηση ξεπερνά τις παράλληλες ακόμα και στην περίπτωση των ασύγχρονων requests. Φαίνεται ότι η γρήγορη πραγματοποίηση των κρυπτογραφικών λειτουργιών που επιτρέπουν οι νέες εντολές είναι προτιμότερη από την επιβάρυνση που προσθέτει η διαχείριση του παραλληλισμού στις ublk-intra και ublk-inter υλοποιήσεις.

Φυσικά θα χρειαστεί να πραγματοποιήσουμε μετρήσεις και σε συστήματα με υψηλότερο εύρος ζώνης, καθώς πολλές μετρήσεις μας (ιδιαίτερα requests μεγαλύτερου μεγέθους) φτάνουν στο “ταβάνι” του εύρους ζώνης τους μηχανήματος από όπου λάβαμε τις μετρήσεις και έτσι δεν μας επιτρέπουν να έχουμε μία πιο ολοκληρωμένη άποψη για την σχετική απόδοση των υλοποιήσεων μας.

0.6 Επίλογος

Το ταξίδι μας ξεκίνησε πριν από έναν περίπου χρόνο με έντονο ενδιαφέρον για τα Λειτουργικά Συστήματα και περιέργεια για τους χαμηλού επιπέδου μηχανισμούς που χρησιμοποιεί ο πυρήνας του Linux. Αυτή η περιέργεια μας οδήγησε στο io_uring. Αφού μελετήσαμε τις βασικές του αρχές, θέλαμε να εφαρμόσουμε τις γνώσεις που αποκομίσαμε σε κάτι πρακτικό, κάτι που μας οδήγησε στο ublk.

Αρχικά, η έλλειψη documentation γύρω από το ublk μας ανάγκασε να μελετήσουμε τον πηγαίο κώδικα του για να κατανοήσουμε τις λειτουργίες του. Αυτή η εμπειρία ήταν πολύτιμη, καθώς μας δίδαξε όχι μόνο νέες έννοιες αλλά και το πώς να προσεγγί-

ζουμε συστηματικά άγνωστα codebases, κάτι που είναι σημαντικό, ιδιαίτερα απέναντι σε πολύπλοκα συστήματα όπως ο πυρήνας του Linux.

Ο στόχος μας στη συνέχεια έγινε πιο ξεκάθαρος: να συνεισφέρουμε στο ublk ενσωματώνοντας απευθείας ένα κρυπτογραφικό μονοπάτι στον ublk server. Αυτό μας επέτρεψε να συνεισφέρουμε ουσιαστικά σε αυτό το framework, ενώ αποκτούσαμε πρακτική εμπειρία με πρότυπα όπως το LUKS και το AES-XTS.

Τέλος, αυτή η διατριβή μας παρείχε την ευκαιρία να εξερευνήσουμε διαφορετικές πτυχές παράλληλου προγραμματισμού, να αξιολογήσουμε τα οφέλη και τους περιορισμούς του και να αποκτήσουμε μία πρακτική αντίληψη στην εφαρμογή του.

0.6.1 Μελλοντικό Έργο

Παρακάτω παρατίθενται κάποιες μελλοντικές κατευθύνσεις σχετικά με αυτή τη διατριβή:

- Εξερεύνηση εναλλακτικών κρυπτογραφικών βιβλιοθηκών (π.χ. libgcrypt) και σύγκριση της απόδοσής τους με τα αποτελέσματα που λήφθηκαν χρησιμοποιώντας την βιβλιοθήκη OpenSSL.

- Περαιτέρω έρευνα για την ιδανική διαμόρφωση του αριθμού των νημάτων-εργατών στις παράλληλες λύσεις μας. Σκοπός μας είναι να δοκιμάσουμε και να μετρήσουμε την απόδοση των ublk-intra και ublk-inter υπό διαφορετικά φορτία εργασίας και σε διαφορετικά περιβάλλοντα με διάφορους αριθμούς νημάτων, για να κατανοήσουμε την επίδρασή τους στην απόδοση.

- Ανάπτυξη μιας αποδοτικότερης μεθόδου επικοινωνίας για την τρίτη μας υλοποίηση (ublk-inter) που μειώνει την ανάγκη για κλειδώματα στην επικοινωνία του κύριου νήματος με τα νήματα-εργάτες. Έχουμε αρχίσει να υλοποιούμε μία εναλλακτική προσέγγιση βασισμένη στην υπάρχουσα, η οποία μειώνει την ανάγκη για απόκτηση ενός κλειδώματος ανά υποβολή αιτήματος.

- Συνεισφορά της εργασίας μας upstream στο ublk.

1 Introduction

1.1 Motivation

Linux is an Operating System (OS) widely used worldwide for both industrial and domestic purposes. The majority of the servers running globally, smartphones running Android and the top 500 supercomputers prefer Linux as their OS. [Top, HS]. Many of Linux's characteristics have made it the go-to OS for many companies and vendors, mainly because it is open source, non-proprietary, and extendable. While Linux undoubtedly thrives on servers, in recent years, it has also been making significant strides in personal computers.

The main purpose of an Operating System, is to act as a bridge between userspace applications and hardware. Applications that run in userspace aren't allowed to communicate directly with the hardware. This is where the OS comes into play. If an application wants to access hardware (e.g. read from a disk), it needs to request the OS to perform the corresponding action on its behalf. These requests are known as **system calls**. When the kernel runs on behalf of a process executing a system call, we refer to it as running in **process context**. The other context in which the kernel can run is **interrupt context**, which occurs when it services an interrupt triggered by a hardware component.

System calls are a set of interfaces (an API) offered by the OS to the userspace applications, in order to allow them to interact with hardware devices such as the CPU and disks.

Putting an extra layer between the application and the hardware has several advantages [BC06]:

- Makes programming easier by freeing users from studying low-level programming characteristics.
- Greatly increases system security.
- Makes programs more portable.

Of course, nothing great comes free. System calls are expensive. Transitioning from user mode to kernel mode in order to execute a request comes at a cost. Apart from this overhead, an application that performs a system call, blocks until the request is satisfied, at least in case of blocking I/O (see Section 3.1 for types of I/O). This means that even if the application has other tasks to perform that don't depend on the results of the system call, it cannot execute them. These issues are particularly critical in applications with high specifications, that require super fast, low-latency data access. Response speed and resource utilization play pivotal roles. Consequently, a lot of effort have been directed towards improving the entire I/O data path and mitigate some of the challenges system calls expose.

Welcome, io_uring!
io_uring [Axbb] is an asynchronous system call interface for the Linux kernel, which has been merged in version 5.1 by Jens Axboe, the current maintainer of the Linux kernel block layer. It was created to solve the asynchronous I/O problem. This is a functionality that Linux has never supported as well as users would have liked [Cora]. In a nutshell, io_uring provides two memory mapped buffers, shared between userspace and kernelspace. The entire communication takes place there. It's architecture supports the natural batching of requests (i.e. system calls) and responses, leading to a significant reduction of the context switches. In general it is an interface designed in a way to provide efficiency, scalability and extendability.

As a natural consequence new frameworks are appearing, aiming to take advantage of the new, faster communication interface, io_uring provides. The idea is that if we spend less time communicating with the kernel, then we can extract functionality of it and

implement it in userspace, which generally provides increased safety, flexibility and ease for debugging.

One such framework is **ublk**, introduced by Ming Lei in Linux v.6.0 [Lei]. Ublk is a framework for implementing block device drivers in userspace. Ublk consists of two components. An in-kernel driver and a userspace component. The driver acts as a bridge between the userspace component and the rest of the Linux kernel. All the implementation is taking place in userspace.

1.2 Problem Statement

In its current version, the ublk framework doesn't support encrypted operations. The encryption of the backing file that serves as a disk, solely relies on the system's encryption capabilities. In other words, if the system running ublk does not implement any form of encryption, the data written to the backing file remains unencrypted.

A very simplified version of ublk framework is illustrated in Figure 1.1.

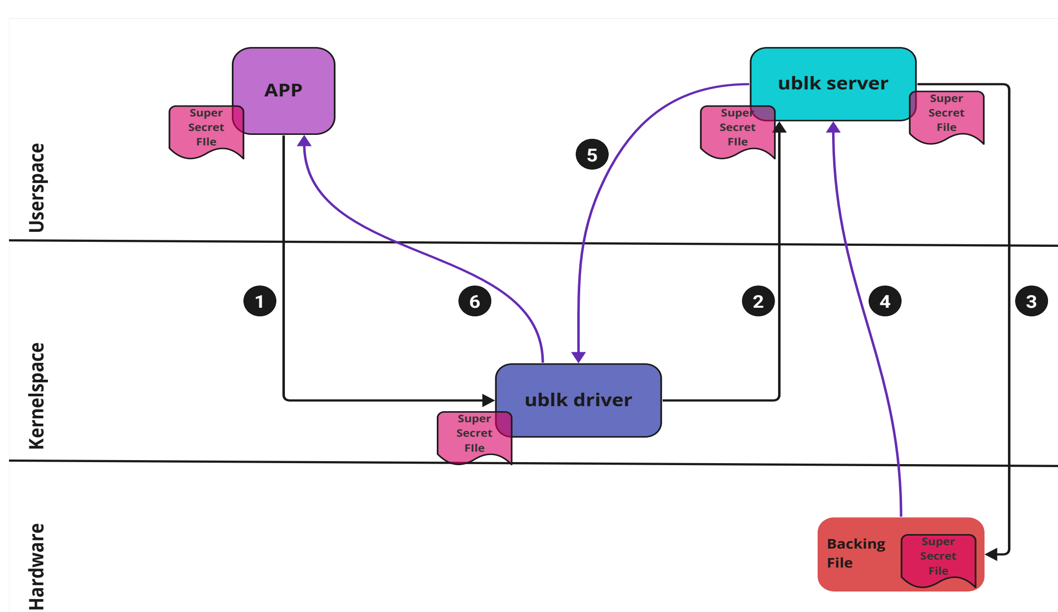

Figure 1.1: *General View of ublk Framework*

Ublk framework consists of two communicating components: an in-kernel driver component, which we will refer to as "**ublk driver**", and a userspace component, which we will refer to as "**ublk server**". These two are communicating via an io_uring instance. Ublk can support many targets (i.e. implement different Virtual Block Devices). In this

diploma thesis our focus is the **loop device target**. A loop device is a block device that maps its data blocks not to a physical device such as a hard disk or optical disk drive, but to the blocks of a regular file in a filesystem or to another block device [mpd].

When an application using ublk, wishes to send an I/O request, this request traverses through the in-kernel subsystems (see Section 2.4 for more), ultimately reaching the ublk driver. Then, the ublk driver forwards the request to ublk server using an established io_uring instance, providing userspace with an opportunity to manipulate the request before it is sent to the backing file.

Let's take a look at Figure 1.1, which helps us understand the challenge we face in the current ublk implementation. Imagine an application wanting to write data to the device. It triggers a system call and provides the corresponding buffer to the kernel. Once the request reaches the ublk driver, the data is copied to a userspace buffer within the ublk server's virtual memory and the driver informs the server that a request arrived. The ublk server then, sends the request to the backing file as is. Consequently, the raw data are stored in the backing file in the same form they had when initially sent by the application. This remains true unless the file system itself operates on the data before the block device driver comes into play or if the backing file is stored on a form of Self Encrypted Drive [Arc].

This fact, places the burden on the application to either be aware of the context and environment in which the ublk framework operates, or to handle encryption on its own if the desired level of privacy is not guaranteed otherwise. For instance, the application may need to encrypt the data prior to sending them to ublk.

Linux represents internally (almost) everything as a file. **"Everything is a file"** in Linux. It manages file access using user identification and permissions, offering a satisfactory level of security for typical use cases. Nevertheless, the lasting nature of disk storage introduces a vulnerability that standard operating system permissions alone cannot fully address. If your disk is seized or stolen, your secrets are exposed. Hence, when handling sensitive files, it becomes mandatory to store data on the disk in a manner that guarantees the safety of information. To achieve this, a lot of effort has gone into how we can protect data stored on a disk. Numerous software and hardware solutions have been suggested over time, but finding a one-size-fits-all answer is not an easy task in a diversify field like software management. Each case has its unique demands and might

need a different approach compared to others.

1.3 Proposed Solution

We designed and implemented an encryption schema that enables ublk to save the data of an application in the backing file encrypted. This way, security-sensitive applications can utilize the device exposed by the ublk framework, without needing to independently manage the security of their data. Our implementation is carried out completely in userspace, as part of the ublk server.

We split the encryption process in two different phases.

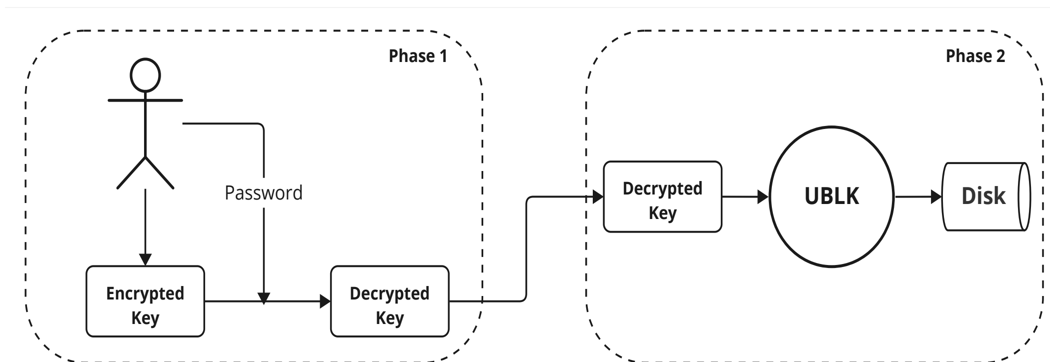

Figure 1.2: *2-phase Encryption Schema*

- **Phase 1:** When starting the ublk framework we specify a .gpg file that contains a cryptographic key. The file itself is encrypted, ensuring its protection. In this phase we decrypt this metadata file and extract the key.
- **Phase 2:** Ublk server uses the derived key from phase 1, to perform cryptographic I/O operations on the device. We employ symmetric cryptography, specifically AES-256 in XTS mode for this purpose.

So, when an application writes data to the device, the ublk server encrypts the data and stores it in the backing file. Similarly, when a user requests data from the device, the ublk server retrieves and decrypts the data. This decrypted data is then passed back to the ublk driver and subsequently to the end user.

Our goal was to ensure the security of data stored in the backing file, and provide applications with the opportunity to use the ublk framework without needing to manage

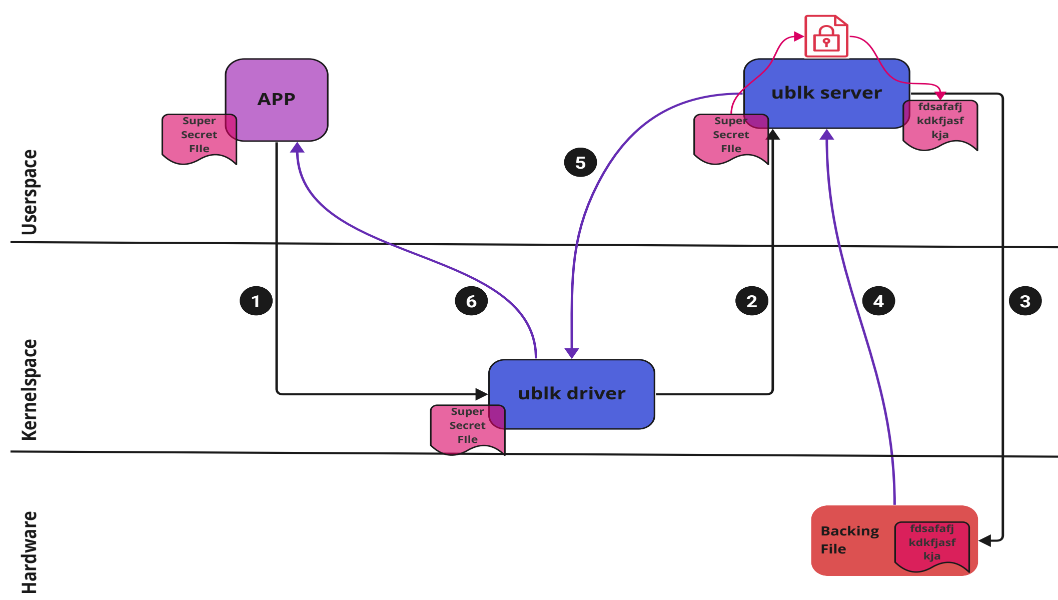

Figure 1.3: *General View of a Write Request in Encrypted Ublk Framework*

the encryption procedure themselves. Our solution guarantees that no file information will ever be stored on the device in plain format. As a result, even if the disk is stolen or seized, no data can be extracted without knowledge of the encryption key.

For each phase, we used a different cryptographic library to achieve our goal. In phase 1, we used the *GnuPG Made Easy* (GPGME) [GNUa] library, to encrypt and decrypt the relevant file and obtain the data key, while in phase 2, we utilized the *OpenSSL* library [Ope] for on-the-fly data encryption and decryption.

Furthermore, we implemented the second phase of the encryption in three different ways:

- **Single-thread:** The main thread of ublk server, responsible for communication with the ublk driver, was also handled encryption and decryption.

- **Intra-block parallelism:** We created a thread pool consisting of working threads. Instead of personally performing encryption/decryption on each buffer, the main thread now divides the buffer and distributes each segment to the working threads.

- **Inter-block parallelism:** Similar to intra-block parallelism, we created a thread pool with working threads. However, in this case, the main thread hands over the entire buffer to a working thread before continuing.

In doing so, we were able to experiment with the efficiency and challenges of parallel computing in two distinct forms (inter and intra block parallelism). We compared these approaches with the single-thread case, and ultimately uncovered the limitations and potential benefits of parallelism. Our ability to implement parallelism in our cryptographic schema, was made possible by the use of the AES algorithm [Wika] in XTS mode [NIS].

The parallelization of XTS mode depends mainly on an important feature of this mode in which each block of data can be encrypted independently without any relation to other blocks. This feature allows the encrypted data to be divided into different portions in such a way that two successive blocks of data can be processed concurrently in two different processing units [MAAR].

1.4 Outline

The rest of this thesis is structured as follows:

- **Chapter 2:** We provide background information for readers who want to understand the rest of the thesis.
- **Chapter 3:** We present the design of the io_uring mechanism, the ublk framework and the design of our three cryptographic solutions. Also we discuss the relevant to our solutions concepts of LUKS, AES and XTS.
- **Chapter 4:** We present the implementation of our three cryptographic solutions along with the common key setup phase.
- **Chapter 5:** We evaluate our solutions and make comments on their efficiency based on the results.
- **Chapter 6:** We conclude with a summary of our work and with some directions for future work.

2 Background

During our survey on ublk, we encountered a lot of interesting features. In this chapter, we will focus on some of these, which are necessary for understanding the concepts presented in the rest of this thesis.

We will provide a brief overview of the Linux operating system. We will examine the subsystems related to the I/O stack, ranging from applications to disks. This exploration will help us clarify how data flows within the Linux system.

Finally, we have included a chapter on elementary cryptographic concepts and some background mathematics. This foundation will be useful when we will encounter the AES algorithm in the next chapter.

2.1 Linux OS

Linux is an operating system (OS) created by Linus Torvalds, a Finnish software engineer. He began its development as a personal project and made it open source on August 25, 1991, 32 years ago. As Linus stated in his first announcement *«I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386 (486) AT clones...»* [Verb].

Of course it eventually turned out to be a very professional one! Linux is a Unix-like OS and quickly gained the attention of the open source community. Because many developers had access to the source code, they helped Linus rebuild and refactor the kernel and by 1994, Linux kernel version 1.0 was released [Wikg].

Nowadays Linux runs everywhere. From mobile phones to personal computers and from embedded devices to supercomputers. One of the more attractive advantages of Linux is its non-commercial nature: the source code, governed by the GNU General Public License version 2 (GPLv2), is open and accessible for anyone interested in studying it. Along with this, some other Linux's characteristics are:

- **Reliability and Stability:** It can handle heavy workloads and runs for long periods without restarts. This makes it a top choice for crucial systems like servers and supercomputers, ensuring consistent operation even in high-pressure situations.

- **Security:** Its design makes it tougher for malware, setting it apart as a secure choice. It also benefits a lot from open source review to quickly fix vulnerabilities.

- **Flexibility:** It offers limitless customization, allowing everyone to find their fit in the diverse Linux ecosystem. With various distributions, desktop environments, package managers, and more, users can explore differences and personalize their Linux experience.

2.1.1 Operating System vs Kernel

The kernel is the big chunk of executable code in charge of handling all requests from processes asking to access system resources. Technically speaking the kernel is *part of* the operating system, not the whole operating system. It is its core component, and the first program that bootloader loads into memory. But an operating system consists of other system programs that are not part of the kernel. Typically, an OS is shipped alongside a Graphical User Interface (GUI), a compiler, a package manager, a command line interface (e.g. bash), system libraries and many more. A kernel without these interfaces would be useless in most cases, and of course all these system programs interact with the kernel to provide to the end users the desired result.

Note: From now on, we will use the terms "kernel" and "operating system" interchangeably. As previously noted, the kernel constitutes the fundamental element of the OS, and this interchangeable usage is a commonly accepted convention.

2.1.2 And...what a kernel does?

Although the distinction between the different kernel tasks is not always clearly marked, the kernel's role can be split (as shown in Figure 2.1) into the following parts [RKH05]:

- **Process management:** Manages process creation, communication, and scheduling, enabling multiple processes to run efficiently on one or more CPUs.
- **Memory management:** Creates virtual address space for each process, coordinating memory management through a range of function calls, from basic allocation to more complex ones.
- **Filesystems:** Linux treats almost everything as a file. It builds a structure filesystem atop hardware, while it supports various filesystem types for organizing data on physical volumes.
- **Device control:** Nearly all system operations correspond to physical devices, and except for a few core components (like memory and CPU), specific device-related code, called "device driver", handles device control. These drivers are part of the kernel.
- **Networking:** Network activity is managed by the OS. It collects, identifies and dispatches outcome and income packets, serving the needs of all processes wanting to access a network.

2.1.3 user mode vs kernel mode

A process is a program that is loaded into the system's memory and executed. In this context "execute" means that the Central Processing Unit (CPU) fetches instructions from a memory region specific to the process and carries out with their execution.

Typically each CPU architecture incorporates multiple protection layers known as **privilege rings**. Each process's instruction is executed within one of these protection rings, having its own resource access rights. The innermost ring has the highest privilege, while the outer ones have progressively fewer privileges. For instance the x86 architecture, employs a 4-ring protection layer schema as depicted in Figure 2.2.

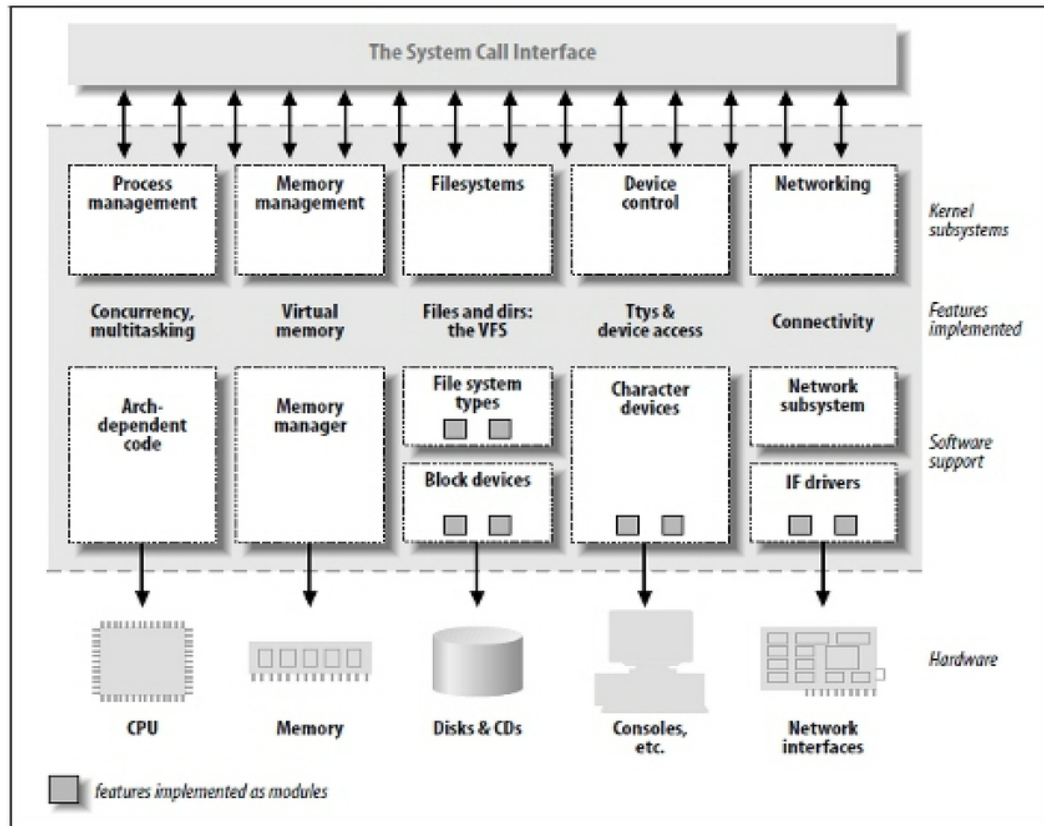

Figure 2.1: *A High-Level View of the Kernel*

Modern CPUs offer at least two distinct *execution modes*: a **nonprivileged mode** for the user and system programs, and a **privileged mode** for the kernel. These modes are referred to as **user mode** and **kernel mode** (also known as **supervisor mode**). User mode corresponds to the outer ring, while kernel mode corresponds to the inner ring. This design enables CPU to differentiate between different level of privileges, thereby preventing users from executing commands that exceed their permissions.

Linux adopts a two-ring protection model: Ring 0 signifies kernel mode, and Ring 3 signifies user mode. Rings 1 and 2 are not used.

2.1.4 userspace vs kernelspace

Userspace denotes the memory space where user applications and operating system-related processes execute. Everything apart from the kernel runs there. Userspace processes can be divided into **system processes**, executing system-related code, and **user processes**, executing user code.

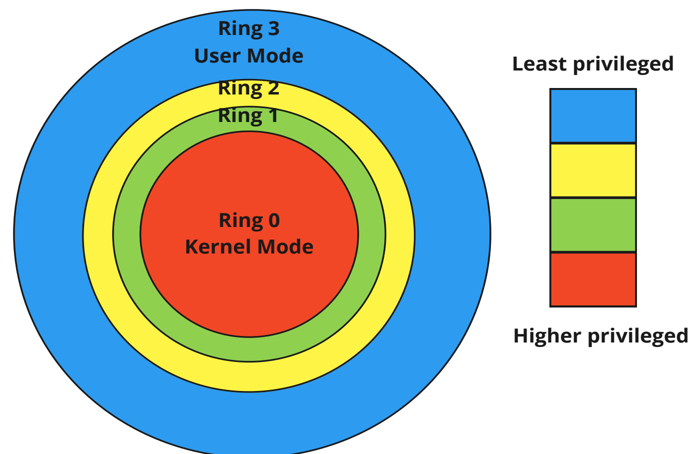

Figure 2.2: *x86 Architecture: Privilege Rings*

On the other hand, kernelspace refers to the memory space in which the kernel operates.

To make the distinction clearer: userspace and kernelspace relate to the **memory** where processes operate, while kernel mode and user mode refer to the **CPU's operational mode** when executing instructions.

Consider a simple "Hello World" program like the one shown in Listing 2.1:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello, World!\n");
    return 0;
}
```

Listing 2.1: *Hello World!*

Let's see what occurs when a user runs the executable generated from the program in Listing 2.1:

1. The OS loads the program into memory and initiates a new process to execute the `hello_world` program. This process operates in userspace, and its instructions run in user mode on the CPU.

2. When the instruction pointer (IP) reaches the `printf()` command, it actually executes code from a library function in the `glibc's` library. This library function

is essentially a wrapper around the `write()` system call. The process remains in userspace.

3. As the `write()` system call begins, the kernel is invoked to fulfill the corresponding request. At this point, the execution shifts from userspace to kernelspace, and the CPU's instruction execution is in kernel mode.

4. Once the kernel completes the system call, it returns a status code indicating the result. Control is then handed back to userspace, enabling the execution of the next instructions in the program (in user mode).

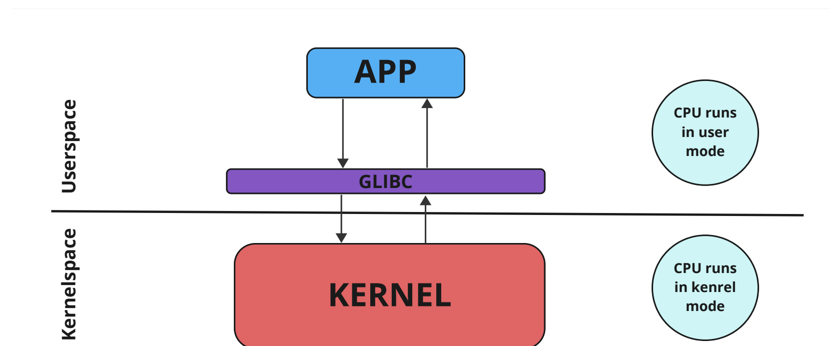

Figure 2.3: *Userspace vs Kernelspace*

Note: In the rest of this thesis, the terms "kernelspace" and "kernel mode" will be used interchangeably to signify the kernel's operation. Similarly, the terms "userspace" and "user mode" will denote all other cases.

2.2 Kernel Architecture

2.2.1 Monolithic vs Microkernel

Linux kernel like most of the Unix kernels, is **monolithic**: each kernel layer is integrated into the whole kernel program and runs in kernel mode on behalf of the current process [BC06]. In other words monolithic is a kernel that there is no protection between the various kernel subsystems and where public functions can be directly called between various subsystems [Teab].

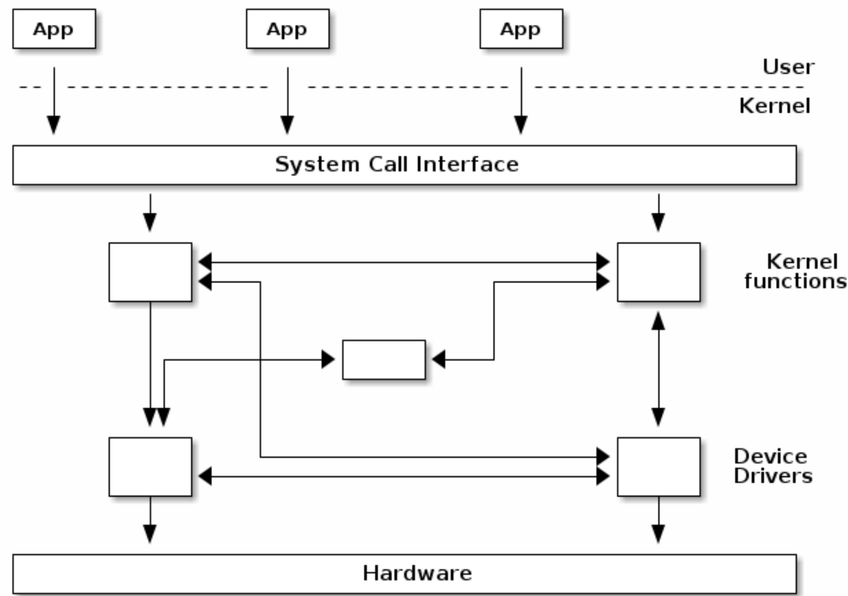

Figure 2.4: *Monolithic Architecture*

In contrast, there exists another kernel design, named **microkernel**. This is a minimalist design approach, where the kernel is kept as small and simple as possible. In a microkernel system, the kernel only includes the most essential functions. Additional functionality, is moved out of the kernel and into separate userspace modules, known as **servers** or **services**. Because significant parts of the kernel are now running in user mode, the remaining code that runs in kernel mode is significantly smaller, hence microkernel term.

In a microkernel architecture the kernel contains just enough code that allows for message passing between different running processes. Essentially, that means implementing an inter-process communication (IPC) mechanism in the kernel, as well as some functionality on memory management to achieve the protection between applications and services.

Both designs have their advantages and disadvantages. It is obvious that microkernels isolate the different subsystems to a greater extend. As a result, if a service crashes, we can just restart it without affecting the whole system. Bugs in one service can't affect other services. On the other hand, such operating systems are generally slower than monolithic ones, because the explicit message passing between the different layers of the operating system incurs an overhead. What is a simple function call between two sub-

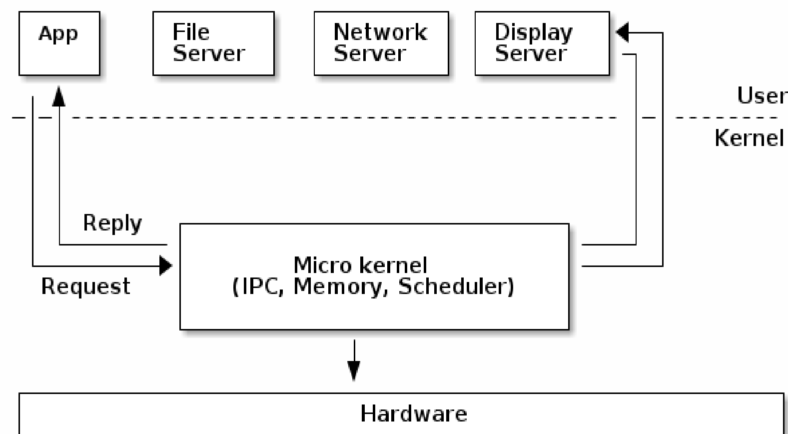

Figure 2.5: *Microkernel Architecture*

systems on monolithic kernels, now requires going through IPC and scheduling which of course, comes at a cost.

Kernel modules

In order to attain many of the theoretical benefits associated with microkernels, while avoiding potential performance drawbacks, the Linux kernel uses a feature known as "modules". A module is essentially an object file containing code that can be dynamically linked and unlinked from the kernel during runtime, via the programs `insmod` and `rmmod` respectively. This object code typically comprises a set of functions that implement what is known as a **"device driver"**. The difference with the microkernel-based operating systems, is that modules do not operate as distinct processes, instead, they run within kernel mode on behalf of the ongoing process, like any other statically linked kernel function.

The main advantages of using modules include:

- Logically organize the kernel in subsystems.
- Save main memory, by loading a module only when it is needed and unloaded afterwards.
- Force developers to adopt a modularized approach on their design.

2.2.2 Device Drivers

Linux categorizes devices into three fundamental types: **character devices**, **block devices** and **network devices**. Each kernel module, usually implements one of these types and hence we classify them as: **character modules**, **block modules** and **network modules**. Of course this classification isn't strict, and developers can choose to combine different types of device drivers, into a single module, although it is highly recommended to create separate modules for different functionalities.

More specifically the three classes are:

1. **Character devices:** Devices that provide *sequential* access of any I/O size down to a single character. The corresponding driver, usually implements at least the `open()`, `close()`, `read()` and `write()` system calls. Such devices include keyboards and serial ports.

2. **Block devices:** Devices that perform I/O in units of blocks (usually in 512 byte chunks named *sectors* or a larger power of two). These blocks can be accessed randomly based on their block offset, which begins at 0 at the start of the block device [Gre20]. Kernel uses the corresponding driver to implement two main functionalities: «*read block with number N and write its data in memory pointed to by buf*»and «*retrieve the data stored at buf and write them at block N*». Although block drivers have a completely different interface *inside* the kernel, Linux enables applications to treat block devices similarly to char devices, presenting a transparent interface for both of them. The difference between them is managed internally by the kernel.

3. **Network devices:** Peripheral devices that can send and receive data packets to and from other computer systems. This category includes physical network cards (like `eth0`, `eth1` for the first and second Ethernet adapters) as well as virtual network adapters like `lo` for packets sent to the same machine.

In this thesis, we didn't come across any network device drivers. However, we encountered both character and block device drivers. Therefore, let's examine them more closely.

Access to devices via special files

As stated earlier, users interact transparently to both block and character devices. The kernel exposes those devices in a special directory named `/dev`. So, on the one hand we have the device driver which provides the communication on behalf of a user program and on the other hand we have `/dev` which represents each piece of hardware in the system.

For example, imagine that you have a solid-state drive (SSD) [Wikh]. This is a block device and the kernel will expose this device in `/dev`. Let's assume that it assigns to it the name `/dev/sda`. Now userspace utilities can use `/dev/sda` without ever knowing what kind of disk they are communicating with. This is a typical example of abstraction in the Linux kernel. Subsystems offer a standard API to interact with, hiding the complexities of their internal workings and allowing other utilities to interact with them in a uniform way.

Let's take a look at some device files:

```
$ ls -l /dev/
...
brw-rw----  1 root disk        8,   0 Aυγ  29 19:16 sda
brw-rw----  1 root disk        8,   1 Aυγ  29 19:16 sda1
brw-rw----  1 root disk        8,   2 Aυγ  29 19:16 sda2
...
crw-rw-rw-  1 root tty         5,   0 Aυγ  30 01:30 tty
crw--w----  1 root tty         4,   0 Aυγ  29 19:16 tty0
...
```

Listing 2.2: *List Device Files*

The first character of the output line, denotes whether it is a block device (`b`), or a character device (`c`).

Two columns of numbers separated by a comma are present. The first number is called the **major number** of the device, and the second the **minor number**. The *major number* tells which driver is used to access the hardware. What type of device is used. Each driver has a unique major number. The *minor number* is used by the driver to distinguish between the various hardware it controls. It identifies a specific device among a group of devices that share the same major number. For instance, in Listing 2.2, we a have a

group of three devices managed by the same disk controller that have the same major number (8) and different minor numbers (0, 1 and 2).

Note: When a device file (e.g. `/dev/sda1`) is accessed, the kernel uses the major number to determine which driver should be used to handle the access. This means that the kernel doesn't really need to use the minor number. The driver itself is the only entity that cares about the minor number. The kernel uses the major number to find the corresponding driver, and passes the execution to it and then the driver itself uses the minor number to distinguish between different instances of the specific device.

In-kernel representation of Device Files

Both major and minor parts of a device file are stored in a `dev_t` type inside the kernel. Practically the `dev_t` type is an `u32`, that stores in the 12 most significant bits the major number and in the 20 least significant bits the minor number.

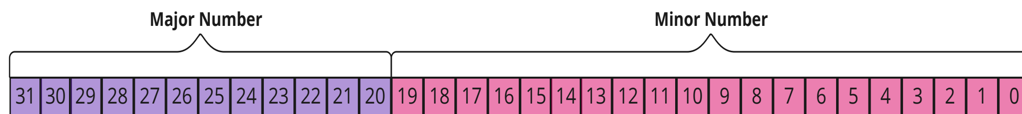

Figure 2.6: *Major and Minor Parts of dev_t Type*

Of course the internal representation of `dev_t` may change in the future, and that's why to obtain the major and minor numbers of a `dev`, the macros `MAJOR(dev)` and `MINOR(dev)` respectively should be used (can be found in `linux/kdev_t.h`).

If instead, you have the major and minor numbers and need to turn them into a `dev_t`, use the macro: `MKDEV(int major, int minor)`.

2.2.3 A dive into /dev directory

Let's take a closer look at `/dev` directory. If we list the status of all currently mounted file systems, we can obtain useful information:

```
$ grep -w dev /proc/mounts
dev /dev devtmpfs rw,nosuid,relatime,size=4007368k,nr_inodes=1001842,mode
    =755,inode64 0 0
```

Listing 2.3: *List /proc/mounts*

The file /proc/mounts actually is not a real file, but part of the virtual file system that represents the status of mounted objects as reported by the Linux kernel. The format is similar to fstab: the system's name, mount point, file system type, etc [Dia].

As we see, in /dev directory there is mounted a filesystem named devtmpfs. devtmpfs replaced devfs and merged in the Linux kernel in version 2.6.34. [LWNc, LWNa, LWNb]

devtmpfs is a special filesystem that the **kernel creates and populates** to expose information about devices.

And who is responsible for mounting devtmpfs in /dev directory?
If CONFIG_DEVTMPFS_MOUNT is set to y (in .config file) when building the kernel, the resulting kernel will automatically attempt to mount devtmpfs to /dev after mounting the root filesystem [Sta].

It can also be done with a rule in /etc/fstab like:

```
$ mount -t devtmpfs junk /dev
```

Although the kernel is responsible for populating devtmpfs filesystem, it provides userspace with the opportunity to process device events further through the "udev" subsystem, which matches events with rules and triggers additional actions based on them.

More specifically, whenever a device is added or removed from the system (or change its state) kernel sends messages, named uevents to userspace. Kernel sends these messages via "*netlink*". Netlink is a socket-like mechanism used in Linux to pass information between kernel and user processes [Dwe]. This mechanism gives the opportunity to userspace, to manipulate further the changes in the system.

Udev subsystem uses a userspace daemon which is now part of systemd, named systemd-udevd. This daemon is responsible to catch the user events (uevents) that kernel sends. Then udev, tries to match these messages with some rules it maintains. These rules can be found in /usr/lib/udev/rules.d, /usr/local/lib/udev/rules.d, /run/udev/rules.d and /etc/udev/rules.d [mpg].

And if it manages to match the uevent with a rule, then it may trigger additional processing based on the specific rule.

Note: The information above applies to both character and block devices.

Now, let's turn our attention specifically to character device drivers.

2.2.4 Character Device Drivers

In the kernel, a character device is represented by `struct cdev`, a structure used to register the driver in the system. A definition of this structure can be found in `/include/linux/cdev.h`.

To complete the addition of a character device to the system, we need to follow three distinct steps:

1. Firstly, we should register a range of device numbers, which can be done either **statically** (if we know the device's major number,) or **dynamically** (if we let the kernel pick one for us).

2. Secondly, we should initialize the data structure `struct cdev` for our character device.

3. Thirdly, once we finish the initialization, we can add the character device to the system. In this step practically we notify the kernel for the driver via associating the character device initialized in step 2, with the range of device numbers registered in step 1.

```
// Statically register a range of device numbers
int register_chrdev_region(dev_t from, unsigned count, const char *name)

 // Dynamically allocate a range of char device numbers.
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count,
      const char *name)

// Initialize a cdev structure
void cdev_init(struct cdev *cdev, const struct file_operations *fops)

// Add a char device to the system
int cdev_add(struct cdev *cdev, dev_t dev, unsigned count)
```

Listing 2.4: *Register a Character Device*

Of course, once we have finished with the device driver, we need to delete and unregister it from the system, allowing the kernel to reclaim resources. To do so, we can use the following functions:

```
// Unregister a range of device numbers
void unregister_chrdev_region(dev_t first, unsigned int count)

// Remove a character device from the system
void cdev_del(struct cdev *dev);
```

Listing 2.5: *Unregister a Character Device*

Device drivers are loaded and unloaded as modules. We explained in Section 2.2.1 the advantages of modularization. The program that loads a module is called `insmod`, while the program that unloads it is called `rmmod`.

Each of these programs triggers a specific function within the device driver. `insmod`, invokes a function declared with the macro `module_init()`, while `rmmod`, triggers a function declared with the macro `module_exit()`. Usually, inside these functions occurs the registration phase of Listing 2.4, and the unregistration phase of Listing 2.5 respectively.

Data structures for character device drivers

As Listing 2.4 shows, initializing a character device structure, with `cdev_init()`, demands the use of a structure named `file_operations`. This structure along with the structures `file` and `inode` are very important kernel data structures and commonly used by many character devices.

struct file_operations

The `file_operations` structure, holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to system calls made by users over device type files.

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

```
6   ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
7   ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
8   long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
9   int (*mmap) (struct file *, struct vm_area_struct *);
10  int (*open) (struct inode *, struct file *);
11  ...
12  int (*uring_cmd)(struct io_uring_cmd *ioucmd, unsigned int issue_flags);
13  int (*uring_cmd_iopoll)(struct io_uring_cmd *, struct io_comp_batch *,
14        unsigned int poll_flags);
15 };
```

Listing 2.6: *struct file_operations*

Thus, implementing a character device driver means defining the system calls `open()`, `close()`, `read()`, `write()`, etc, for the specific device. The driver of course, may not implement some of the above operations (i.e. system calls), because it may not support (or need) the specific functionality. Usually we use the C99 way of assigning elements to the `file_operations` structure [989]. Structure members who do not explicitly appear in this initialization will be set to `NULL`.

```
1 struct file_operations my_functions {
2   .read = my_read,
3   .write = my_write,
4   .mmap = my_mmap,
5   .open = my_open,
6   .release = my_release,
7 };
```

struct file

The `struct file`, represents an **open** file. This structure is not specific to device drivers. The kernel itself is responsible of allocating a `struct file` each time an `open()` system call is called. All open files have an associated `file` structure.

struct inode

This structure, represents a file **on the disk**. It's used by the kernel internally. There can be numerous `file` structures representing multiple open descriptors on a single file, but

they all point to a single `inode` structure.

As [Teaa] aptly stated: «*To understand the differences between* ***inode*** *and* ***file****, we will use an analogy from object-oriented programming: if we consider a class inode, then the files are objects, that is, instances of the inode class. Inode represents the static image of the file (the inode has no state), while the file represents the dynamic image of the file (the file has state).*»

2.2.5 Miscellaneous Device Drivers

Misc (or miscellaneous) drivers **are simple character drivers** that share certain common characteristics. The kernel abstracts these commonalities into an API (implemented in `drivers/char/misc.c`), and this simplifies the way these drivers are initialized. All misc devices are assigned a major number of 10, but each can choose a single minor number [Ven08].

A misc driver accomplishes all of the initialization steps of a character device (`alloc_chrdev_region()`, `cdev_init()`, `cdev_add()` see 2.2.4) with a single call to `misc_register()`. This function registers a `miscdevice` structure, which is the representation of the misc device in the kernel (the same as `cdev` in character devices).

Block Device Drivers

Block device drivers are discussed in Section 2.4.5 because we believe they would be better understood if we provided an overview of the block layer before delving into them.

2.3 Disks

Throughout this thesis, we came across the notion of "disks" quite often. Although ublk framework is handling virtual disks (see Section 1.2), we still encountered disk-related terms and features frequently. Virtual devices are still manipulated like regular disks after all. In this section, we'll explore some concepts that will come in handy later on [Gre20].

With the term **disk** we refer to the primary storage device of the system. There are a lot of disk types like Optical Discs, Floppy Disks, Hard Disk Drives (HDDs), the faster flash memory-based solid-state disks (SSDs) etc. Generally, there's a trade-off between storage speed and value or endurance. Faster options tend to be more expensive or less durable.

Illustration of a simple disk

Disks are asynchronous in nature (more on asynchronous API in 3.1.2). They follow an interrupt-driven approach to notify the CPU with their response. The kernel performs an I/O and the disk responds with an interrupt when it is ready. An **interrupt** is a signal sent by another device to the CPU. Then CPU typically stops whatever it does at this moment, acknowledges the interrupt and passes the control to the operating system to serve it.

At this point let's **clarify what an I/O is**: it is an abbreviation for Input/Output and refers to disk reads and writes. To describe an I/O, we need to include at least:

1. The type of the I/O (read or write).
2. The size of the I/O (how many sectors).
3. The starting sector.

Due to their asynchronous nature, disks often include some kind of buffering. This type of buffering is one (or more) in-disk queues.

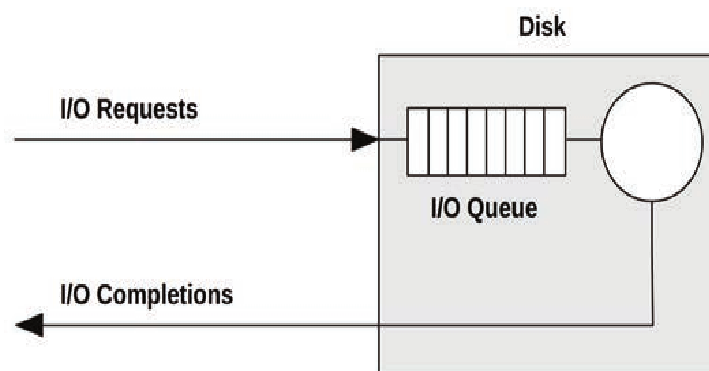

Figure 2.7: *Simple Disk with Queue*

The I/O operations that the disk handles can either be waiting in a queue or currently being processed. Although it might seem like a first-come, first-served queue, disk controllers can employ various algorithms to enhance performance.

Recent disks, usually ship with an in-disk cache. This cache is used by the controller to finish its job faster. When a read request comes to the device, the controller checks the cache if the desired data is stored. If it is it replies immediately. If it's not, then the regular procedure will be followed. In case of a write request, this cache can be used as a *write-back* cache. This means that the write is considered as completed as soon as the data transfer to cache and before the slower transfer to persistent disk storage. This new caching schema enhances the disk's performance because this cache is a faster memory (usually DRAM) and can reply to requests significantly faster.

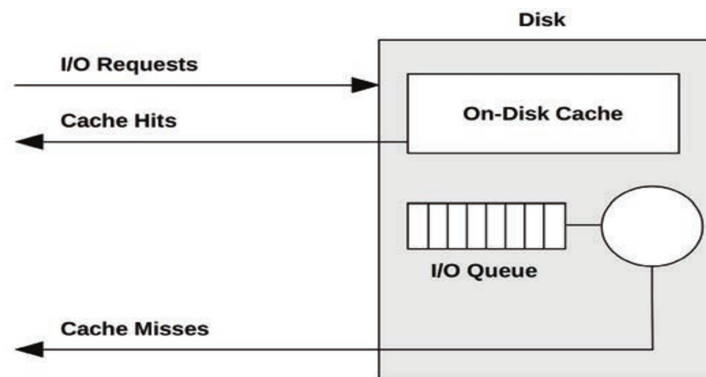

Figure 2.8: *Simple Disk with Queue and Cache*

2.3.1 Time related concepts

An essential aspect when evaluating disks is their responsiveness to requests. Throughout the evolution of technology, disks have remained an integral part of this progression, continually adapting to new demands and requirements. Therefore, it holds significant importance to measure their capabilities, both in terms of response time and throughput. To establish a foundation, let's begin by defining some fundamental terminology:

Throughput: With disks, throughput commonly refers to the current data transfer rate, measured in bytes per second (Bps).

Bandwidth: Signifies the maximum achievable data transfer rate for storage transports.

I/O latency: Indicates the time duration required for an I/O operation to complete, encompassing the entire process from initiation to conclusion.

Measuring time

Let's take a closer look to **I/O latency**. We can split this time, into two phases:

1. **I/O wait time:** This interval represents the time a request spends in a queue, awaiting execution.

2. **I/O service time:** This is the actual time it took to process the request.

The summation of these two times, is the entire time from issuing an I/O to it's completion. This time is called **I/O request** or **I/O response time**.

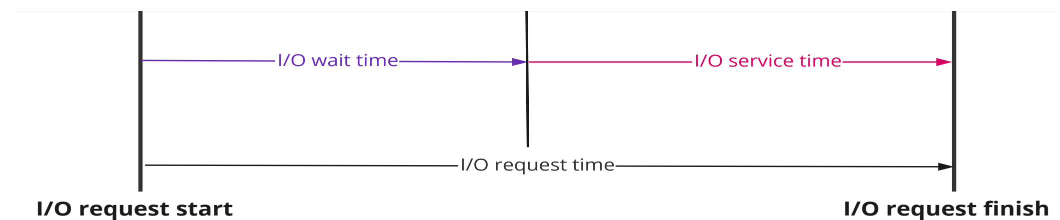

Figure 2.9: *Time-related Terminology*

Of course, when talking about I/O time, it's really important to specify which starting and ending events we're considering. This matters because it's not just the disk involved. The operating system plays a role too. To break it down, there are two main categories of events: those happening in the kernel (**kernel-based events**) and those related to the disk (**disk-based events**). When an application starts an I/O operation, the operating system forms the final stage of this request in a subsystem named **block layer** (more detailed information about block layer in Section 2.4.4). From there on, we start measuring the time a request spends in the kernel.

This exploration leads us to distinct time components that collectively constitute the I/O time. On one hand, there exists the *Block I/O wait time*, denoting the duration a request resides in in-kernel queues, the *Block I/O service time* measured from from the time kernel issues the request to the device until the completion interrupt from the device and finally the summation of these two times that gives us the overall *Block I/O request time*.

Equally, a parallel set of components characterizes disk I/O, encompassing *disk I/O wait time, disk I/O service time, and disk I/O request time.*

Naturally, these two sets may intersect in certain instances. For instance, the *Block service time* corresponds to the time spent on the disk, which coincides with the *disk I/O request time*.

Consequently, I/O latency may refer to block I/O request time (the entire I/O time), or to the entire time spent on device (disk I/O request time).

The Figure 2.10 below, referenced from [Gre20], effectively summarizes the concepts we discussed:

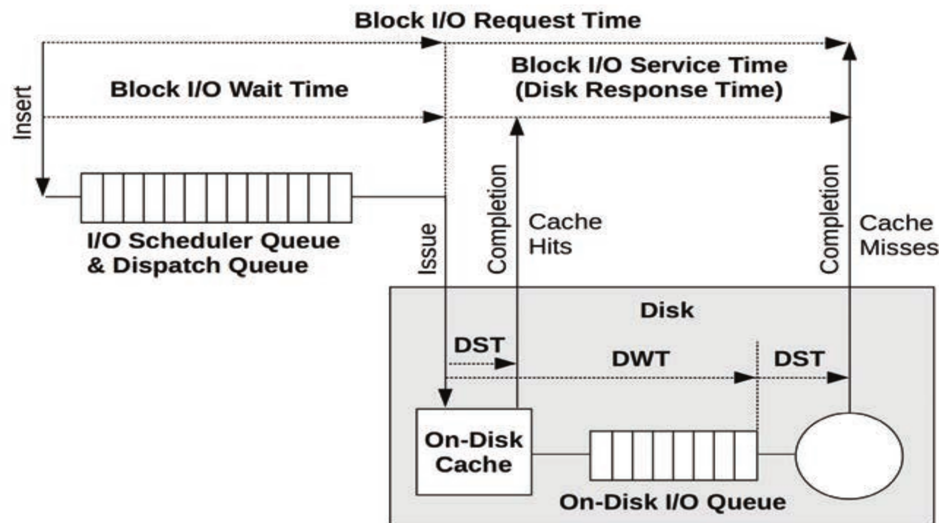

Figure 2.10: *Kernel and Disk Time Terminology*

2.3.2 Disk Types

The two most common disk types that are being used in industry are magnetic disks (HDDs) and solid-state drives (SSDs). Let's take a brief look at both of them and define some basic terminology that we'll use throughout the thesis.

Hard Disk Drives

HDDs are magnetic rotational disks, that are made of one or more **platters**. Each platter consists of a number of **tracks**, and each track is divided into **sectors**. The platters rotate, while a mechanical **arm**, with circuitry to read and write data from a **head**, reaches across the surface.

Sector is the smallest block of storage on a disk. Traditionally it was 512-bytes in size, but today they are often 4K in size (or another power of 2). Sectors should be considered as the **basic unit of data transfer**. It is never possible to transfer less than one sector.

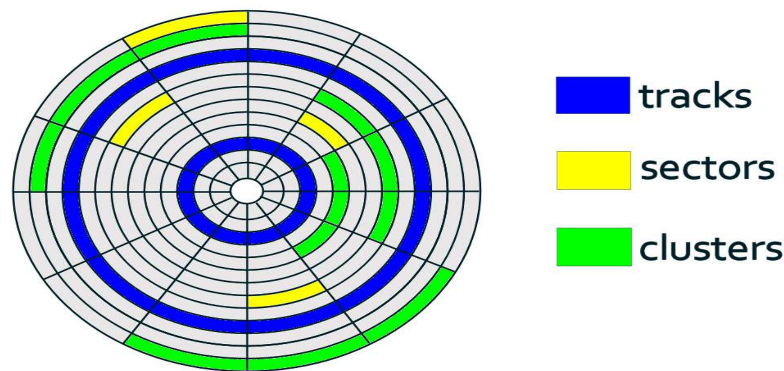

Figure 2.11: *HDD's Anatomy*

While the sector is the basic unit of data transfer for the hardware devices, the **block** is the basic unit of data transfer for the filesystems. Each block must include an integral number of sectors, thus it must be multiple or equal to the sector size. In other words the kernel requests are in magnitude of a block, and the device translates these requests into sectors.

This correlation is shown in the Figure 2.11, with **clusters**. Clusters are the smallest unit of storage from filesystem's perspective (i.e. *block* in kernel's terminology).

Due to the mechanical part of the HDD, and the rotation of the platter, this storage medium, is slow and it consists of three parts:

1. **Seek time:** The time taken by the head to reach the desired track from its current position.
2. **Rotational latency:** Time is taken by the sector to come under the head.
3. **Data transfer time:** The actual time to read/write the data, which depends upon the rotational speed.

Total time is the aggregation of these three times. Because of the peculiarities of HDDs a lot of effort has been put into inventing a new type of storage, free of these symptoms.

Solid-State Drives

Solid-state drives are storage devices that mostly use as storing medium NAND flash memory [Wike]. This type of disks have faster data access compared to HDDs, lower

power consumption, and due to the lack of moving parts they are also physically durable.

We won't go into much details in the internal structures of SSDs. We will just refer to some core components of it, to see how they differ from HDDs.

A flush-based SSD consists of arrays of **memory cells**. Each memory cell can store from 1 to 5 bits at the moment [Tec]. The smallest unit of an SSD is a **page**, which is composed of several memory cells, and is usually 4 KB in size. One page is the smallest structure which can be read or written. Multiple pages form one **block**. We oversimplifying a little bit but someone can think these concepts of NAND flash memory like a grid. Each point in the grid is the **cell**, each row is the **page** and the whole grid is the **block**.

One thing worth mentioning, is that flash memory has asymmetrical read/write performance: **fast reads and slower writes**. This is due to the fact that flash memory cannot **overwrite** a page. The fact that you can read and write in pages but only erase in blocks leads to some odd behavior when compared to traditional storage. A magnetic hard disk can always write wherever it likes and update data "in-place". Flash storage can't. It can (essentially) only write to empty, freshly erased pages.

The most obviously bad side effect of this kind of scheme is that, unless the SSD has an available erased page ready and waiting for data, it can't immediately perform a write. In cases where no erased pages are present, the SSD must identify a block with unused (yet unerased) pages, erase the entire block, and subsequently write out the old contents of the block alongside the new page [Hut].

2.4 I/O Stack

In this section we will analyze the data path from an application in a computing system, to the physical non-volatile memory (disk). More specifically, we will follow a `read()` or `write()` system call, and we will make a brief tour in the various Linux kernel subsystems it passes until the request finally reaches the disk. In sections 2.2.2 and 2.3, actually we touched the 2 last components of this path, the Device Drivers, and the disk itself. Now, we will take a look in the overall picture.

In Figure 2.12, we present a visual representation of the subsystems involved. This mental model should be in our minds when we think about a `read()` or a `write()` system

call. It's important to note that this figure is not exhaustive and represents a high-level overview of the I/O stack.

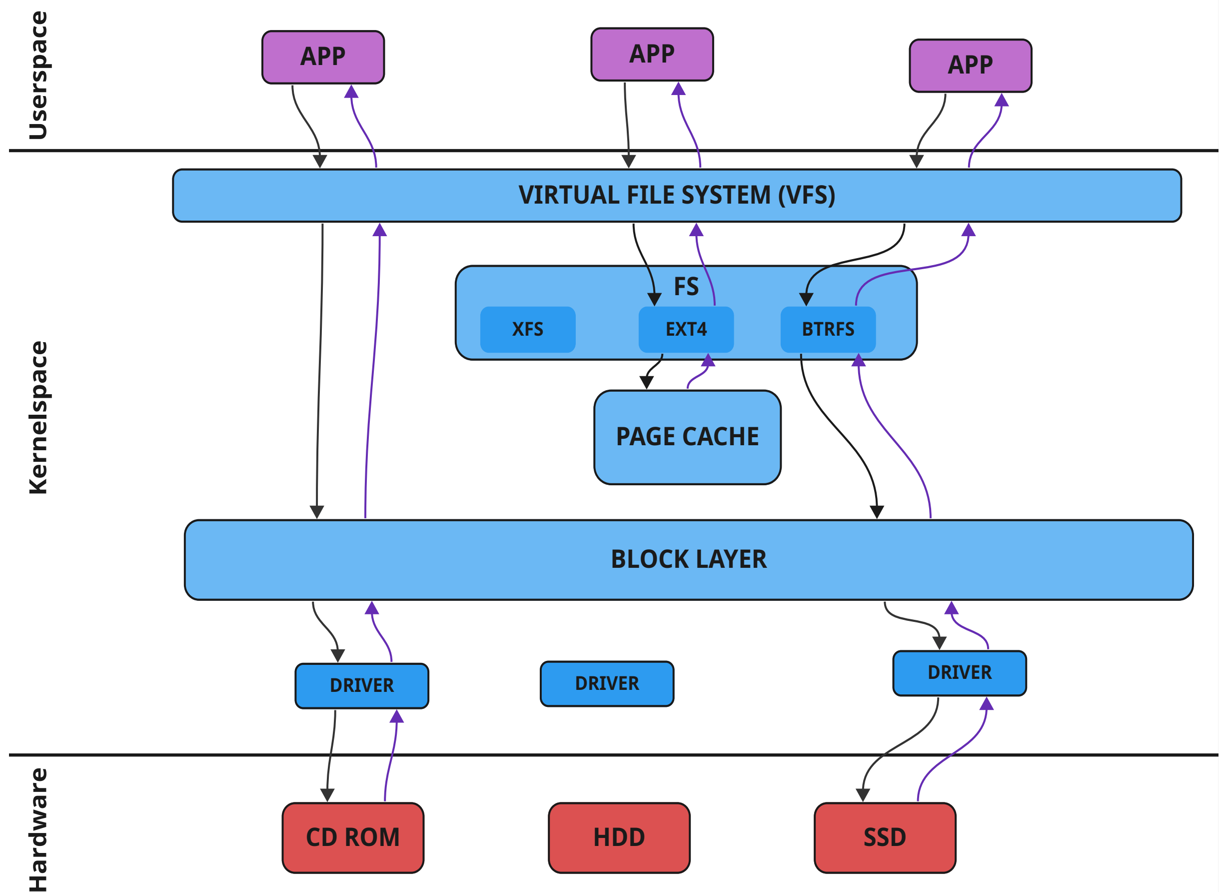

Figure 2.12: *I/O Path*

2.4.1 Application

At the highest level, an application, running as a process in the operating system, interacts with a file through system calls exposed by the VFS, such as open(), read(), write(). To a user application a file is just a linear sequence of bytes, with the ability to access and modify each of them.

2.4.2 Virtual File System (VFS)

In order to support multiple filesystem types Linux implements a large and complex subsystem that deals with filesystem management: the **Virtual File System**. This subsystem serves two main purposes:

- Exposes a consistent API to the applications above it.
- Enforces a **common file model** on the filesystems, beneath it.

Imagine a system that requires users to know precisely what type of filesystem is mounted on the storage device they're interacting with, and based on this type, call a different function. Consider how cumbersome it will be for an application to implement something like this:

```
[...]
fd = open("desired_file");
if (is_filesystem_ext4(fd))
  ext4_read();
else if (is_filesystem_xfs(fd))
  xfs_read();
else if (is_filesystem_vfat(fd))
  vfat_read();
[...]
```

This complexity is hidden from end users (applications) by the common file model. The common file model demands filesystems to implement this consistent API. In other words, it's as if VFS is telling filesystems: «I don't care how you'll implement the `read()` system call, but be aware that I enforce these policies, I'm making this assumptions, and I expect your `read()` implementation to adhere to them.».

For example, as noted in [BC06], in the common file model, each directory is regarded as a file, which contains a list of files and other directories. However, several non-Unix disk-based filesystems use a File Allocation Table (FAT), which stores the position of each file in the directory tree. In these filesystems, directories are not files. To stick to the VFS's common file model, the Linux implementations of such FAT-based filesystems must be able to construct on the fly, when needed, the files corresponding to the directories.

This logic, of hiding complexity from subsystems and enforcing rules for others to follow is a common feature in the Linux kernel.

To achieve this, VFS uses four important objects. Some of these data structures exist solely in memory, and some reside both on disk and in memory:

1. **Supeblock object:** Contains information about the filesystem instance. It exists on disk, and usually also in memory for caching purposes. Essentially it is the filesystem's metadata.

2. **Inode object:** This object uniquely identifies a file on disk (see also 2.2.4). It exists on disk and in memory for caching purposes. While superblock contains filesystem metadata, an inode contains metadata for a specific file, including information such as *file type*, the *file size*, *access rights* etc.

3. **File object:** This object exists solely in memory and corresponds to an open file (see also section 2.2.4).

4. **Dentry object:** This object, is a memory entity (similar to a file) and associates a *name* with an *inode* (i.e. a name with a corresponding file on disk). It is the *glue* that holds inodes and files together.

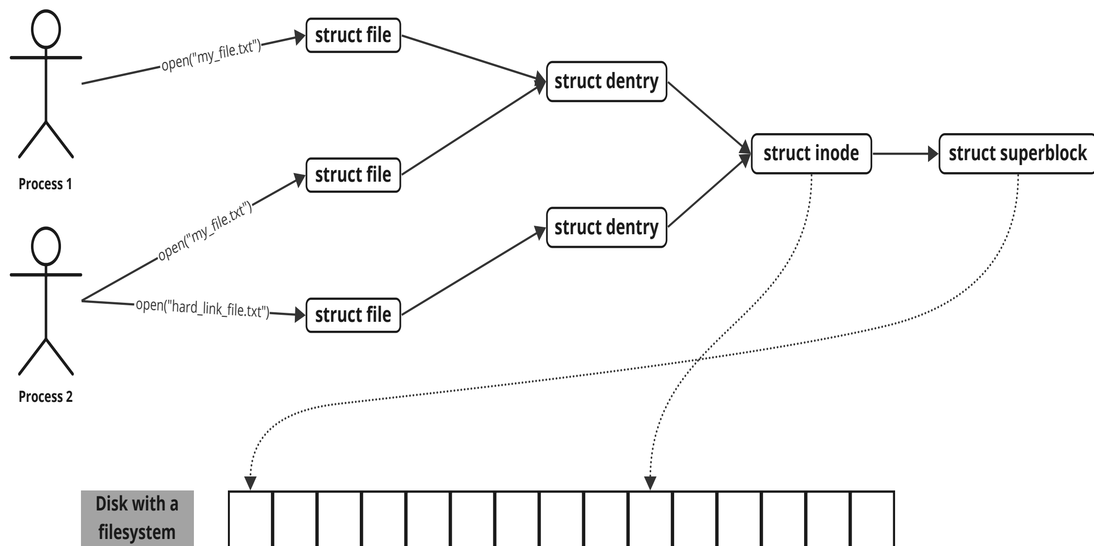

Figure 2.13: *From Processes to VFS Objects*

Figure 2.13 illustrates the relationships between these objects.

Process 1 opens a file named `my_file.txt`, and process 2 opens the same file `my_file.txt` and a hard link [Wikf] to this file named `hard_link_file.txt`. The kernel creates three struct file objects associating each of them with an open operation. Two of these objects point to the same struct dentry, because they were opened using the same pathname. Both of these dentry structures point to the same struct inode, because they refer to

the same file on disk. The Figure 2.13 also demonstrates that the superblock is typically stored at the beginning of the filesystem, (and because it is critical, copies of it exist in other places as well).

2.4.3 Filesystem (FS)

The filesystem is a special data hierarchical storage structure, which contains files, directories and related control information. It provides a file-based interface for access and file permissions to control the access. As mentioned previously each compatible filesystem, needs to implement various functions defined from VFS.

Filesystems are also responsible for a very important type of cache: **the page cache**. Page cache is a region in memory that is used from filesystems to save pages that the application accessed (or may access) in order to speed up I/O operations. New pages are added to the page cache to fulfill **read requests** from processes. If the page is not already in the cache, a new entry is added, and it is populated with the data read from the disk. If the page is there, then it returns with the result without the need to reach the disk and block the calling process . For **write requests**, filesystems can use the page cache as a write-back cache. This will improve the speed of the write because it will treat it as *completed* after data reach main memory and will postpone the "real", costly write to disk sometime later, *asynchronously*.

An application can bypass the page cache by specifying the `O_DIRECT` flag in the `open()` system call. According to this flag, the file system communicates directly with the generic block layer, and data is transferred to or from the disk directly from the userspace buffer, without first being stored in pages of the page cache. This can be ideal for applications that manages some form of temporary storage and does not wish to rely on the kernel's caching mechanism, like databases.

Of course, this behavior creates an *inconsistency* in our system. What if an application (app1) writes to a page in the page cache and another application (app2) tries to read the same page while having the file open with the `O_DIRECT` flag? For this reason, the kernel marks each updated page in the page cache as *dirty*. Dirty means that the data stored on a page in the page cache differs from the corresponding "block" on the disk. The kernel detects that there is an updated version of the page that app2 wants to read

and instructs a kernel thread to *flush* the page cache to disk so that app2 can now read the correct data.

A `read()` from a file opened with the `O_DIRECT` flag leads to **page cache flush** and a `write()` leads to **page cache invalidation**.

The I/O operation that passes through the page cache is also known as **buffered I/O**, while the I/O operation that is not passing is called **direct I/O**.

Note: The page cache effect is depicted in Fig. 2.12 in the middle path, while the `O_DIRECT` approach is illustrated in the rightmost path. The leftmost path indicates a call to a block device that doesn't have a filesystem.

2.4.4 Block Layer

The block layer glues together all the upper and lower components of the I/O stack. I/O requests targeting block devices, will (most probably) pass through this layer. Just as VFS hides from userspace the complexity of the files, block layer hides from filesystems the complexity of block devices, by enforcing a unified way in order to access device drivers. As previously mentioned this is a common technique followed in Linux kernel.

The bio structure

The fundamental structure governing this layer, starting from kernel version 2.6 onwards, is the "struct bio", which **describes a current I/O operation**. This structure fills the gap between how the kernel allocates memory and how disks operate.

Generally, a read or write operation consists of at least one buffer. The buffer is where the user either waits for data in the case of a read or demands to write to the disk in the case of a write. There are also variants to the classical read and write, like readv or writev system calls [mph], where you specify an array of buffers in each call. This means that when the I/O request reaches the block layer, we can have something like what Fig. 2.14 shows.

Here, we should note that even for the classical read/write where we only specify one buffer, it may not be contiguous in physical memory. So, in general, what Fig. 2.14 illustrates, applies to all read/write family system calls.

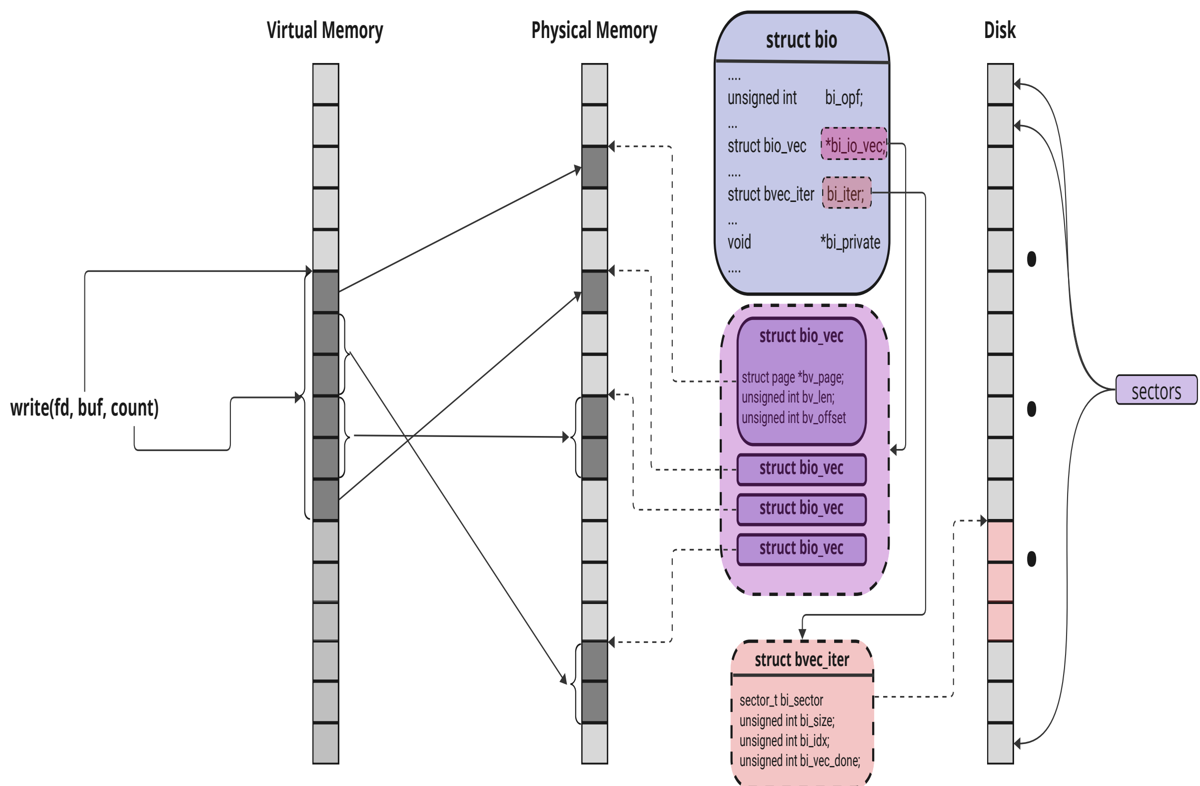

Figure 2.14: *The Role of struct bio*

The struct bio structure contains a `bi_io_vec` vector of struct `bio_vec` structures. A `bio_vec` structure, consists of the individual pages in the physical memory to be transferred, the offset within the page and the size of the buffer. This is the field in bio that «knows what is happening in memory». The struct `bvec_iter` on the other hand «knows what is happening on disk». From which sector this I/O will start, what is the length etc.

Also, the request type (read or write) is encoded in `bi_opf` field of the bio structure.

A bio defines a block I/O operation as a group of consecutive sectors that need to be read or written to dispersed memory segments. But block device drivers do not generally receive bio structures directly; they service block requests received as struct requests, which emerge after the bio structures have undergone processing by the Linux block layer [Kou].

Single-Queue Linux block layer

The block layer has gone through numerous changes over the last decade. The previous design was known as a Single-Queue (SQ) design. In this design, each device had a corresponding queue in the block layer, of type "`struct request_queue`". This structure was made up of requests, of type "`struct request`". Each request consisted of struct bios, and each request described an I/O operation from a location (or locations) in memory to a **physically adjacent** area on the disk. This means that every time we registered a block device driver, the kernel initiated a `request_queue` dedicated to that driver. The block layer was responsible for populating the queue with requests, and the driver was responsible for dispatching them.

The block layer treated the struct request_queue practically as a staging area. Once a request was in the staging area, the block layer could perform I/O scheduling and adjust accounting information before scheduling I/O submissions to the appropriate storage device driver.

At this point, it is useful to refer to a mechanism that the scheduler implements, named **plugging**. In short, plugging is a way to temporarily "lock" the request queue, preventing drivers from dequeuing any requests from it. This allows the scheduler to wait for and merge new bios with pre-existing requests, creating larger requests.

In a world where HDDs were the primary hardware target for disk I/O, the main bottleneck was the seek time of the disk head. Therefore, the primary concern was to hand over to the device as many consecutive sectors as possible in one request.

This mechanism illustrates a general solution in engineering. Many times, when we seek a solution, we may need to perform an action that, at first glance, appears counterproductive to our goal (in this case, introducing a delay via plugging). However, this delay is much smaller than the delay we would have incurred if we had sent random requests to the disk.

Multi-Queue Linux block layer

The single queue architecture described in the previous section was sufficient when the secondary storage device was HDDs. However, the arrival of SSDs, changed the characteristics of I/O and significantly increased the response speed of the disks.

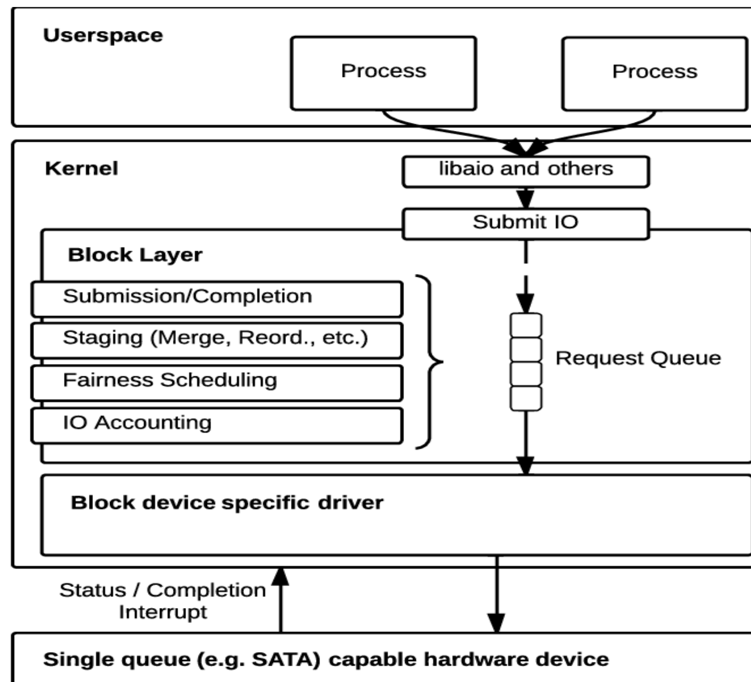

Figure 2.15: *Single-Queue Linux Block Layer Design*

It was observed that the biggest performance bottleneck in I/O appeared in the block layer [MBB]. The previously sufficient single queue architecture was reducing the speed of the requests basically because of the lock contention between different CPUs. As explained, there was one request queue for every block device driver. So if we had different applications running on different CPUs we spent a lot of time competing for the lock of the queue.

This problem was not noticeable when the disks responded slowly. However, when this changed, it became exposed and led many device driver designers to bypass the block layer in order to gain performance.

The new block layer named **blk-mq** was merged in the Linux kernel on 25, October 2013 in 3.17 kernel version [Axba].

The new design of the block layer introduced a two level, Multi-Queue approach. Two level, means that it uses of two **separate** sets of request queues:

1. **Software staging queues:** One per CPU or per NUMA node.
2. **Hardware dispatch queue(s):** The corresponding device driver specifies how many hardware queues supports (but they cannot surpass the number of cores in the system).

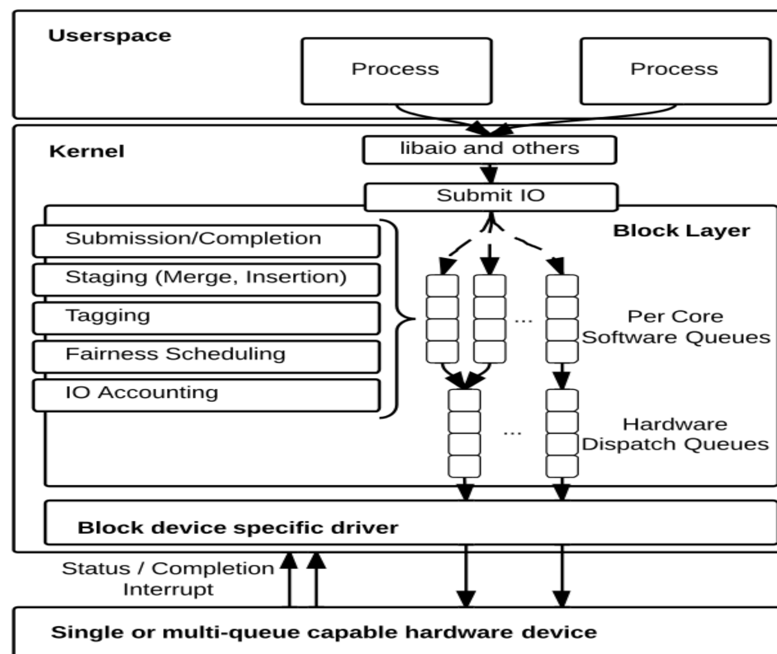

Figure 2.16: *blk_mq Linux Block Layer Design*

There were two main benefits from the blk-mq design: (a) The spread of queues across CPUs obviously led to a reduction in the number of locks required for each request queue. Since each CPU has its own queue, there is no need for other CPUs to compete for the lock. (b) It became clearer which objects the block layer was handling and which the drivers were handling. With the previous design, both the block layer and the drivers shared a common request queue. With the new design, the block layer mainly deals with software queues, while the drivers "watch" the hardware queues. This facilitates the design of the drivers, as it makes the role of each layer in the kernel more clear.

Note: Neither the block layer nor the device protocols guarantee the order of completion of requests. This must be handled by higher layers, like the filesystem [docb].

To facilitate the completions, blk-mq associates each request structure a **tag** number that is unique among requests for that device. This tag accompanies the request throughout its journey, extending into the hardware if supported, and returning in the same manner. Allocating the tag number early ensures a more seamless transit for the request through lower layers, until eventually the block layer releases it.

2.4.5 Block Device Drivers

The kernel must interact with a variety of different hardware devices. Device drivers are the final in-kernel component in the I/O stack, responsible for handling this communication, and they are often provided by the vendors who develop the hardware devices. Specifically, for the I/O stack, we are interested in **block device drivers** because they are responsible for communicating with storage devices.

In the kernel, a disk is represented by a structure called `gendisk`, which contains essential information describing a block device. This information includes the major and minor numbers of the device and its partitions (if any), the device's name, associated disk operations, a queue responsible for managing I/O requests and more.

This queue is the `request_queue` referred to earlier in Section 2.4.4. It servers as the container for the two distinct queues types that make up blk_mq architecture: the "software staging queues" and the "hardware dispatch queues". This linkage illustrates how a block device becomes connected with the queues in the blk_mq architecture.

The allocation of both the `gendisk` and `request_queue` occurs in a single step, performed by the `blk_mq_alloc_disk` function, commonly within the device driver's initialization routine.

It's worth noting that the disk operations embedded in the `gendisk` structure are equivalent to the `file_operations` used for character devices, as explained in Section 2.2.4. These operations are represented as a collection of function pointers, where each function corresponds to an operation on the disk. Here are some of the fields within `block_device_operations`:

```
1
2 struct block_device_operations {
3   void (*submit_bio)(struct bio *bio);
4   int (*poll_bio)(struct bio *bio, struct io_comp_batch *iob,
5       unsigned int flags);
6   int (*open) (struct block_device *, fmode_t);
7   void (*release) (struct gendisk *, fmode_t);
8   int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
9   [...]
10  void (*free_disk)(struct gendisk *disk);
11  [...]
```

```
12 };
```

Listing 2.7: *struct block_device_operations*

Notably, there are no specific `read` or `write` functions within `block_device_operations`. This raises the question of how a block device driver handles reads and writes.

Before drivers allocate the `gendisk` and `request_queue`, they have the opportunity to fine-tune this allocation. In the Linux kernel, there exists a structure called `blk_mq_tag_set`, which stores metadata information about `struct_request`. Using `blk_mq_tag_set`, driver designers can specify essential parameters, including the number of hardware queues and the depth of each queue. Additionally, they can provide a pointer to a structure known as `blk_mq_ops`, which holds various function pointers responsible for defining driver-specific behavior and handling I/O operations.

The `blk_mq_tag_set` structure plays a pivotal role in establishing the connection between these configurable parameters, and the subsequent initialization of both the `gendisk` and the `request_queue`. It serves as the linchpin that configures how the block device driver manages I/O operations.

`blk_mq_tag_set` contains a vital field that stores pointers to functions responsible for implementing block driver behavior. Among these functions, the most significant one is `queue_rq`. This function gets called when the kernel decides that the driver should process I/O requests. It serves as the equivalent of the `read` and `write` functions encountered in character devices. `queue_rq` receives the requests for the device as arguments and can use various functions to process them.

To initialize and add a block driver to the system, the driver typically:

1. Specifies essential information that the disk supports in the `blk_mq_tag_set` structure and allocates it.

2. Initializes a `gendisk` with a corresponding `reqeust_queue`, by passing the previously allocated structure as an argument to the initialization function.

3. Finally, adds the disk to the system by calling the `add_disk()` function. This addition must occur when everything is ready, as the disk may already be active and can receive calls during this function's execution.

From this point forward, the driver is prepared to handle any request by invoking the queue_rq function to process it.

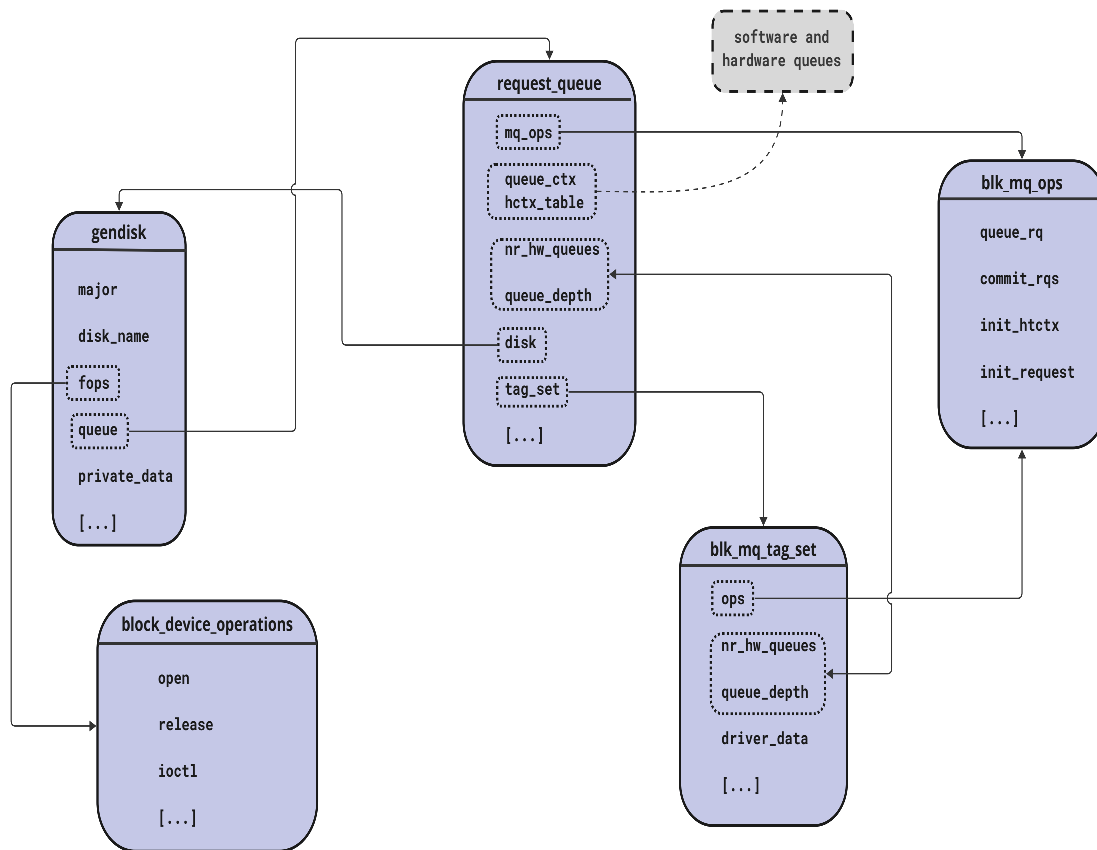

Figure 2.17: *Relation Between Structures on Block Drivers*

2.4.6 Storage Device

The last component of the I/O stack is the disk. It is where the data we want to read/write are stored. We covered disks in Section 2.3.

2.5 Cryptography

We utilized the Advanced Encryption Standard (AES) algorithm to implement the encryption scheme of the ublk framework. In this section, we will present some basic cryptographic knowledge, introduce the AES algorithm, and finally, we will provide

the mathematical background that is important for understanding the internals of the AES algorithm.

2.5.1 Introduction to cryptography

Cryptography is the science of encoding information and transforming it into a format that is unintelligible and useless to any unauthorized party, ensuring secure communication and data protection. The need to hide information and disguise it dates back to ancient times. Many different cryptographic methods were employed by ancient civilizations, including the Egyptians, Greeks, Romans, and others.

In today's digital age, cryptography is often associated with the process of transforming ordinary plaintext into ciphertext. Ciphertext is a form of text that is modified in such a way that only the intended recipient can decode it. In our contemporary world, where every piece of personal information is stored in a database somewhere in the world and can potentially be accessed by others, it is a necessity to be able to protect against malicious users. This is precisely what cryptography aims to achieve.

Cryptography achieves its goals through various means:

- **Confidentiality:** Ensures that data are useless for unauthorized parties.
- **Integrity:** Safeguards the reliability and accuracy of data, preventing tampering and unauthorized alterations.
- **Authentication:** Verifies the identity of parties engaged in communication, ensuring that interactions occur with the correct and trusted entities.
- **Non-repudiation:** Provides evidence that a message was sent or received, preventing individuals from denying their involvement in a transaction.

2.5.2 Symmetric vs Asymmetric Cryptography

We can envision encryption as a black-box operation that takes plaintext as input and produces ciphertext. Alongside the plaintext, this transformation requires a critical component known as a **key**, which is used in the encryption process.

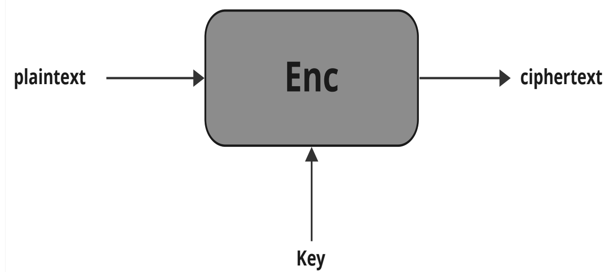

Figure 2.18: *Encryption*

Similarly, decryption is the opposite operation. It functions as a black-box, taking ciphertext as input, along with a key (not necessarily the same key used for encrypting the original message), and producing the plaintext.

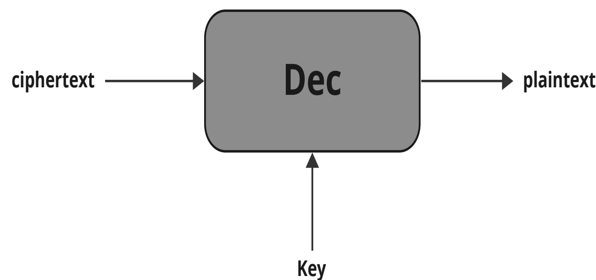

Figure 2.19: *Decryption*

The black-boxes are essentially algorithms. The encryption algorithm can be represented as a function that takes plaintext and a key as input yielding ciphertext. $E(k, p) = c$, where E is the encryption function, k is the key, p is the plaintext and c is the ciphertext. Similarly decryption is denoted as the function $D(k, c) = p$, where k is the key for decryption, c is the ciphertext and p is the plaintext.

According to what key we use to decrypt the ciphertext there are two types of encryption: **symmetric** and **asymmetric** encryption. Symmetric encryption relies on a single shared key for both encryption and decryption. In contrast, asymmetric encryption employs a pair of keys: a public key for encryption and a private key for decryption.

Each method has its own advantages and disadvantages. Let's examine some of their differences:

- Symmetric encryption is generally faster than asymmetric encryption because it requires less computational power. **This makes it well-suited for encrypting large volumes of data quickly**.

- In symmetric encryption, the secure distribution of the shared key is vital since it serves for both encryption and decryption. On the other hand, asymmetric encryption simplifies key distribution by requiring only the sharing of the public key, while the private key remains confidential.
- Symmetric encryption is ideal when transferring large amounts of data within closed systems, because it provides only confidentiality without authentication or non-repudiation. Asymmetric encryption, on the other hand, is often used for secure key exchanges, digital signatures, and authentication in open systems because in addition to confidentiality it can provide both authentication and non-repudiation.

Of course, symmetric and asymmetric encryption can collaborate to create a more comprehensive security solution. For instance, SSL/TLS encryption employs asymmetric encryption to establish a secure session between a client and a server, and then relies on symmetric encryption for exchanging data within the secured session [F5]. This approach allows to leverage the advantages of both encryption methods in a unified security framework.

2.5.3 Introduction to AES

AES is currently the most popular symmetric encryption algorithm in use. It replaced the DES (Data Encryption Standard), after the National Institute of Standards and Technology (NIST) announced in 1997 their intention to select a successor to DES, which they named AES. After three years of searching and evaluating various algorithm submissions, NIST selected the Rijndael algorithm as the final successor of DES, and named it AES. Rijndael was designed by two Belgian cryptographers Vincent Rijmen and Joan Daemen [Wika].

AES is a **block cipher**, that employs symmetric encryption with key sizes of 128, 192 and 256 bits. The larger the key size, the greater the level of security and resistance to brute-force attacks. In essence, as the key size increases, so does the complexity of breaking the encryption, making it more robust and suitable for protecting sensitive information.

Block cipher: is a deterministic algorithm that operates on fixed-length group of bits, called **blocks** [Wikb].

AES divides the plaintext into 128-bit blocks and processes each block individually. A high-level overview of the encryption and decryption processes of AES is depicted in Figures 2.20 and 2.21, respectively.

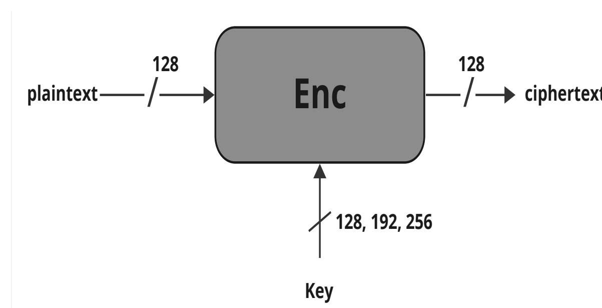

Figure 2.20: *AES Encryption*

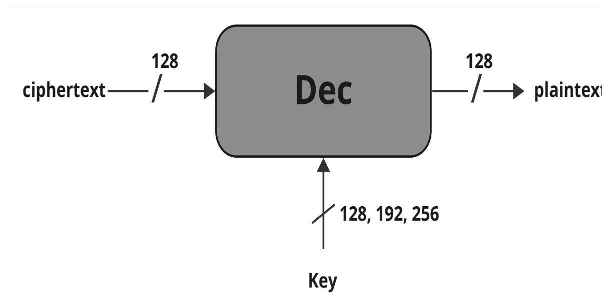

Figure 2.21: *AES Decryption*

The AES algorithm operates in rounds, with the number of rounds varying depending on the key size as Table 2.1 illustrates:

key	# rounds
128	10
192	12
256	14

Table 2.1: *Key Size and Rounds*

To comprehend the AES internals, it is essential to introduce and establish a mathematical foundation. All internal operations of AES are rooted in **Finite Fields**, also known as **Galois Fields**. Therefore, we will dedicate the next section to explore and grasp certain mathematical concepts that are crucial for understanding AES. Of course, those uninterested in these concepts can skip the following section.

2.5.4 Mathematical Background

Before we can understand what a field is, we need to introduce a simpler algebraic structure named **Group**. Theorems and definitions are taken from [CP10].

Definition 2.5.1 (Group)**.** A Group is a set of elements G, together with an operation $\circ$ which combines two elements of G. A Group has the following properties:

1. The Group operation $\circ$ is **closed**. That is $\forall a, b \in G$, it holds that $a \circ b = c \in G$.
2. The Group operation is **associative**. That is, $a \circ (b \circ c) = (a \circ b) \circ c, \forall a, b, c \in G$
3. There is an element $1 \in G$, called the **neutral element** (or identity element), such that $a \circ 1 = 1 \circ a = a, \forall a \in G$.
4. $\forall a \in G$, there exists an element $a^{-1} \in G$, called the **inverse** of a, such that $a \circ a^{-1} = a^{-1} \circ a = 1$.
5. A group G is abelian (or commutative) if, furthermore, $a \circ b = b \circ a, \forall a, b \in G$.

For example:

- For $G = Z$ and $\circ = +$, the $(Z, +)$ forms a Group, with 0 being the neutral element.
- $(Z, -)$ is not a group. For example for $a = 3, b = 2, c = 1 \Rightarrow (a - b) - c = 0 \neq a - (b - c) = 2$, so it isn't associative.
- For $G = Z_m = \{0, 1, ..., m - 1\}$ and the operation addition modulo m forms a group with the neutral element 0. Every element a has an inverse $-a$ such that $a + (-a) \equiv (-a) + a \equiv 0 \pmod{m}$.

Definition 2.5.2 (Field)**.** A field F is a set of elements with the following properties:

1. All elements of F form an **additive group** with the group operation " $+$ " and the neutral element 0.
2. All elements of F except 0 form a **multiplicative group** with the group operation " $\times$ " and the neutral element 1.

3. When the two group operations are mixed, the distributivity law holds, i.e., $\forall a, b, c \in F : a \times (b + c) = (a \times b) + (a \times c)$.

Examples of fields are $\mathbb{R}$, $\mathbb{Q}$ and $\mathbb{C}$.

Fields can form an infinite set (like $\mathbb{R}$), or finite sets. In cryptography we are interested in the latter. These fields are called **Finite Fields (FF)** or **Galois Fields (GF)**.

The following theorem is very important:

Theorem 2.5.1. Finite Fields (FF) only exists if they have p^m elements, where p is prime and m is a positive integer.

From this theorem, we can derive for example that there exists FF with $2, 4, 8, 16, ...$ elements, since 2 is prime and $2 = 2^1, 4 = 2^2$ and so on.

We obtain also, that we can distinguish between two types of GF: Both of them have the form of $GF(p^m)$, where p is a prime, m is a positive integer and p^m are the elements of the field:

1. **Prime Fields:** for $m = 1$.

2. **Extension Fields** for $m > 1$.

Prime Fields $(GF(p))$

The elements of a prime Galois Field, $GF(p)$ are the integers $\{1, 2, 3, ..., p - 1\}$. Arithmetic in $GF(p)$ is done modulo p.

Let $a, b \in GF(p) = \{0, 1, 2, ..., p - 1\}$:
$a + b \equiv c \pmod{p}$
$a - b \equiv d \pmod{p}$
$a \cdot b \equiv e \pmod{p}$
$a \cdot a^{-1} \equiv 1 \pmod{p}$

Note: All conditions of fields are satisfied with these computations.

Extension Fields $(GF(p^m))$

In cryptography we are interested in Extension Fields, with $p = 2$. So we will investigate $GF(2^m)$. The elements of $GF(2^m)$ are **polynomials**, and can be represented as:

$$\boxed{A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + ... + a_1x + a_0}$$

The coefficients $a_i \in GF(2) = \{0, 1\}$. So each a_i is either 0 or 1.

Example: $GF(2^3)$ is a Galois Field with 8 elements. Each element $A(x)$ has the form:

$$A(x) = a_2x^2 + a_1x + a_0, where\ a_2, a_1, a_0 \in \{0, 1\}.$$

The 8 polynomials of $GF(2^3)$ can be represented by the combinations of (a_2, a_1, a_0) in $\{0, 1\}$: For $(a_2, a_1, a_0) = (0, 0, 0) \Rightarrow A(x) = 0$, for $(a_2, a_1, a_0) = (0, 0, 1) \Rightarrow A(x) = 1$, and so on.

$$GF(2^3) = \{0, 1, x, x + 1, x^2, x^2 + 1, x^2 + x, x^2 + x + 1\}$$

□

Addition and Subtraction in $GF(2^m)$

Definition 2.5.3 (Extension field Addition and Subtraction)**.** Let $A(x), B(x) \in GF(2^m)$. The **sum** of the two elements is then computed according to:

$$C(x) = A(x) + B(x) = \sum_{n=0}^{m-1} c_i x^i, c_i \equiv a_i + b_i \equiv a_i - b_i \pmod 2$$

And the **difference** is computed according to to:

$$C(x) = A(x) + B(x) = \sum_{n=0}^{m-1} c_i x^i, c_i \equiv a_i + b_i \equiv a_i - b_i \pmod 2$$

Example: Let $A(x) = x^2 + x$ and $B(x) = x^2 + x + 1$. Then $A(x) + B(x) = (1 + 1)x^2 + (1 + 1)x + (0 + 1) = 0x^2 + 0x + 1 = 1$, because the coefficients are computed modulo 2.

Note: Subtraction and addition in $GF(2^m)$ are the same operations: $A(x) + B(x) = A(x) - B(x)$.

Multiplication in $GF(2^m)$

In order to understand the multiplication and inversion in $GF(2^m)$, we need to introduce the concept of **irreducible polynomials**. Irreducible polynomials are roughly comparable with prime numbers, (i.e. their only factors are 1 and the polynomial itself).

For example, $A(x) = x^2 + x$, is reducible because $A(x) = x(x + 1)$. On the other hand, $A(x) = x^3 + x + 1$ is irreducible because we cannot factor it.

Let $A(x), B(x) \in GF(2^m)$ and

$$P(x) \equiv \sum_{i=0}^{m} p_i x^i, p_i \in GF(2)$$

be an irreducible polynomial. Multiplication takes part in three steps:

1. Multiply $A(x)$ and $B(x)$ like normal
2. Reduce the *coefficients* of the resulting polynomial modulo 2
3. Reduce the entire polynomial modulo $P(x)$

Note that the irreducible polynomial $P(x)$ has degree of number m, while the elements of the field has degree up to $m - 1$. Both of them have coefficients $a_i \in \{0, 1\}$.

Example: Let $A(x) = x^3 + x^2 + 1$ and $B(x) = x^2 + x$ and the $GF(2^4)$. An irreducible polynomial for this field is $P(x) = x^4 + x + 1$.

Step 1: $C'(x) = A(x) \cdot B(x) = x^5 + 2x^4 + x^3 + x^2 + x$
Step 2: Reduce the coefficients modulo 2. $2x^4 \equiv 0 \pmod 2$, so after this step we have the polynomial $x^5 + x^3 + x^2 + x$
Step 3: We perform the polynomial division which gives $x^5 + x^3 + x^2 + x \equiv x^3 \mod P(x)$. So, $A(x) \cdot B(x) = x^3$.

□

Inversion in $GF(2^m)$

The inverse $A^{-1}(x)$ of an element $A(x) \in GF(2^m)$ must satisfy:

$$A(x) \cdot A^{-1}(x) \equiv A^{-1}(x) \cdot A(x) \equiv 1 \mod P(x)$$

where $P(x)$ is an irreducible polynomial.

Even though $A^{-1}(x)$ can be computed via the extended Euclidean algorithm [CP10], in practice, we use just a lookup table for Galois Fields with relatively small number of elements.

Note: Division is simply a matter of multiplying the first operand by the inverse of the second.

3 Design

3.1 Synchronous vs Asynchronous vs Blocking vs Non-Blocking

Synchronous, asynchronous, blocking and non-blocking are four concepts that can sometimes be little vague, and we encountered them frequently during our thesis. In this section, we will attempt to provide our understanding on these matters.

Programming is a diverse field. The types of tasks programmers and engineers face are varied and cover a wide range of applications. Of course, these tasks may have different requirements, and they may involve inherently different applications. This necessitates software engineers to design software tailored to the demands and specific characteristics of each application.

One of the fundamental concepts engineers encounter when constructing a solution to a problem is the type of communication they will employ to achieve their objectives, if any communication is required.

In this investigation our communication parties are processes (or threads) on the one hand and the Linux kernel on the other.

3.1.1 Synchronous API

We refer to a process (or a thread) as making a **synchronous call** to the operating system (e.g. a synchronous read), when we can be certain that the desired task will be completed

by the time the instruction pointer advances to the next instruction of the process (e.g. for a read this means that the buffer will contain the requested data).

Of course, there will be a waiting period *if* the operating system isn't ready to respond immediately. However not all synchronous calls necessarily lead to blocking. In fact, the ideal scenario is quite the opposite: to have a flawless system where requests are always ready to be serviced without blocking. This is the rationale behind mechanisms like the page cache, as discussed in 2.4.3.

For example, a synchronous read will not block if the data it requires are present in the page cache **and** the page is not marked as dirty. However, if the required data are not in the page cache, **or** if the file is opened with the O_DIRECT flag (bypassing the page cache entirely), then it will block until the operating system retrieves the data from the disk.

The write semantics exhibit some differences. When using the page cache and sufficient space is available to store the data, the operating system employs it as a write-back cache, ensuring that the process never blocks. Once the operating system transfers the desired data to the page cache, it considers the request as completed (see also section 2.4.3).

Synchronous writes are those made to files opened with the O_SYNC flag, or one of the variants O_DSYNC and O_RSYNC. A synchronous write only completes when the data is fully written to persistent storage (e.g., disk devices), including any necessary filesystem metadata changes. Consequently, the kernel places the process initiating the write on a waiting queue until the data is written on the disk.

In all of these cases, whether for reading or writing, and whether the process experienced blocking or not: **synchronous calls guarantee that when control returns to the process, the task will be accomplished**.

An illustration of synchronous behavior is shown in Figure 3.1

3.1.2 Asynchronous API

We refer to a process as making an **asynchronous call** to the operating system, when the job is being done without the process waiting in the kernel. The process starts the job and if the kernel is not ready, doesn't stuck and returns. So, in instruction pointer

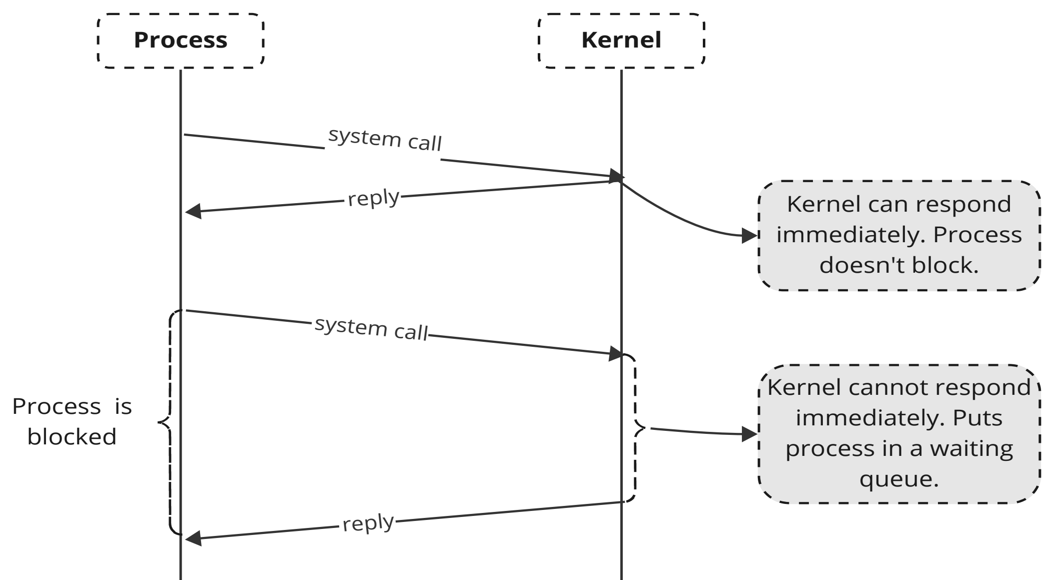

Figure 3.1: *Synchronous API*

terms, when the instruction pointer points to the next instruction of the asynchronous call, the job **may** not have finished yet.

This situation implies the need to establish a means of communication between the kernel and the process in order to enable notifications when the task is completed.

Asynchronous APIs may boost the application's performance if the results of the call are not needed immediately. This way, the process initiates a job, continues performing other tasks, and receives a notification from the kernel when the response is ready.

An illustration of asynchronous behavior is shown in Figure 3.2

3.1.3 Blocking

A **blocking call** is a call that will block the caller until the result is ready. Blocking has the same semantics as **synchronous**, and that is why they are used interchangeably.

3.1.4 Non-Blocking

A **non-blocking call** from an application will not block if the answer is not ready, **but it will not initiate the desired operation either**. Non-blocking is like asking the operating-system «Hey, can you do this job without putting me to sleep? ». If the operating system

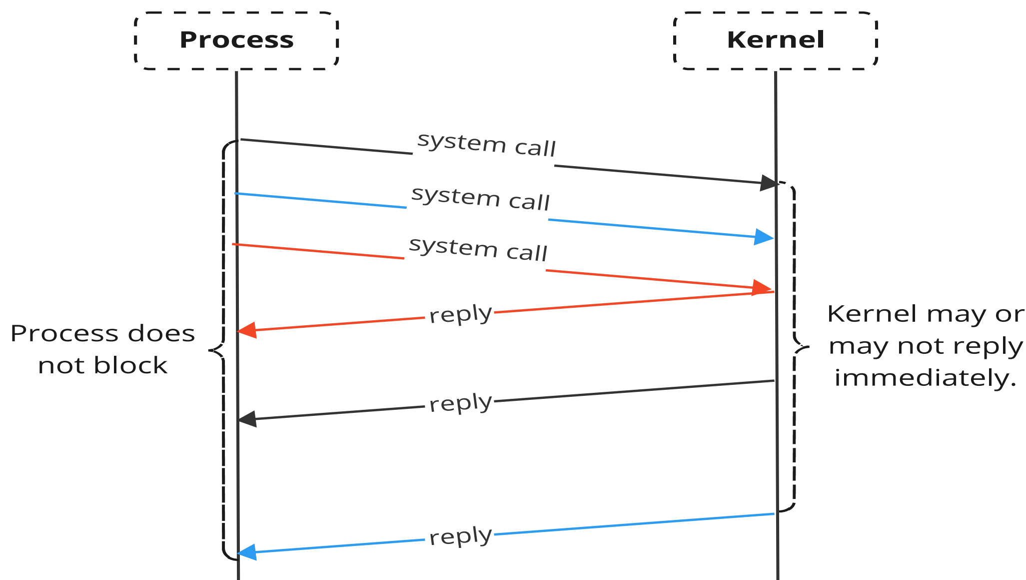

Figure 3.2: *Asynchronous API*

can, then the job will complete successfully. If not, it will just return an error message.

This is a delicate point in the difference between non-blocking and asynchronous. In an asynchronous API, you start a background effort to fulfil your request. In non-blocking this is not the case.

Some general observations

- I have synchronous and asynchronous communication in mind as the general "strategy" of the communication. **As the communication API**.
- Blocking and non-blocking as **mechanisms**.
- If the communication API is synchronous, we can't have a non-blocking mechanism. Because their definitions inherently conflict. **Synchronous means blocking**. An example of this is that if you open a regular file as O_NONBLOCK (i.e. as non-blocking), this will not make any difference in the subsequent calls to the file (see [mpe]). Reading and writing from regular files have a synchronous interface.
- On the other hand, an Asynchronous API can be blocking only if it *chooses* to block and wait for the results. Or it may implement a **part** of the communication as blocking in the background. For instance, asynchronous communication can be achieved if the API creates a thread that will block and wait for the response

and then notify the caller of the result. In this case, we have an asynchronous API with a blocking part. This part is not "visible" to the caller.

Note: Of course synchronous and asynchronous concepts extend out of programming. Imagine if you have to call someone to ask for information. If the other person doesn't have the response and needs to search before the reply you can either wait on the phone till then, or give the other person your phone, and tell him or her to call you back when they have the answer. This simulates exactly the concept of synchronous, where you wait on your phone, and asynchronous where you establish a way to find you (you gave him/her your number) and you close the phone.

3.2 io_uring

io_uring...why?

The most common communication model used by applications today with respect to I/O is *synchronous*. This involves either the oldest read/write system calls or one of the later added variants: pread/pwrite and preadv/pwritev. However, due to their nature, in certain applications, an asynchronous approach matches better. Linux supports two asynchronous APIs: (a) The Linux native asynchronous API (named Linux AIO or libaio) and (b) the one POSIX defines, called POSIX AIO.

These two are implemented differently in Linux. The POSIX AIO is actually a thread-based implementation, manipulated by the glibc library [mpa]. An application that wants to use this API generates library calls which result in offloading the real work to another thread, which then actually blocks while waiting for the result. This implies the need to establish a notification channel so that the "main thread" can be notified that a result is ready. The most notable limitation of POSIX AIO is that maintaining multiple threads to perform I/O operations on a one-thread-per-request basis is expensive and doesn't scale well.

On the other hand, Linux AIO uses system calls directly. Thus, the kernel is responsible for handling the requests without the overhead of one-thread-per-request that POSIX AIO has. However, it also suffers from many limitations. The biggest one is that it doesn't support buffered I/O, but only I/O issued to file descriptors open with the `O_DIRECT` flag

(see 2.4.3). This imposes a great limitation on applications that want to use this API, excluding any application that uses the page cache.

3.2.1 io_uring overview

All these limitations of both Linux AIO and POSIX AIO have led the majority of applications that want to enhance the performance of their application via an asynchronous interface to implement it themselves. Usually, this means that they have to manage a pool of threads that the main core of the application will offload any blocking call to. Of course, this leads to additional burden on the application's side because they have to take into account the thread pool administration, including the communication between the evolving parties.

io_uring tries to fill in this gap in asynchronous Linux I/O. It got merged in the 5.1 release in May 2019. Initially, it was focused on block I/O, but then it evolved to support more system calls. So, it has become like a generic framework for performing system calls in an asynchronous way.

According to its creator, Jens Axboe, the objectives that guided the creation of io_uring, addressed the challenge of balancing **usability**, **flexibility**, **scalability** and of course **efficiency** in asynchronous I/O.

At the core of an io_uring instance are two ring buffers: one for submitting requests named the *Submission Queue (SQ)*, and one for reaping the responses named the *Completion Queue (CQ)*. These buffers are shared between the application and the kernel via `mmap()`, and each queue consists of entries. The SQ consists of *Submission Queue Entries (SQEs)*, while the CQ consists of *Completion Queue Entries (CQEs)*. It's worth noting that while CQEs are indeed embedded in the CQ, SQ descriptor doesn't actually contain the SQEs. They reside in a different mmaped space, and there is an indirection reference from the SQ to them. However, this doesn't change the conceptual semantics of io_uring communication, which can be thought of as depicted in Figure 3.3.

The application submits requests at the *tail* of the SQ and reaps responses from the *head* of the CQ. The kernel extracts requests from the *head* of the SQ and puts responses at the *tail* of the CQ. Thus, although the two rings are shared between the application and the kernel, there is a single consumer and a single producer in each ring (i.e., only one side

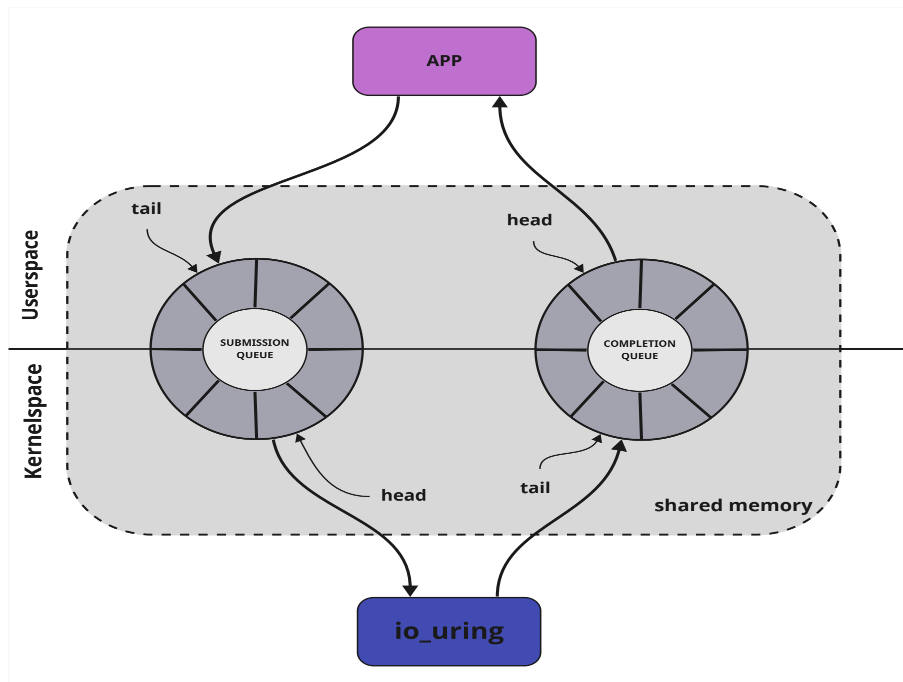

Figure 3.3: *io_uring Visualization*

reads and one writes, and never both). This is the reason why io_uring gets away with only some barriers that guarantee the synchronization between the two communication parties when updating the *head* and *tail*, and avoids locking.

Submission Queue Entry (SQE)

Every Submission Queue Entry is a descriptor of a request that an application wants to perform. It is a 64-byte structure (there is an option to send a 128-byte request as we will see) named `io_uring_sqe` that describes a *traditional* system call with its parameters: what operation we want to perform, and with which arguments. Some important fields of the SQE's descriptor are depicted in Listing 3.1.

```
struct io_uring_sqe {
        __u8     opcode;          /* type of operation for this sqe */
        __s32    fd;              /* file descriptor to do IO on */
        __u64    off;             /* offset into file */
        __u64    addr;            /* pointer to buffer or iovecs */
        __u32    len;             /* buffer size or number of iovecs */
        __u64    user_data;       /* data to be passed back at completion time
    */
        [...]
```

```
};
```

Listing 3.1: *struct io_uring_sqe*

For example, if an application wants to read from a file, it will fill in the opcode with the read identifier (`IORING_OP_READ`) or one of its variants, pass the file descriptor of the file from which it wants to read (`sqe->fd`), specify the buffer where the data will be written (`sqe->addr`), and provide the size of the buffer (`sqe->len`). Then, it will submit the request to the ring.

Completion Queue Entry (CQE)

For every SQE, the kernel will place a corresponding CQE in the Completion Queue when the desired result is ready in order to inform the userspace. CQEs are described by a descriptor named `io_uring_cqe`, and they are fairly straightforward as shown in Listing 3.2. It is a 16-byte struct (there is a case to enable 32-byte CQEs that we will examine later) that packs the information that a system call usually returns (`res` field) along with a flags field and an important identifier named `user_data`.

```
struct io_uring_cqe {
   __u64 user_data  /* sqe->data submission passed back */
   __s32 res;       /* result code for this event */
   __u32 flags;

   __u64 big_cqe[];
};
```

Listing 3.2: *struct io_uring_cqe*

The `user_data` field serves as the link between a Submission Queue Entry and a Completion Queue Entry. This field is never touched by the kernel, and is copied as is from the SQE to the corresponding CQE. This way, the application can identify which request the CQE corresponds to.

Note: It is important to understand that the kernel reaps the requests from the SQ, and there is no guarantee that it will execute them in the same order they were submitted. This means that the CQEs may be out of order. This is why applications need an identifier field (the `user_data` in this case) that will help them recognize which request this

reply belongs to.

Due to the asynchronous nature of io_uring, it cannot use `errno` to notify for an error. That is why when an error happens, the kernel places the corresponding negative error number in the `res` field. For instance, if a read request fails due to an invalid argument (error `EINVAL`), a regular system call will have set `errno` equal to `EINVAL` and will have returned -1. But in io_uring, the kernel places `-EINVAL` in the `res` field of the CQE. If the request is successful, it returns the ordinary positive value (e.g. for a read request, the number of bytes read successfully).

So practically, when the operation an SQE asked for is completed, the kernel simply places a result at the tail of CQ by copying the `user_data` field of the corresponding SQE and informing the `res` field with the result of the operation.

3.2.2 io_uring system calls

io_uring supports three system calls that set up the rings and enable communication with the kernel. Typically, a userspace application will not directly use these calls, as there is a wrapper library named liburing [Axbe] that wraps these functions and provides a higher-level perspective on the operations. We will examine liburing later.

Now, let's focus on the io_uring system calls to gain a better understanding of how io_uring functions.

There are three system calls provided by the io_uring interface:

1. `io_uring_setup`
2. `io_uring_enter`
3. `io_uring_register`

io_uring_setup system call

```
int io_uring_setup(u32 entries, struct io_uring_params *p);
```

Listing 3.3: *io_uring_setup system call*

`io_uring_setup` is the first system call that an application must make to inform the kernel about its intention to start an io_uring instance. The user must fill in the `entries` field and can also specify some flags, which is a field inside the second parameter of the system call.

The kernel takes the `entries` value and uses it to set up the submission and completion buffers, ensuring there are at least `entries` slots available. Additionally, it reads the `flags` field of the `io_uring_params` structure to understand which specific *features* this particular instance will employ. We will explore alternative modes of using io_uring beyond the standard one later. Furthermore, the kernel is responsible for populating other fields within the `io_uring_params` structure to provide crucial information about the configured instance to the userspace.

In other words, `io_uring_params` serves as a "handshake object". Applications set up the flags to indicate the kind of communication they want, and the kernel responds with important information for the instance. Finally, the kernel will return a file descriptor, which the application can use from now on to refer to the io_uring instance.

After the successful return of `io_uring_setup`, the first thing an application must do is to map the submission and completion queues, so that it can access this memory. This can be done via an `mmap` system call, using the descriptor returned by `io_uring_setup`. The offsets of the rings can be retrieved by the application via specific fields of `io_uring_params` (the "handshake object").

A visualization of the setup process along with the `mmap` call is depicted in Figure 3.4.

Once the rings have been successfully mapped, the application can now use them to communicate with the kernel.

io_uring_enter() System Call

```
int io_uring_enter(unsigned fd, u32 to_submit, u32 min_complete, u32 flags,
    const void* argp, size_t argsz);
```

Listing 3.4: *io_uring_enter system call*

This system call notifies the kernel that there are SQEs in the Submission Queue. The parameter `fd` must be the file descriptor of the io_uring instance an application wants to use (the one `io_uring_setup` returned). `to_submit` informs the kernel about how

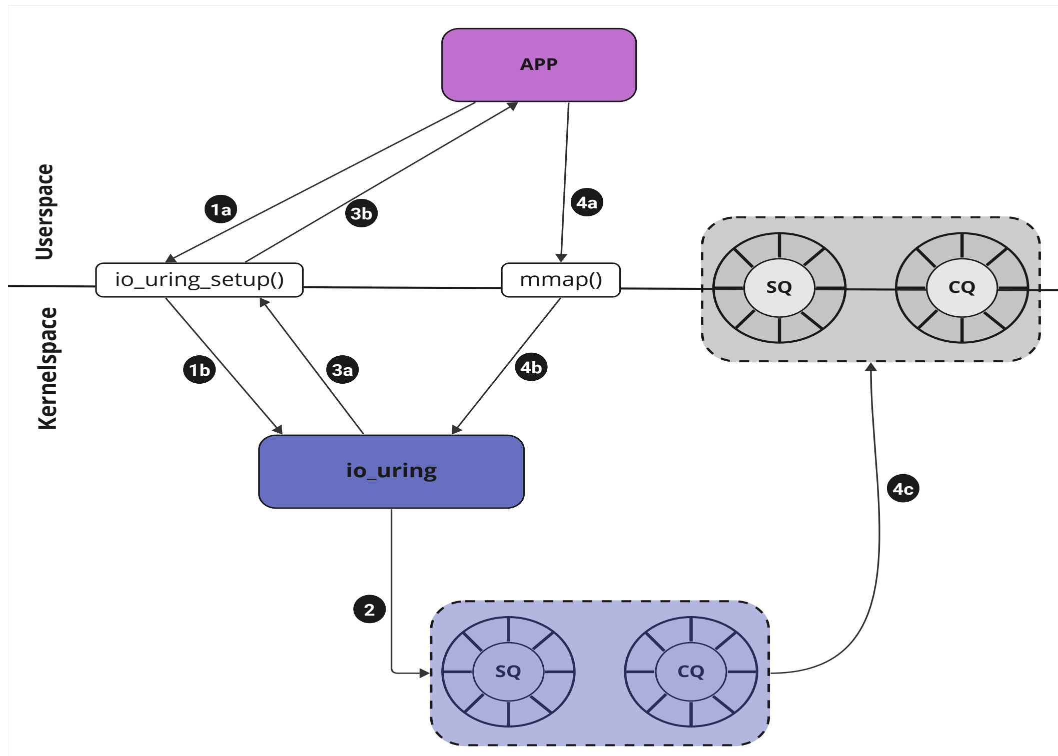

Figure 3.4: *Setup an io_uring Instance*

many SQEs are ready for consumption, while `min_complete` is used to instruct the kernel to return control back to the application only when `min_complete` CQEs are ready. Setting this field enables both the submission and the waiting for requests to take place in one step, with one system call. In practice, this provides *synchronous* semantics in the call, as control returns to the application only when `min_complete` requests have been satisfied. Of course, an application can set this parameter to 0 and not wait for any response.

An important thing to note in this design is that an application can batch up a lot of requests and submit them all at once with **only one system call**. This is an immediate optimization compared to traditional system calls, where you have to pay the overhead for every context switch.

Every application that wants to submit an SQE to the ring must follow four distinct steps:

1. Extract an empty SQE.
2. Prepare a request by filling in the desired operation and arguments.
3. Place the request in the SQ.
4. Submit the SQE(s) to the kernel via `io_uring_enter`.

Only the fourth step leads to a system call. Due to batching, an application can perform steps 1-2-3 as many times as the available (non-used) entries in the SQ allow, and only perform Step 4 when it wants to notify the kernel.

This specific design, of course, helps in the submission of requests that don't require any synchronization (i.e. they don't depend on each other). However, this is not always the case. There are times when users want to submit requests that need to be executed in a specific order. For example, can I submit a write and a read request, and be sure that the read request will retrieve the updated data? The answer is yes. io_uring provides a specific flag (`IOSQE_IO_LINK`) that, if set in an SQE, it guarantees that as long as the next submitted SQEs have the same flag set, the execution will occur in the order they were submitted. This chain of "serial" requests continues until the first SQE that doesn't have the `IOSQE_IO_LINK` set.

io_uring_register() System Call

```
1 int io_uring_register(unsigned fd, unsigned opcode, void *arg, unsigned int
    nr_args);
```

Listing 3.5: *io_uring_register System Call*

Using userspace buffers and file descriptors in the kernel comes with some overhead. Therefore, it is possible to preregister buffers and descriptors using the `io_uring_register` syscall.

The `fd` field refers to the instance of io_uring, while the opcode is a flag that indicates the type of the resource we want to register with the kernel. Registering files or user buffers allows the kernel to take long term references to internal data structures or create long term mappings of application memory, greatly reducing per-I/O overhead [mpc].

3.2.3 Thankfully...liburing!

io_uring comes with a userspace library, named liburing [Axbe], developed and maintained by Jens Axboe, the creator of io_uring. Liburing abstracts away all the nitty-gritty details, complexity, and low-level adjustments that must be made when setting and manipulating the ring buffers directly. This library aims to achieve one of io_uring's goals:

"easy-to-use, hard-to-misuse", making it highly recommended for applications to use the wrappers it provides instead of directly interacting with the system calls whenever possible.

In the following example, we demonstrate a typical workflow with explanatory comments to show how an application can perform all of the previously discussed steps. The primary structure that describes an io_uring instance in liburing is the `struct io_uring`. This descriptor consolidates all the important information that defines this instance, including the Submission Queue and Completion Queue descriptors, the file descriptor returned by `io_uring_setup` (see Listing 3.3), and other flags.

Of course, due to the versatility of io_uring, this example only scratches the surface of its potential uses. Nonetheless, it illustrates the basic workflow and the essential wrappers provided by liburing. Specifically, in the following code snippet, we create an io_uring instance with 8 slots, retrieve an SQE, prepare a read request, submit the request, wait for the response, and finally update the CQ head counter after retrieving the result from the ring.

Note: For the sake of simplicity, we omit error handling.

```
1  struct io_uring ring; // The descriptor of an io_uring instance
2  struct io_uring_sqe *sqe
3  uint64_t user_data = 1;
4  char buf[64];
5  int fd;
6
7  fd = open("file_to_read_from", O_RDONLY);
8
9  // Set up a communication channel. This wrapper does both
10 // the io_uring_setup() and the mmap() of the rings.
11 io_uring_queue_init(8, &ring, 0);
12
13 // Retrieve an sqe from SQ's tail
14 sqe = io_uring_get_sqe(&ring);
15
16 // Prepare a read request from file fd. We want to
17 // read 64 bytes at buf, starting from offset 0
18 io_uring_prep_read(sqe, fd, buf, 64, 0);
19
20 // Set user_data to recognize the completion
```

```
io_uring_sqe_set_data64(sqe, &user_data);

// Submit all requests available in SQ.
// Will invoke the io_uring_enter syscall
io_uring_submit(&ring);

// Do whatever you want. We didn't block

struct io_uring_cqe *cqe;
// wait for a completion
io_uring_wait_cqe(&ring, &cqe);

// A completion has arrived. Check the user_data
// to verify if it is the one we want
uint64_t check = io_uring_cqe_get_data64(cqe);
if (check == user_data) {
  // Yes this was the response to the read
}

// Mark cqe as seen. Increment the ring head of CQ
io_uring_cqe_seen(&ring, cqe);

// exit io_uring instance and free the resources
io_uring_queue_exit(&ring);
```

Listing 3.6: *Simple Liburing Workflow*

3.2.4 Advanced modes of operation

io_uring provides two different kinds of polling modes for applications aiming for very low latencies: "SQPOLL" and "IOPOLL".

SQPOLL

This is an operational mode of io_uring designed for applications that prioritize low latency and have a high request submission flow. It is enabled during the setup phase of the io_uring instance using the `IORING_SETUP_SQPOLL` flag. As a consequence, the kernel spawns a dedicated thread that polls the Submission Queue (SQ) for available requests. This allows the application to submit requests without the need for a system

call to notify the kernel. The application simply prepares a request, updates the tail index of SQ, and as long as this is different from the head, the kernel thread recognizes that there are pending requests. It then extracts them and proceeds with the necessary steps to fulfill them.

This is another step towards efficiency after request batching: **the ability to submit tasks without a single system call**. It targets applications with high submission rates that don't mind incurring an additional CPU utilization cost. To prevent unnecessary consumption of system resources, if the application doesn't submit an SQE for a predefined amount of time (which is configurable), the kernel thread will go to sleep. The application will need to call `io_uring_enter` the next time it submits a request to wake it up again.

IOPOLL

This mode can be enabled via the `IORING_SETUP_IOPOLL` flag during the initialization of the ring. It specifically pertains to I/O operations against block devices and filesystems. In general the responses of block devices are interrupt driven, meaning that when a device finishes a requested operation, it interrupts the CPU to signal completion. Enabling IOPOLL means that the process that made the request (via `io_uring_enter`) will actively poll for completions on the target device. This reduces overhead for high IOPS applications, and reduces latency in general [Axbf].

Note: This mode resembles a flag that can be passed in regular `preadv2()` and `pwritev2()` named `RWF_HIPRI`, which «allows block-based filesystems to use polling of the device, which provides lower latency, but may use additional resources»[mph].

How kernel treats an SQE

io_uring is a complex subsystem within the kernel that is continuously evolving. The following analysis is an attempt to articulate my understanding of this matter, it is not exhaustive, and may, at some points, not be entirely correct.

There are three different execution paths that a submitted SQE may follow inside the kernel:

1. Executed in the process context of the process that submitted the request, without blocking (inline execution).

2. If the result is not available, but it can be polled for readiness, it is placed in a set of pollable requests.

3. If the result is not ready, and cannot be polled for readiness, it is placed in a waiting queue and will be executed by a thread from a dedicated worker pool.

io_uring attempts to complete a request inline. If this is not possible, it checks whether it is a request that has a non-blocking behavior. By this, we mean a request on a file descriptor that can be checked for readiness, such as a socket. This does not hold true for file descriptors that refer to "disk files". As discussed in 3.1.4, *real files* have a blocking/synchronous interface. If the request supports non-blocking I/O, io_uring monitors it for readiness without needing to spawn a new kernel thread. When the file descriptor becomes ready, io_uring is notified and continues with the execution.

Finally, there is a third path for requests that cannot be handled in a non-blocking manner and are expected to block. These requests are placed in a queue, and a kernel thread is spawned and assigned to their execution. Additionally, there is a specific flag (`IOSQE_ASYNC`) that an application can assign to an SQE before submission, indicating to the kernel that «we are aware that the request will likely block, thus bypass the non-blocking path (step two) and assign this task to a worker ».

io_uring classifies tasks, in this third path, into two types: those that can be completed in a **bound time** (such as reading from a file) and those that may potentially never complete, known as **unbounded work** (like reading from a socket). These two categories of requests are managed by separate groups of workers. Bounded tasks are delegated to bounded workers, which are kernel threads, with the number of these threads being limited by the size of the Submission Queue. In contrast, unbounded tasks are assigned to unbounded workers, which are also kernel threads, and their quantity is determined by the system's resource limit RLIMIT_NPROC [Axbd, Sit].

3.3 Coroutines

Ublk server, uses a feature of C++20, called "coroutines" to communicate with the "target" after receiving a request from the ublk driver. In this section, we will explore what coroutines are and how are implemented in C++20. Before explaining coroutines, we

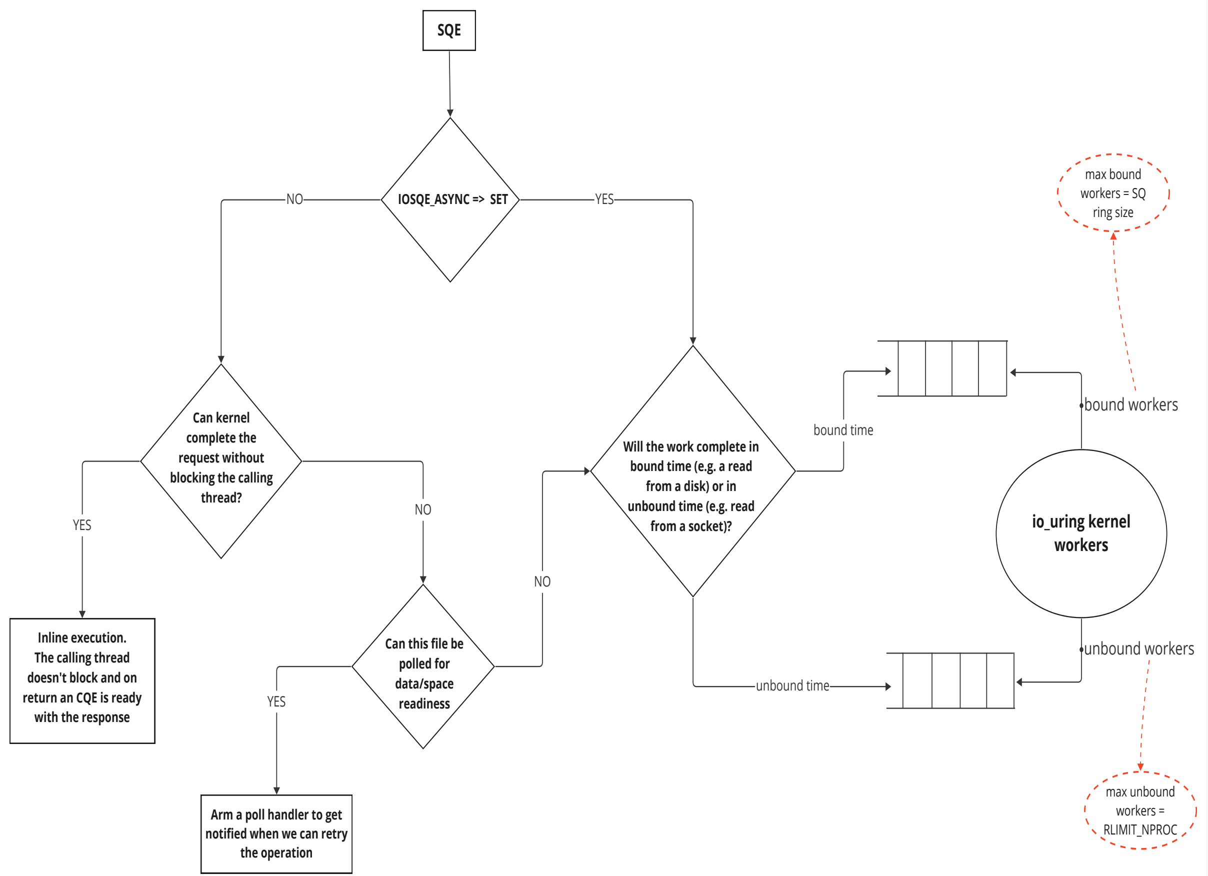

Figure 3.5: *In-Kernel Path of a SQE*

will provide some general theoretical background to understand what problems they address and their role in the realm of program, processes, subroutines, etc.

Let's begin with definitions of fundamental programming entities:

- **Programs** are binaries residing in a storage medium, ready for execution but not yet loaded into memory.
- **Processes** are running programs that have been loaded into the memory.
- **Threads** are execution units *within* processes. They are the smallest execution unit in Linux.

A process contains at least one thread, but it can also contain many. If a process has only one thread, it is considered **single-threaded**. Conversely, if a process contains multiple threads, it is **multithreaded**, indicating that more than one task is occurring concurrently.

The implementation of threads in a system, can vary according to how much the kernel and userspace is involved [Lov13]:

- 1-1 model, known as **kernel-level threading**, where the kernel provides support for every thread in the system. Each thread is represented in the kernel by a structure called `task_struct`. Threads belonging to the same process share some of these resources saved in their `task_struct`, which is why they are traditionally called *lightweight processes*.
- N:1 model, known as **user-level threading**. A process with N threads maps to a single kernel process. This model requires minimal support from the kernel for thread implementation, but it necessitates the development of userspace components, such as a scheduler, to manage thread operations.
- N:M model, a **hybrid model**, where N userspace threads are mapped to M kernel threads in the kernel (where $M < N$).

Linux has native support for the first model. Each thread in userspace corresponds to a thread (`struct task_struct`) in kernel. To implement other models in Linux, attention to userspace implementation details is required.

The advantage of the 1-1 model is that it enables true parallelism, as each thread has its own CPU and can run concurrently with other threads, whether from the same process or not. This is not the case for userspace threads. As mentioned earlier, all userspace threads "belong" to one descriptor in the kernel and are scheduled by the OS as a unit. This means that true parallelism is not achievable in userspace threads.

At this point, let's clarify two "similar but distinct" definitions regarding ***concurrency*** and ***parallelism***. Concurrency is an overarching concept that encompasses parallelism. When we say that two or more threads run **concurrently**, we mean that all of them make forward progress, though not necessarily simultaneously. For instance, concurrency can occur in a multithreaded program running on a single CPU. All threads make forward progress, but a scheduler arbitrates their turns, because only one thread can run at a given time. Conversely, threads run in **parallel** when they execute simultaneously. This implies that parallelism is limited by the number of cores in a system. For example, an 8-core machine can run up to 8 threads in parallel, even though thousands may run concurrently.

Now, it is clearer why user-level threading does not support "parallelism" as opposed to kernel-level threading. However, user-level threading incurs (almost) no overhead for thread switching, as they do not require the OS's intervention to switch.

3.3.1 And...what are coroutines?

Coroutines are based upon the concept of "cooperative multitasking". This means that a task (an execution flow in our concept) willingly relinquishing control to allow other tasks (other execution flows) to run. This resembles the notion of functions, yet differs in a crucial aspect. Unlike classical functions (subroutines) that follow a linear start-to-finish execution without interruption, coroutines have the ability to pause their execution voluntarily and later resume. In this sense, they can be viewed as an extension of classical subroutines, possessing the ability not only to be invoked and return but also to temporarily suspend and then resume.

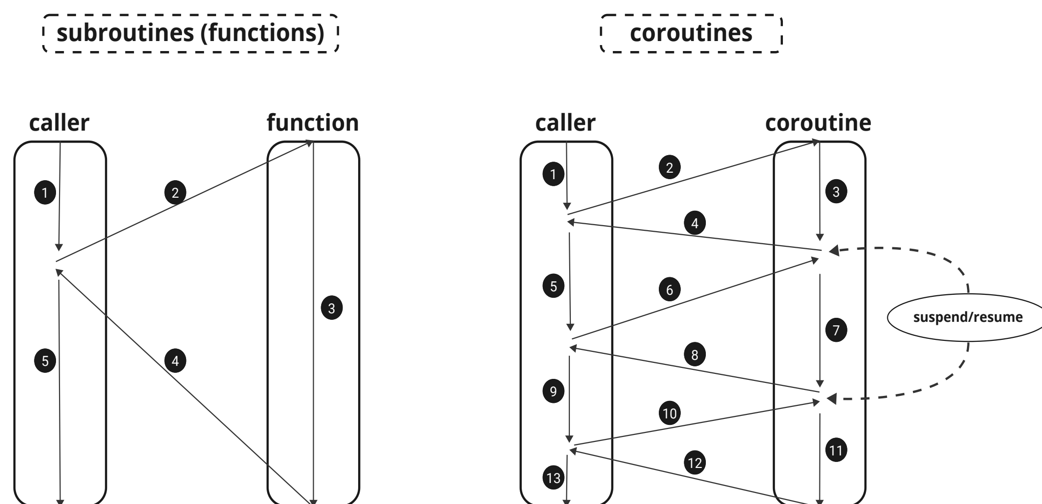

Figure 3.6: *Subroutines vs Coroutines*

Coroutines are userspace phenomena, much like user-level threads (N:1 model), but unlike user-level threads, they require little or no userspace support for their scheduling and their execution. Instead they work cooperatively relinquishing voluntarily the CPU in order to run another execution flow (other coroutine or function). They are more about *program control* than *concurrency*.

Since coroutines function within userspace, the specific implementation details are typically determined by the designers of the supporting library. As a result, different pro-

gramming languages may implement coroutines in different ways. One key distinction lies in whether they are **stackful** or **stackless**. In traditional functions, due to their nested structure, their activation frames can be efficiently stored in the stack, which is good at quick allocation and deallocation. However, coroutines require the preservation of certain information to enable later resumption. Stackful coroutines save their entire activation frame upon suspension, while stackless coroutines take a different approach, retaining only essential information needed for resuming execution.

3.3.2 Coroutines in C++

Coroutines support in C++ is a relatively recent addition, introduced in C++20, and it's still evolving. C++ coroutines are considered **stackless**, meaning they don't save the entire stack of the coroutine when they suspend.

However, to resume execution, certain crucial information must be preserved. When a coroutine is called, the compiler assigns it a stack frame just like it would for a regular function. It then allocates a memory frame in the **heap** and saves the coroutine's parameters, which are typically used after the first suspension point. Figure 3.7 provides a visual representation of the activation frame of the coroutine (`bar()`) after being called from the function `foo()`. It can be thought of as divided into two parts: the conventional **stack frame** and an additional frame saved in the heap, often referred to as the "**coroutine frame**". The compiler is smart enough to determine what information will be needed after the first suspension point and ensures it's stored in the coroutine frame. Any value that is not used post-suspension is saved on the stack [Bak].

Elements of a coroutine

There are three new keywords essential for coroutine support: `co_await`, `co_yield`, and `co_return`. The compiler recognizes a routine as a coroutine, rather than a regular function, if it encounters one of these three keywords within the body of the routine. Suspension points are identified by the use of `co_await` or `co_yield`. The third keyword, `co_return`, comes into play when the coroutine is finished. If there's no `co_return` at the end of the coroutine's body, it is implicitly executed.

To declare a coroutine you need to specify at least two types:

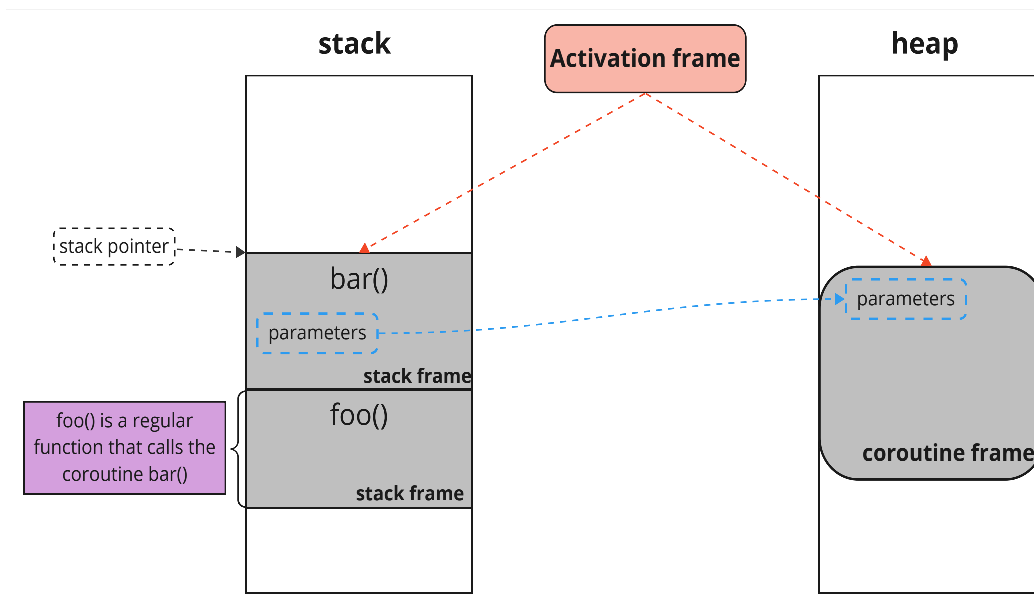

Figure 3.7: *Coroutine's Activation Frame*

1. **A wrapper type**. This is the return type of the coroutine function's prototype. With this type we can control the coroutine from the outside. For example, resuming the coroutine or getting data into or from the coroutine. This is basically done by storing a so called "handler" to the coroutine, more or less a function pointer which knows how to invoke functions like `resume`.

2. **A Promise Type**: Within the *wrapper type*, the compiler searches for a type with the **exact** name `promise_type`. This serves as internal control. Therefore, `promise_type` is a hardcoded thing the compiler looks for within the wrapper type. Its presence is crucial for a valid coroutine.

So, these are the main two components a coroutine requires: a type that encompasses another type. The outer type serves as the return type of the coroutine, while the inner type (the promise object) is utilized by the compiler to generate calls to specific methods it defines at critical points during the coroutine's execution. The wrapper type typically also contains the "coroutine handle", akin to a function pointer, which facilitates the resumption (or destruction) of the coroutine. The return type of the coroutine is the only access the caller has to it. So, if further interaction is anticipated, the coroutine handle must be embedded within the wrapper type, ensuring the caller knows how to invoke and manage the coroutine's state.

Upon the initial call of the coroutine, as depicted in Fig. 3.7, the compiler proceeds with

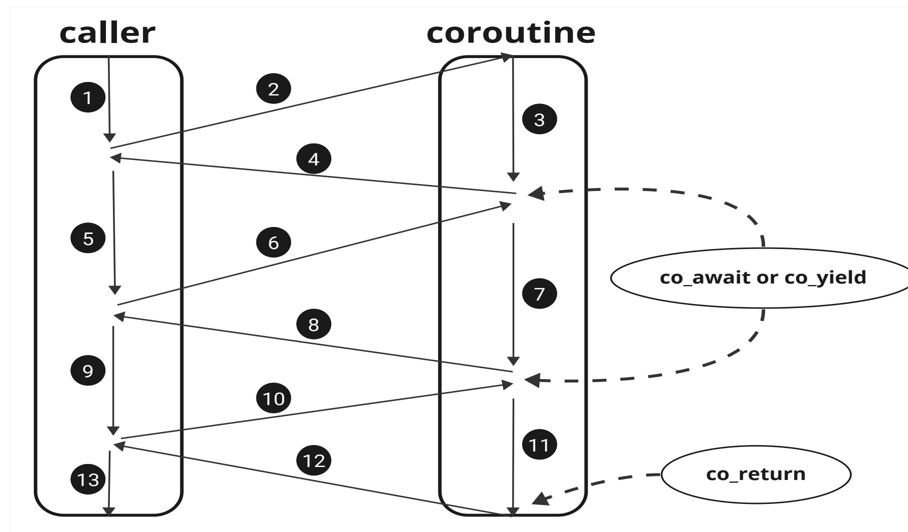

Figure 3.8: *co_await, co_yield and co_return*

the following steps: (a) allocates a coroutine frame in heap (b) copies the parameters from the stack frame to the coroutine frame (c) identifies the promise type inside of the wrapper type and allocates a promise object inside coroutine's frame.

From this point onward, the compiler triggers method calls on key points in the coroutine's life-cycle that are defined in this promise object.

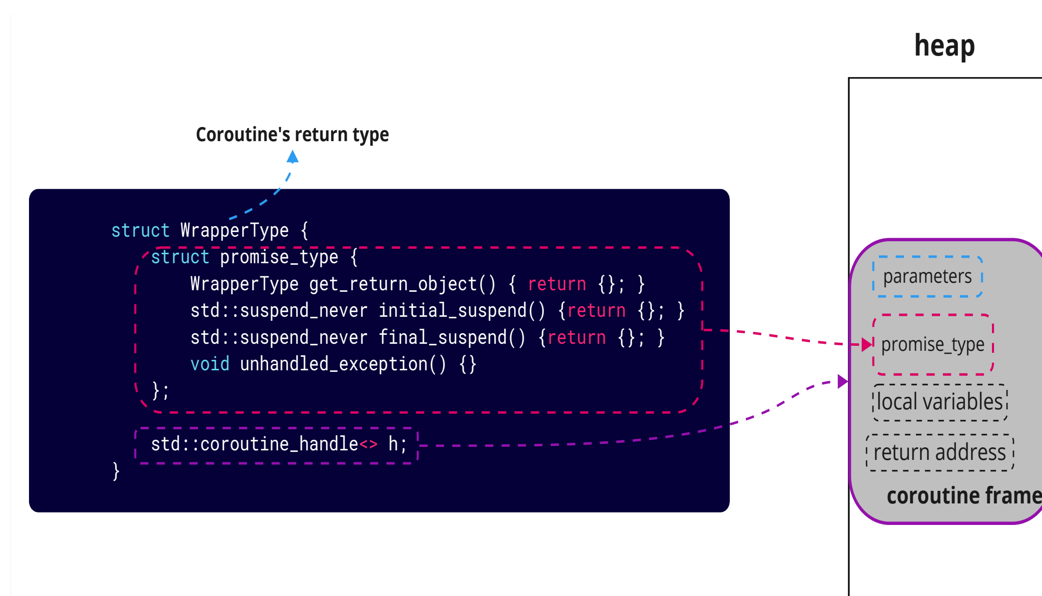

Figure 3.9: *Return Type, promise_type and Coroutine Handler*

Figure 3.9, shows the contents of a coroutine frame when the coroutine is suspended. It saves the parameters, the `promise_type`, various local variables for post-resumption use and a return address, specifying where execution should continue once it resumes.

Additionally it illustrates a simple wrapper type with an embedded promise type and a coroutine handle. The coroutine handle references the coroutine frame, signifying that this pointer "knows" how to resume the coroutine's execution.

Furthermore, within the promise object, there are several methods that the compiler invokes at various stages of coroutine execution, as previously discussed. At the very start, the `get_return_object()` method is invoked which returns the wrapper type that the coroutine will return to the caller the first time it is suspended. `initial_suspend()` is triggered just *before* the main body of the coroutine and `final_suspend()` is called *after* the main body of the coroutine. Essentially, the compiler transforms the body of a coroutine, as depicted in Figure 3.10, by invoking these two calls on the promise object methods at the beginning and end. This means that, for instance, the coroutine can be suspended even before its main body begins execution.

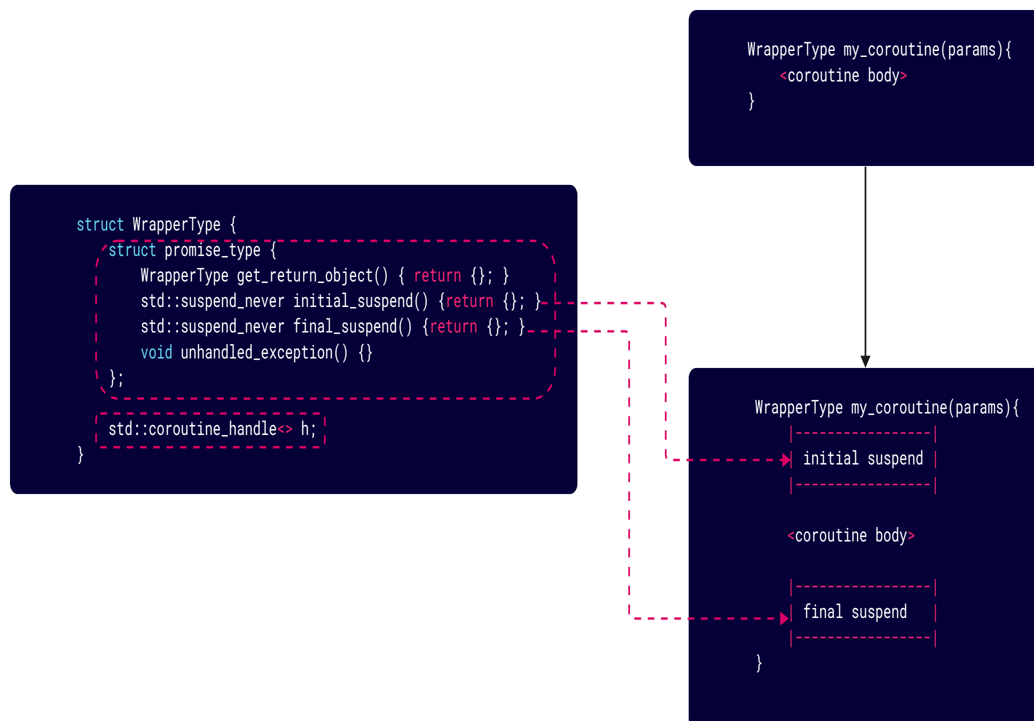

Figure 3.10: *Compiler Transformation on Coroutine's Body*

The co_await operator

The `co_await` is a *unary operator* that can be applied to an expression, like `co_await <expr>`. The type of `<expr>` **must** support specific methods and is called an Awaitable. Awaitable items are any type that offers three functions: `await_ready()`, `await_suspend()`, and `await_resume()`. When the coroutine encounters `co_await <expr>`, these functions

dictate its behavior.

For example, in Figure 3.9, we see the type `std::suspend_never` as a return type of two methods of the `promise_type`. This is a special Awaitable type offered by the standard library which when invoked never suspends the execution. If it were an `std::suspend_always`, another special Awaitable from the standard library, then execution would have suspended.

In general, the `co_await` operator serves as a point where we can customize the behavior of the coroutine. It allows us to pause the execution and wait for something, then return once we have it. We can extract a value from the coroutine or pass something to it. `co_await` can behave in a way similar to passing parameters to a regular function. Because with a coroutine, you can only pass parameters at the start. What if, somewhere in between, we need to provide additional data to the coroutine? `co_await` allows the coroutine to pause and say «Hey, I need more data from the outside. Please instruct me on what to do next».

The `co_yield` operator allows a coroutine to produce a value and then suspend, often used in generator patterns. Meanwhile, `co_return` signals the coroutine's end, indicating its completion and potentially returning a value.

3.4 Ublk

The introduction of the new communication mechanism, io_uring, has generated interest within the Linux community in relocating functionality from the kernel to userspace. In cases where communication time between userspace and the kernel was the main limitation, io_uring's reduced latency in userspace/kernelspace communication is prompting a reconsideration of operational strategies.

This has led to a reevaluation of certain concepts, in the way of trying to find ways to move things to userspace, including the implementation of Virtual Block Devices. Virtual Block Devices are software-emulated devices that replicate the behavior of physical devices. This process remains transparent to the end user, who interacts with it as if it were a "real" storage medium.

Linux already includes several implementations of Virtual Block Devices within the ker-

nel. Examples include the **loop block device**, which maps a regular file to a block device, and the **Network Based Device (NBD)**, which uses a remote server as a block device [Doca].

The question that arises is: why choose to implement these devices in userspace when they can be directly implemented in the kernel? Here are some compelling reasons:

1. **Programming flexibility:**
    - They can be developed in a much less restrictive environment compared to the Linux kernel.
    - They can be written in a variety of programming languages.
    - They can use already existing libraries and frameworks that are not available in the kernel.
    - Debugging is significantly more straightforward using tools familiar to application developers.
2. **Security:**
    - If a userspace block device encounters an issue, it won't cause a complete system crash or kernel panic, ensuring that the rest of the system remains stable.
    - Bugs in userspace block devices are likely to have a lower impact on system security compared to bugs in kernel code, leading to a more robust system.
3. **Independent in Development and Maintenance:**
    - They can be developed and maintained independently of the kernel, providing greater flexibility in managing system's components.
    - They can be a lot easier for testing, facilitating more efficient development workflows.

However, it's important to note that critical operations requiring fast response times, which are implemented within the kernel, cannot be easily migrated to userspace. One must carefully weigh the advantages mentioned above against the need for fast response times.

Of course, virtual devices still require a small piece of kernel code, a module, to interact with the Linux kernel. As mentioned previously, end users may not understand the differences between using a Virtual Block Device or a regular one. They expect to communicate with it as they would with any device under the /dev directory (which we covered in 2.2.3).

The Virtual Block Device is transparent not only to end users but also to other Linux subsystems. For example, the block layer (see 2.4.4) is unaware of the device to which the request it prepares will be directed. Therefore, even for Virtual Block Devices implemented in userspace, a kernel module is needed to interact with the rest of the kernel as usual, but it will forward the information to the userspace component to take care the implementation.

Figure 3.11, illustrates this design.

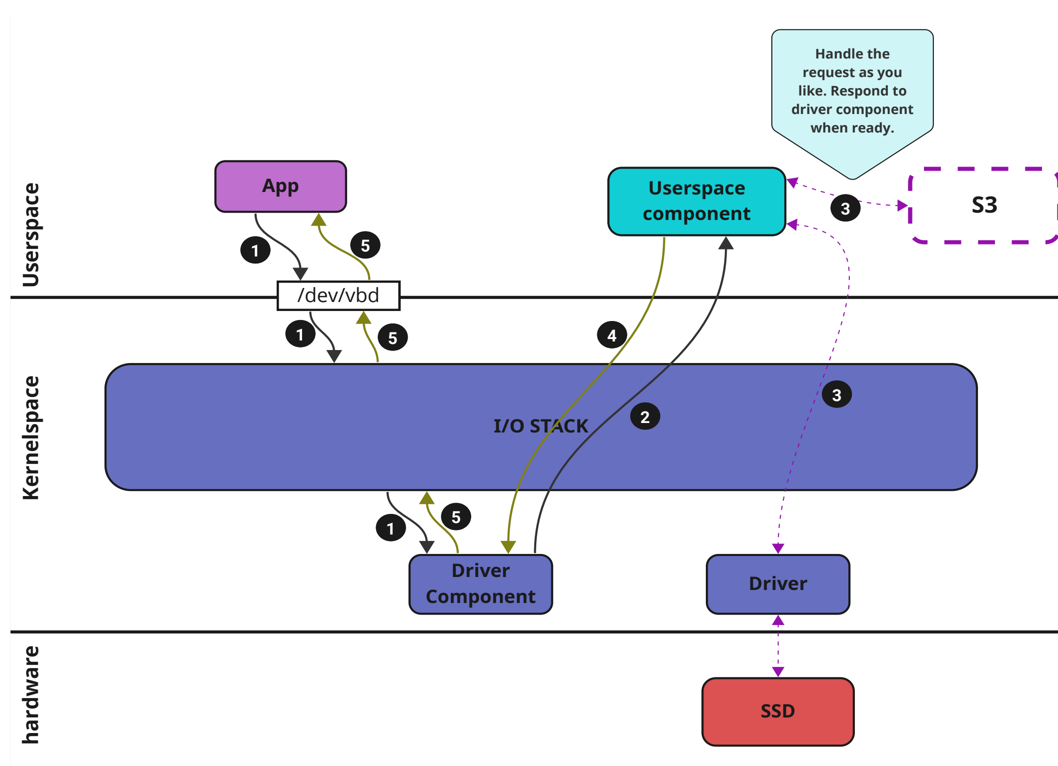

Figure 3.11: *Virtual Block Devices General Design*

3.4.1 Ublk overview

This concept is followed by the ublk framework as well. As we stated in the Introduction chapter 1.1, ublk was designed by Ming Lei, and its driver component was merged

into the Linux kernel v.6.0 as an experimental module. The ublk framework consists of two communicating components: a userspace component, which we will call "**ublk server**" or simply "**server**", and a corresponding kernel module, which we refer to as "**ublk driver**" or just "**driver**".

At the moment of writing this thesis, ublk can support a specific number of targets. We'll use the name "**targets**" to refer to the emulated devices. Null, loop, NBD and qcow2 are the currently supported types of emulated block devices, with a lot of new features being under development, especially in the qcow2 target.

In this thesis, we will focus on the **loop** target. In other words, we implemented a cryptographic system that specifically works with the loop target, which is why we will examine the design of this target. Of course, the general concepts remain the same for all targets. Only the specific target code changes. We will explore this design in the following sections.

Ublk general design

The ublk driver, exposes three different special devices under `/dev`. The first one is registered in the kernel when we `insmod` the ublk driver component. This is a simple character device (a miscellaneous device) and it appears in the system as `/dev/ublk-control` (we covered misc devices in Section 2.2.5). This special device exposes an interface that can be used for controlling the kind of operation we want to perform. More specifically, we can use `ublk-control`, to **add** a new device, **delete** a new device to **set** and **get** parameters, among other operations.

To utilize this interface, we must start the ublk server and instruct it to perform a specific operation. For example, we can add a new emulated loop device by running "`ublk add -t loop -f backing_file`", where "backing_file" is the file that will be used as the storage device.

For each new emulated device we add, we use `ublk-control` and two new devices are registered in the system and appear under `/dev`. A character device (`ublkcN`) and a block device (`ublkbN`). The block device is the device we emulate, the device that end users want to use as a disk. The character device is used to control this block device and orchestrate its operation. There is a 1-1 correspondence between a character and a

block device. Each time we want to add a new emulated device, both a character and block device are created. The "N" in their name, which is the device identifier, can be configurable by the user when the device starts or by the driver component itself in incremental order if no specific device ID is passed by the user.

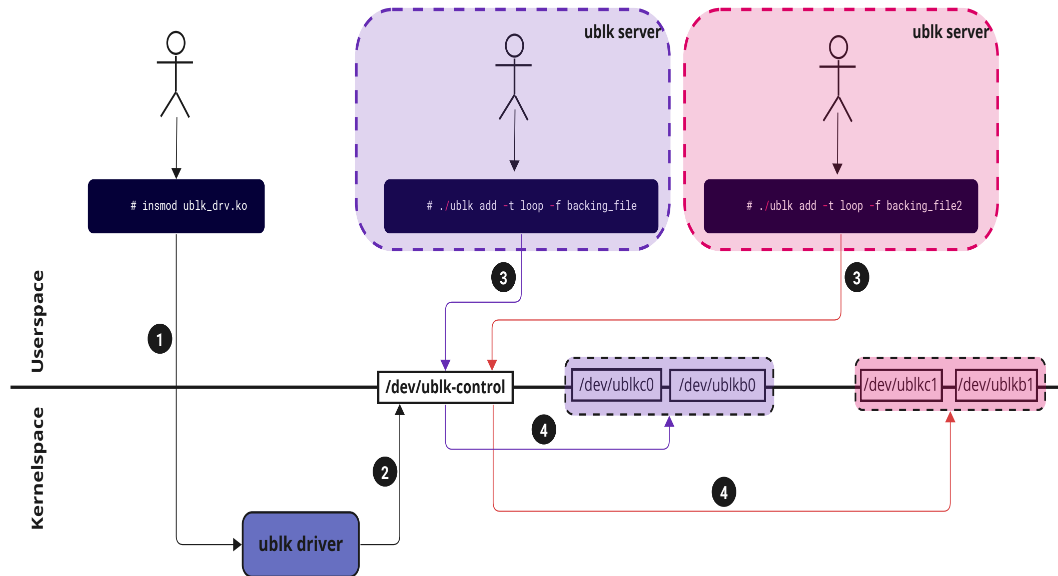

Figure 3.12: *Ublk, the Big Picture*

Figure 3.12 illustrates this concept. We will examine each of these steps in more depth later on, but this provides a general idea. We add the module in the kernel (1), which exposes a special device (2), and this is used every time we want to create a new emulated device (3, 4). Each emulated device comes with a character device that is used during the data path to control its execution.

IOCTL and io_uring command passthrough

As mentioned, the communication between the driver and the server is performed via io_uring. More specifically, the operation performed for each Submission Queue Entry is an ioctl-like operation referred to in io_uring terminology as a "io_uring passthrough command". The opcode filled in each SQE is `IORING_OP_URING_CMD`.

`ioctl()` is a system call that enables userspace to issue arbitrary commands. In addition to `read()` and `write()` operations, drivers sometimes need the ability to perform

certain device control tasks that fall outside the scope of classical read/write. `ioctl()` provides developers with the ability to pass specific commands to the device. These commands can be implemented in a driver-specific manner and may hold meaning only for a particular type of driver.

As discussed in Section 2.2.4, each character device is associated with a `file_operations` structure, which defines the functionality of this driver and the available operations it can manage. One of these operations is the `unlocked_ioctl` function pointer, which handles the `ioctl` system call on the character device. For more information on the "unlocked" prefix see [Corb].

To add support for io_uring operations to standard character devices, one needs to incorporate this functionality into the `file_operations` struct. Since it is common to use I/O operations in conjunction with ioctl commands, the ability to handle these asynchronously would be beneficial.

This need is addressed by the io_uring command passthrough. To achieve this, a new operation was added to the `file_operations` struct. This operation is called `uring_cmd`, as seen in Listing 2.6.

Thus, we can handle operations in a character device that come through an io_uring instance. This concept is utilized in ublk. Both the `ublk-control` and `ublkcN` character devices implement the `uring_cmd` function to provide the desired functionality.

Let's now explore more deeply into all these concepts we have referred to so far. In order to understand ublk framework properly, we will separate its examination into 3 phases:

- **The initial phase**, involving communication/negotiation between the server and driver to setup the basic attributes, which is centered around `ublk-control`.

- **The second phase**, which consists of the setup of the server itself, where it creates a pthread for every queue.

- **The third phase**, where an application uses the ublk framework for an I/O operation.

3.4.2 Initial Phase: Setting up the Environment

The miscellaneous device `ublk-control` is used for controlling ublk devices. The corresponding struct `file_operations` is listed in Listing 3.7. Basically the only useful operation this special device implements is the io_uring command passthrough, which is implemented via `ublk_ctrl_uring_cmd`.

```
static const struct file_operations ublk_ctl_fops = {
  .open    = nonseekable_open,
  .uring_cmd      = ublk_ctrl_uring_cmd,
  .owner     = THIS_MODULE,
  .llseek    = noop_llseek,
};
```

Listing 3.7: *ublk-control file_operations*

Some of the supported operations, that are important to us include:

- Adding a new character device (`UBLK_CMD_ADD_DEV`). This is the `ublkcN` device mentioned earlier, which serves as the communication medium between the ublk server and the ublk driver. When sending this command, the ublk server can customize the new device, such as setting the number of hardware queues it supports (for more on hardware queues, see 2.4.4), the depth of each queue, the length of the I/O buffers, and more. These settings sent by ublk server cannot exceed internal kernel limits, which is why the final information is sent back to the server with the reply to this command.

- Setting parameters for the device (`UBLK_CMD_SET_PARAMS`). These parameters refer to the backing file attributes and include the number of sectors of the emulated device, the number of sectors for each I/O buffer, the block size, etc. These can only be set before the real device (`ublkbN`) starts. These parameters also undergo validation before being put into practice.

- Starting a new block device (`UBLK_CMD_START_DEV`). This registers the real device. After this command succeeds, the block device is up and registered in the kernel and can be used by applications.

- Getting parameters from the device (`UBLK_CMD_GET_PARAMS`). This is the reverse of `UBLK_CMD_SET_PARAMS`. The ublk server gets the parameters from the driver.

- Stopping a device (`UBLK_CMD_STOP_DEV`). This command will halt the ublk device (`ublkbN`).

- Deleting a device (`UBLK_CMD_DEL_DEV`). This command will remove the character device (`ublkcN`). From this point on the device ID can be reused.

These are the most basic functionalities that `ublk-control` exposes. For a full list of available commands see [kD].

The high-level flow of adding a device in the system is depicted in Figure 3.13. The server sends an SQE to add the character device (1), customizing and revealing device information to the driver. Then it sets specific device parameters (2). Finally (3) after setting up all the userspace components, the ublk server sends the start command to inform the driver that everything is ready and the block device can be exposed in the system.

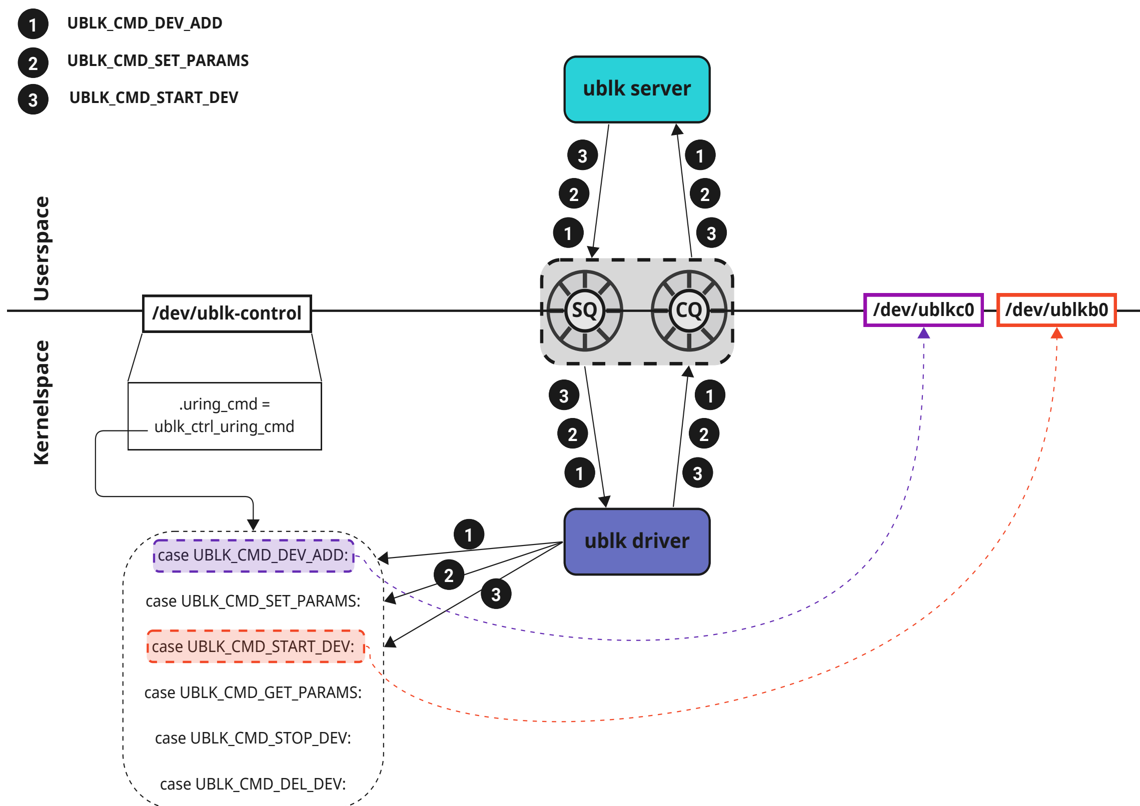

Figure 3.13: *Initial Commands to Add a Device*

A Deeper Dive into Setting up the Ublk Environment

Until now, we have presented a general overview of the control path of the ublk framework. Now, let's dive deeper into how this communication is set up: what userspace components are used and how they interact with the ublk driver.

Let's decipher the ublk server starting command:

```
./ublk add -t loop -f backing_file [-n device_id] [-q number_of_hw_queues] [-d queue_depth]
```

Listing 3.8: *Command to Add a Ublk Loop Device*

We will examine the most important steps this command takes. We will shift from userspace (server) to kernelspace (driver), and by the end, we will have a clear understanding of the essential parts and how they are set up along the way. This will be a more detailed explanation of the concepts we touched on previously.

Before that, let's understand what the given options do:

- `-t loop`: Identifies the type of the emulated device.
- `-n device_id`: Starts a device with specific device ID. (If not set, the driver will pick one by itself).
- `-f backing_file`: Indicates which file will be used as a disk.
- `-q number_of_hw_queues`: Specifies the number of queues the device will expose. These queues correspond to the hardware queues each real device exposes and match with the hardware queues of the block layer (in blk-mq design) for the specific device (default value: 1, maximum value: min {number of CPU cores, 32}).
- `-d queue_depth`: Specifies the size of each queue (default value: 128, maximum value: 4096).

The number of queues and the queue size determine how many on-the-fly requests our emulated block device can support. For example, if we have 2 queues and each queue has a depth of 1024, then we can have 2048 requests on-the-fly.

Note: These are not all the available configuration commands that can be given when starting the ublk server, but rather the most important for us to understand the concept.

With this information in mind, let's begin investigating how the command to add a ublk device is satisfied:
Step 1: The ublk server starts and saves any command line arguments in an internal structure.

Step 2: It opens the `/dev/ublk-control` special device exposed by the driver, and saves the file descriptor for later usage.

Step 3: It sets up an io_uring instance so that it can "talk" to the driver. Every SQE submitted from now on in this io_uring instance will have the open file from Step 2 as the `fd` field (see the SQE internals in Section 3.2.1).

Step 4: It submits the first SQE to the io_uring instance. This SQE has the opened file from Step 2 as the file descriptor and the opcode `IORING_OP_URING_CMD`. These two fields are crucial for the io_uring subsystem to locate the device that will serve this command. It also stores in another structure that ships with the SQE the type of command the driver is going to execute and the device information needed to fulfill this command. Remember that io_uring command passthrough is an ioctl-like command, so we need to specify which command the driver will execute. This command operation is `UBLK_CMD_ADD_DEV`. After the submission, the ublk server waits for the response and does not continue until the driver submits a CQE.

Step 5: The io_uring subsystem extracts the SQE from the Submission Queue and, using the file descriptor field (`fd` of `ublk-control`) along with the opcode (`IORING_OP_URING_CMD`), it locates the registered `struct file_operations` and calls the specific `ublk_ctrl_uring_cmd` function.

Step 6: The ublk driver runs and checks the command operation passed in the io_uring command passthrough to recognize the type of request it is serving. It then runs the

corresponding function that handles the addition of a device. This function is responsible for:

Step 6a: Allocating a structure named `ublk_device`, which will be the internal driver representation of the device.

Step 6b: Allocating a unique identifier and saving it in the `ublk_device` structure. If the server has requested a specific ID (via the `-n device_id` option), it asks for this ID, otherwise it allocates one on its own. This ID will be used later when the driver needs to retrieve the `ublk_device` structure.

Step 6c: It validates the information sent by the server and initializes internal fields of the `ublk_device`.

Step 6d: It allocates a specific structure for each hardware queue requested by the server (via the `-q nr_hw_queues` option). Each such structure serves as an information container for the queue. We'll see later how this corresponds `1-1` with a ublk server's thread.

Step 6e: It registers a blk-mq tag set. In Section 2.4.4, we analyze the importance of this structure. The block layer needs to know how many hardware queues are supposed to exist, what the number of each queue's capacity is, etc., in order to set up the block layer's relevant components.

Step 6f: Finally it adds the character device (`ublkcN`) in the system. From this point on, the ublk server can open and use this device by sending `IORING_OP_URING_CMD` commands.

Step 7: The driver fills a CQE and submits it into the Completion Queue.

Step 8: The server wakes up, checks the `res` of the CQE to verify that everything went ok, and forks a daemon child. This daemon, in turn, will create one thread for every hardware queue that our device has declared it can serve, and then it will wait for them (`pthread_join`). We will see exactly how the daemon with its threads are set up in the next section. For now, we can think of them as ublk server's services that will eventually serve an I/O request in the data path.

Step 9: After the daemon has been successfully set up, the ublk server submits another SQE with the `UBLK_CMD_SET_PARAMS` command operation flag set, to notify the kernel

of the device's parameters (as noted in the previous Section 3.4.2).

Step 10: The io_uring subsystem passes the request to the ublk driver, which, in turn, updates the internal parameter structure in its `ublk_device` descriptor. The driver then responds with a CQE to indicate that the operation has been completed.

Step 11: The ublk server retrieves the Completion Queue Entry and verifies that the parameter setting concluded without any issues. It then prepares and submits the third SQE with the operation `UBLK_CMD_START_DEV` to initiate the device startup process. As of this moment, there is no block device registered in the system.

Step 12: The io_uring subsystem forwards the request to the ublk driver, similar to steps 10 and 5. The ublk driver executes the corresponding function for the `UBLK_CMD_START_DEV` command. This function handles the registration of `ublkbN` in the system, following the steps outlined in Section 2.4.5. This involves the allocation of a `gendisk` descriptor along with a `request_queue`, utilizing the `blk_mq_tag_set` initialized in Step 6e. The block device is then added to the system using `add_disk()`. Just like previous submissions, the driver populates a CQE and submits it to the Completion Queue.

Step 13: The ublk server awakens, extracts the CQE, and verifies the outcome. It prints device-specific information to the standard output and then exits. From this point forward, the active components of the ublk server are the daemon threads, each dedicated to serving a specific hardware queue.

Note: This process exhibits a consistent pattern for each io_uring submission: the ublk server sends an SQE with the desired command operation and awaits the response. The ublk driver processes it and submits the corresponding CQE. The server then awakens, retrieves the CQE, verifies that everything has proceeded as expected, and moves forward. It's noteworthy that in this initial phase, the io_uring is employed in a synchronous manner. The ublk server waits after each SQE submission to obtain the CQE. This aligns with the request semantics. For instance, upon submitting the `UBLK_CMD_ADD_DEV` command, the server must ensure that everything proceeded smoothly before forking the daemon. Additionally, as previously mentioned, these requests entail a negotiation

between the server and driver. The driver may alter some of the provided parameters, and the server must be aware of these changes to appropriately initiate the daemon and the subsequent queue-threads.

3.4.3 Second Phase: Ublk Server Internal Setup

So far, we've covered the basic control path but haven't yet explored Step 8 from the previous steps, which involves crucial configuration on the server's side. In this section, we'll take a closer look at what happens in this step.

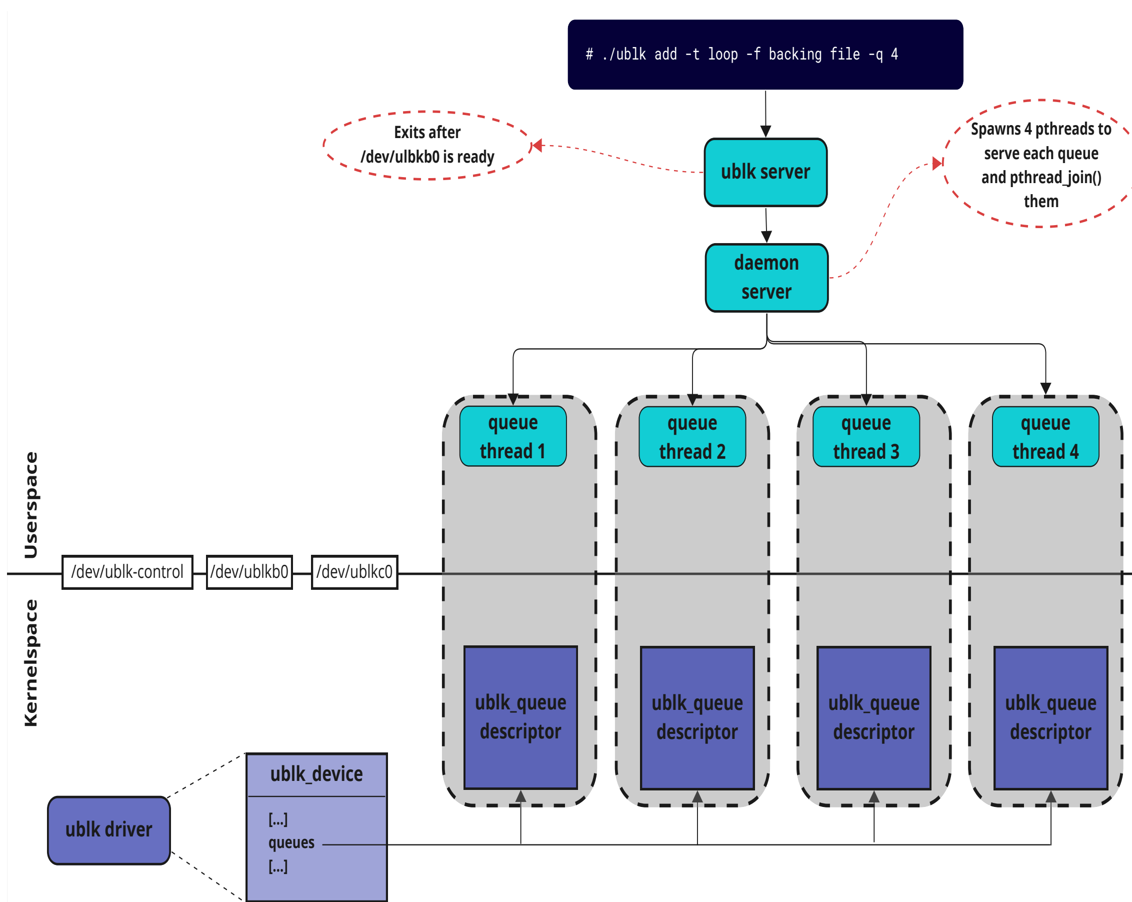

Figure 3.14: *One pthread for Every Queue*

In Figure 3.14, we illustrate the relationship between the initial ublk server entities and the one-to-one correspondence between in-kernel descriptors and userspace queue threads. In Step 6d, we saw that the driver allocates a specific structure for each queue. This is shown as "ublk_queue" in the figure. Also we used the term "ublk server" to refer to the initial process started by the "ublk add" command. The term "daemon server"

is used to identify the daemon spawned from the "ublk server", which in turn creates one pthread for each queue. While these are all part of the userspace entities in the ublk server, we use these terms for clarity. To reiterate, the "queue threads" depicted in Figure 3.14 are part of the ublk server.

Now, let's take a closer look into Step 8, to explore how userspace is configured, how the "queue threads" are created, and how they are prepared to facilitate the I/O path:

Step 1: After the ublk server retrieves the CQE in response to `UBLK_CMD_ADD_DEV`, it forks a child process, which in turn forks a daemon process (a process without access to the terminal) and then exits. The parent process waits until the daemon completes its setup. These two entities correspond to the "ublk server" and "daemon server" in the Figure 3.14.

Step 2: While the ublk server is waiting, the daemon server:
Step 2a: Opens the `/dev/ublkcN` character device and saves the file descriptor. It's important to note that if the device is already open, this operation will fail, preventing the character device from being opened again.
Step 2b: Executes a target-specific initialization callback function. This function probes and sets parameters for the backing file, known as the "target". At this point, we determine attributes such as the device size, the number of sectors, the block size, whether it operates in buffered or direct mode, etc.
Step 2c: After obtaining the target's parameters, the daemon opens a file in a predefined directory and writes its PID to this file.

Step 3: The "ublk server", which was waiting for the daemon to complete, reads this PID and proceeds with Step 9 from the previous list of numbered steps.

Step 4: The "daemon server" creates one thread for every queue and waits for them to finish. From this point onward, each of the following steps is executed independently by **every** queue thread.

Step 5: It maps a memory region of `ublkcN` using the file descriptor obtained in Step 2a. This memory region is pivotal in the ublk framework. Its size is determined by the

queue depth multiplied by the size of a descriptor called `ublksrv_io_desc`. Each descriptor provides information about an I/O operation. So, each queue thread possesses a dedicated memory-mapped area, which is read-only for the server and write-only for the driver. This helps the driver to describe the request sent from an application to the server.

Step 6: It allocates one buffer for each available queue slot. These buffers are used for serving read/write requests in the data path.

Step 7: It sets up an io_uring instance with a ring size of at least the queue depth. Each queue thread uses its own io_uring instance to communicate with the driver. Additionally, the same ring is used to communicate with the target (the backing file). We will see how this is achieved in the "third phase" section (3.4.4).

Step 8: It prepares a batch of SQEs equal to the queue depth of the Submission Queue. Each request is characterized by an `IORING_OP_URING_CMD`, and the specific command passthrough is set to `UBLK_IO_FETCH_REQ`. It is noteworthy that the server populates the queue depth SQEs but refrains from submitting them individually to the io_uring instance, thereby avoiding a system call for each SQE. Instead, they will be collectively submitted in the next step with **one** system call.

Step 9: With the initialization phase complete, each queue thread now starts its primary task. This task centers around an infinite loop. Within this loop, the thread both submits any pending SQEs and awaits one response in the form of a CQE. Once a response is received, it checks the type of CQE and proceeds with the corresponding action. Following the completion of request manipulation, the thread reverts back to the process of submitting SQEs and awaiting CQE(s). So, every queue thread continuously follows this loop.

In Figure 3.15, we've outlined the basic components discussed in the previous steps. In this example, we set up the ublk server with 2 queues, making use of the default queue depth (which is 128), since we didn't specify a different depth for each queue. This means the system can handle up to 256 I/O requests at the same time, 128 requests for each queue. Each Submission Queue contains 128 SQEs, each corresponding to an available

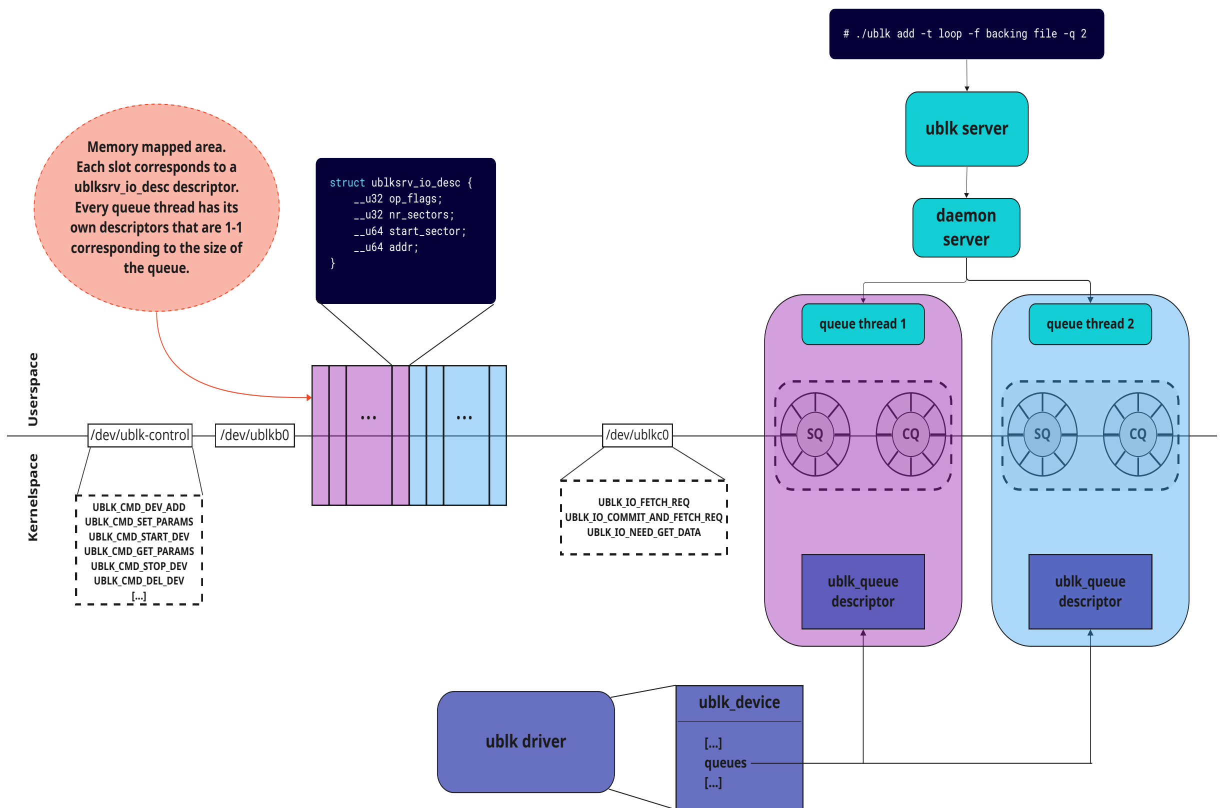

Figure 3.15: *Ublk Basic Components*

queue slot. Additionally, in the memory-mapped area, there are 256 `ublksrv_io_desc` descriptors, 128 for each queue. It's important to understand that the depth of each queue determines how many requests it can handle simultaneously. This, in turn, affects the size of the io_uring instance's ring and the number of descriptors in the memory-mapped area.

struct ublksrv_io_desc

The `ublksrv_io_desc` structure plays a vital role in the communication between the two ublk components.

```
struct ublksrv_io_desc {
  __u32 op_flags;
  __u32 nr_sectors
  __u64 start_sector;
  __u64 addr;
};
```

Listing 3.9: *struct ublksrv_io_desc*

The fields of this descriptor describe exactly what ublk server needs to know for any request:

- `op_flags`: Stores the type of operation (read or write) along with specific flags.
- `nr_sectors`: Specifies the number of sectors involved in the read/write operation.
- `start_sector`: Indicates the starting sector in the backing file.
- `addr`: Represents the address of a buffer in the ublk server's memory. This buffer is employed in the read/write operation and all of them are preallocated (see previous Step 6).

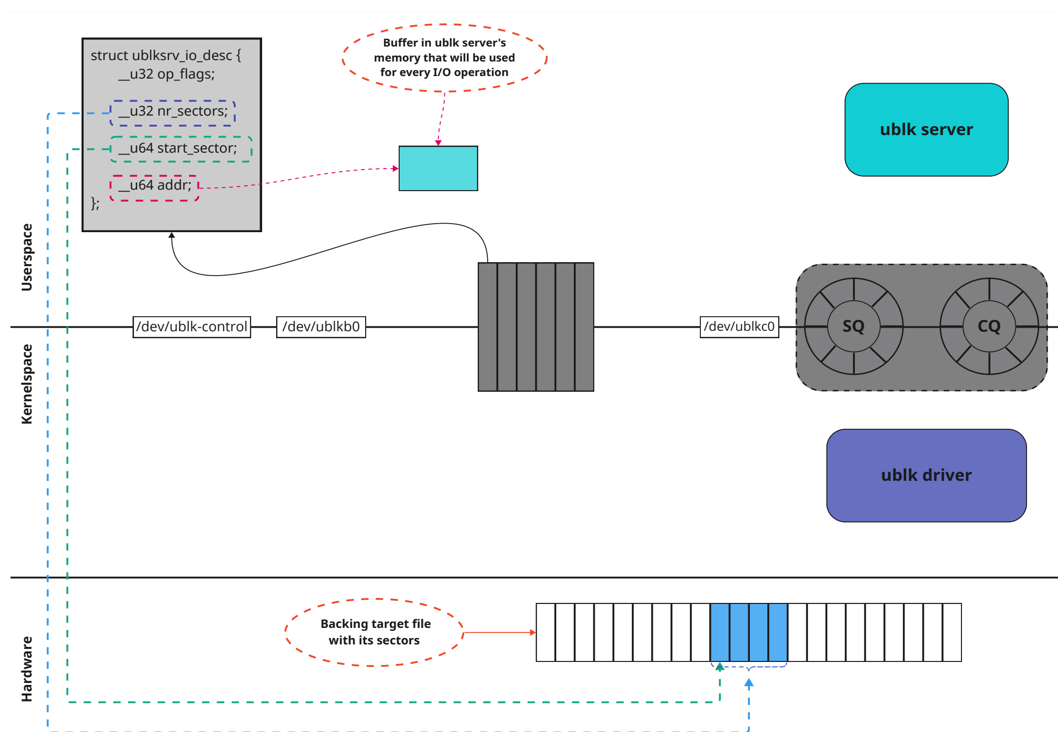

Figure 3.16: *A Zoom in ublksrv_io_desc*

The driver populates this descriptor to communicate the request details to the server. Equipped with this information, the server is ready to handle the request effectively. It knows the type of the operation, the starting point for reading/writing in the backing file (`start_sector`), the size of the request (`nr_sectors`), and, in the case of a write request, which buffer to use for writing data to the backing file, or in the case of a read

request, which buffer to fill with data from the backing file. All of these aspects are illustrated in Figure 3.16.

Tag-Based Communication

Each I/O request reaching the driver is assigned a unique number within the range of the queue's depth. This number accompanies the request throughout its processing cycle and allows the block layer to track the progress of each request. The tag number is unique within each queue slot. For instance, if our device has two hardware queues, each with 256 slots, then each queue slot is assigned a tag from 0 to 255.

The driver uses this tag number to find the corresponding `ublksrv_io_desc` structure and fills it with the necessary information for the request. This is why we have queue depth `ublksrv_io_desc` descriptors.

In steps 8 and 9, we noticed that each SQE sends an `UBLK_IO_FETCH_REQ` request to the driver. Now, let's take a closer look and examine the specific information contained within each SQE. This includes: (a) the tag of the request, (b) the address of the userspace buffer for I/O, and (c) the queue identifier (indicating from which queue this request comes).

The first SQE will be tagged as 0 and correspond to the first `ublksrv_io_desc`. The second will carry tag 1 and correspond to the second `ublksrv_io_desc`, and so forth. Consequently, when a request arrives from the block layer, the driver uses the tag to locate the corresponding `ublksrv_io_desc` in the memory-mapped area and complete the corresponding CQE for this tag. This is why we refer to this communication scheme as "tag-based".

This communication method is depicted in Figure 3.17. In this example, we show an instance of a ublk framework started with 2 hardware queues, and thus having two pthreads to serve each one of them ("queue thread 1" and "queue thread 2"). The block layer hands a read/write request to the driver for the first queue (queue thread 1) with a tag equal to 2 (1). The driver, based on the tag and the queue, identifies the appropriate `ublksrv_io_desc` descriptor (2) and updates it with the I/O information (3). Subsequently, it populates the CQE corresponding to the specific tag, prompting the server to wake up (4). The server then retrieves the `user_data` field of the CQE, which contains

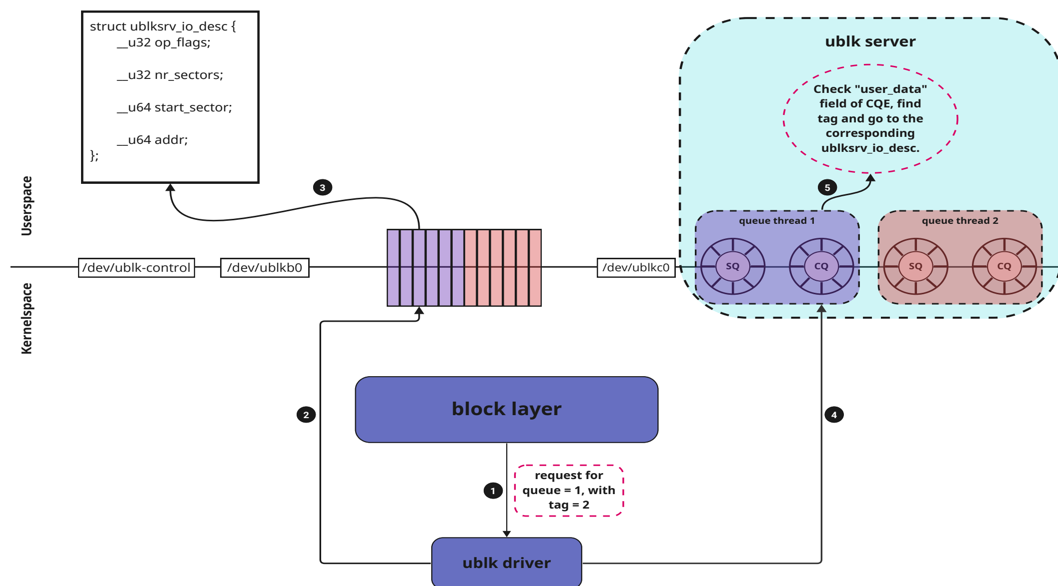

Figure 3.17: *Tag-Based Communication*

the tag of this request. Now, it can reference the descriptor to determine the details of the received request (5).

3.4.4 Third phase: The Data Path

Up until now, we've covered the setup phase and the roles of each important component. In this final section, we'll see how all the previous work together to fulfill an I/O request from an application. We'll follow both a read and a write request to understand how they are processed.

Along the way, we'll also explore how the ublk framework leverages coroutines, a concept discussed in Section 3.3.2.

Figure 3.18 assumes that the setup phase is complete, the ublk server has dispatched queue depth `UBLK_IO_FETCH_REQ` operations, and waits for an application to interact with the block device.

Write request:

Step 1: An application sends a write request, either directly to `ublkbN`, or via a filesystem. The request traverses the I/O stack (for more details about the I/O stack, refer to Section 2.4) and ends up at the ublk driver.

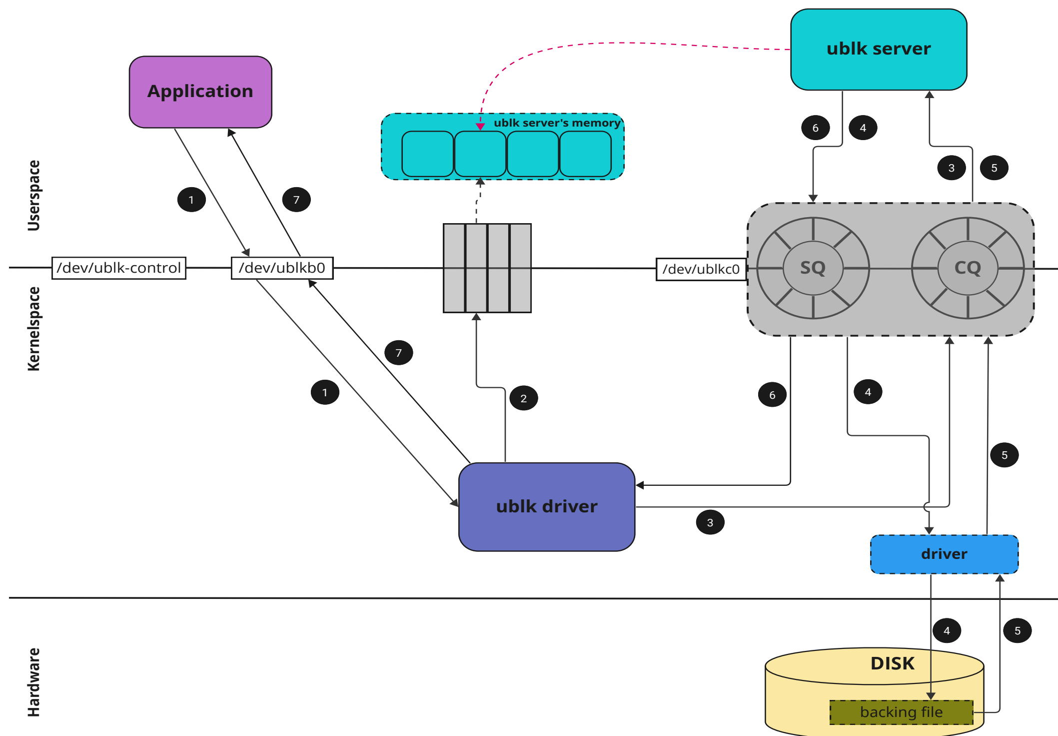

Figure 3.18: *I/O Request from an Application*

Step 2: The driver detects the incoming request, checks the tag, and informs the corresponding `ublksrv_io_desc`. Since it's a write request, it maps the provided application buffer and the ublk server's buffer (the one taken from the `ublksrv_io_desc`) to kernel addresses and copies the data from the first to the second.

Step 3: It submits the corresponding CQE to the Completion Queue. As a result, the ublk server wakes up to check the CQE. It specifically examines the `user_data` field to determine the type of request (see the Table 3.1 for a detailed layout of the `user_data` field). This 64-bit value indicates that the request originates from the driver. It also reveals the accompanying tag. With this tag, the server can retrieve the buffer containing the data ready for writing. It then calls a target-specific callback function. This callback, in turn, invokes a coroutine whose task is to prepare the corresponding write request. The coroutine doesn't submit the request at this point, instead it populates an SQE (of the same io_uring instance that communicates with the driver) with the write request details and suspends its execution using `co_await` (refer to 3.3.2, for more on `co_await` keyword).

Step 4: With the coroutine suspended, the server continues its loop, submitting any pending SQEs from the Submission Queue ring. There is at least one SQE ready for processing, prepared by the coroutine in the previous step. However, this time, the file descriptor of the request is not that of the `ublkcN` device, but rather the file descriptor of the backing file. Thus, after the submission, the generic io_uring code takes charge, fulfilling the write request. At this point, the data provided by the application is written to the backing file.

Step 5: A Completion Queue Entry with the response from the target awakens the server. The server examines the `user_data` field and identifies it as a response from the backing file. It then resumes the suspended coroutine. The coroutine finalizes this request by preparing a SQE, this time with the recipient being the ublk driver. The server needs to ack that the operation was successful (or not). Therefore, this request embeds a flag with the value `UBLK_IO_COMMIT_AND_FETCH_REQ`.

Step 6: The ublk driver receives the SQE. Upon recognizing the `UBLK_IO_COMMIT_AND_FETCH_REQ` flag, it performs two tasks: (a) It finalizes the request by providing the number of bytes written to the block layer, allowing it to propagate up to the application and (b) it prepares the environment for fetching future requests with the same tag.

Step 7: The completed request travels all the way up to the application, which now can check the result to confirm its success.

The steps for a **read request** are identical, except for the copying to and from the ublk server's I/O buffers. In Step 2 of a read request, there is no writing to an I/O buffer. Then, in Step 4, when the server submits a SQE with a read request to the backing file, the backing file populates an I/O buffer in the ublk server's memory with the requested data. Consequently, in Step 6, when the driver receives the SQE and identifies it as a read request, it copies the results from the ublk server's buffers to the application's buffers after mapping them to kernel addresses. The rest of the steps follow the same process as described above for the `write` request.

Note: The ublk server uses the same io_uring instance for both communicating with the target and the driver. The 64-bit `user_data` field is the only source of information upon

receiving a CQE. This field has the layout shown in Table 3.1, encompassing three vital pieces of information: (a) 0-15 bits denote the tag of the request. (b) 16-23 bits store the type of the operation, whether it is a read or a write request. (c) The 63rd bit denotes if it is a response from the target or a request from the driver. Bits 24-39 and 40-62 are unused in this context.

Thus, in Steps 3 and 5, the "is_target_io" bit (the last bit) of the `user_data` is the one that informs the server about the initiator of the request (target I/O or driver request). Based on this information, it determines the subsequent action.

63	62-40	39-24	23-16	15-0
is_target_io	unused	tgt_data	op	tag

Table 3.1: *user_data Layout per Bits*

We've now covered the core functionality of the ublk framework. We've seen how it sets up various components, spawns threads for each queue, and ultimately processes real I/O operations on the emulated block device through the collaboration of the ublk driver and ublk server.

Of course, there are some attributes that we haven't touched on this journey, as they won't play a role in our encryption scheme.

3.5 Encrypted Ublk

From the design of the ublk framework (Section 3.4), it is evident that no cryptographic operations occur in either the ublk server or the driver. Consequently, when an application utilizing ublk sends data, they are simply copied into the ublk server's virtual memory (by the ublk driver) and subsequently forwarded to the backing file for storage. This way the data provided by an application are saved in exactly the same format as acquired from the block layer by the ublk driver.

As a result, an application employing ublk and aiming to secure its data must be aware of the environment in which the ublk framework operates. For instance, it should consider whether the disk on which the data is eventually stored is a Self-Encrypted Disk, or if the filesystem itself uses any form of encryption. Therefore, it cannot rely solely on the

ublk framework for data security. If the environment is not secure, the application must take proactive measures by encrypting the data prior to interacting with the disk.

Our aim was to enable applications to use the ublk framework securely, without necessitating extensive knowledge about the underlying environment. To achieve this, we integrated an encryption scheme directly within ublk, implemented entirely in userspace. This integration guarantees that no data will be written to the backing file in an unencrypted form.

In this section, we will discuss the key architectural decisions we made to implement encryption within the ublk framework. As mentioned in the Introduction chapter, we achieved encryption with three three distinct methods:

- **Single-thread encryption:** Ublk server itself handles the encryption of each buffer.
- **Intra-block parallelism:** Ublk server splits the buffer, and activates a thread pool where each thread is responsible for encrypting/decrypting a specific range in the buffer, while the server waits for them to finish.
- **Inter-block parallelism:** Ublk server offloads the whole buffer to a thread from a thread pool and continues.

While each of these methods has a different approach to encryption, they share a common method for key setup. The term "key setup" refers to how we incorporate a cryptographic key into ublk, how we store it, and how it is utilized for encryption operations. This setup is the same across all three implementations.

To provide a more organized presentation, we have divided this discussion into five sections. We will start with an overview of the Encrypted Ublk. Then, we will move on outlining our decisions regarding the key setup phase, which is common to all the implementations. Following that, we will go into the specific design choices for each implementation.

3.5.1 Overview of Encrypted Ublk

At the beginning, we needed to decide on the type of encryption to use: symmetric or asymmetric. This was rather easy to decide. As we discussed in Section 2.5.2 symmetric

encryption is the way to go when we need to quickly encrypt large amounts of data. Given that our system requires on-the-fly encryption and decryption for each write/read operation, opting for a symmetric encryption algorithm was clear.

Once we settled on symmetric encryption, the next question was: which symmetric encryption method should we use? This one was also straightforward: AES. It's widely used, secure, and fast. AES stands as the de facto standard for the majority of the online cryptographic operations employing symmetric cryptography.

However, as we explained in Section 3.6.2, AES can work in different modes. So, the last question was: which mode should we use? This decision wasn't as easy as the previous ones, but we ended up going with XTS mode for a few good reasons. We explained the advantages of AES-XTS in Section 3.6.3. In short, XTS mode is tailored specifically for disk encryption, serves as the default mode of operation for many major cryptographic distributors, and facilitates parallel encryption and decryption. This last point was important because we wanted to explore how parallelism affects the data path and if different parallel strategies make a noticeable difference.

Consequently, the chosen encryption algorithm was AES, operated in XTS mode with a 256-bit key size. Since XTS mode uses a key size twice the size of AES, we need a 512-bit key. This means that during the use of the encrypted ublk, a 512-bit key (64 bytes) resides in memory in unencrypted form and is utilized for every encryption and decryption operation. This key will be referred to as the "master key" or "data key".

Figure 3.19 provides a high-level visualization of the ublk encryption process. "P" stands for the original, unencrypted data (Plaintext), and "C" represents the encrypted data (Ciphertext), enabling a clear depiction of the data's path. An application initiates a write request, the driver maps its buffer(s) into the kernel memory, and copies the data to the ublk server's memory buffers (the data path in ublk is detailed in Section 3.4.4). Subsequently, the ublk server uses the master key to encrypt the buffer before forwarding it to the backing file, ensuring that everything is transmitted in encrypted form. For brevity and to emphasize the data encryption, we omit the io_uring instance from this diagram.

The same process applies to a read request from an application. The ublk server retrieves the encrypted data from the backing file, decrypts it using the master key, and stores the plain data in its buffers. Then, once the server notifies the driver, the driver copies the

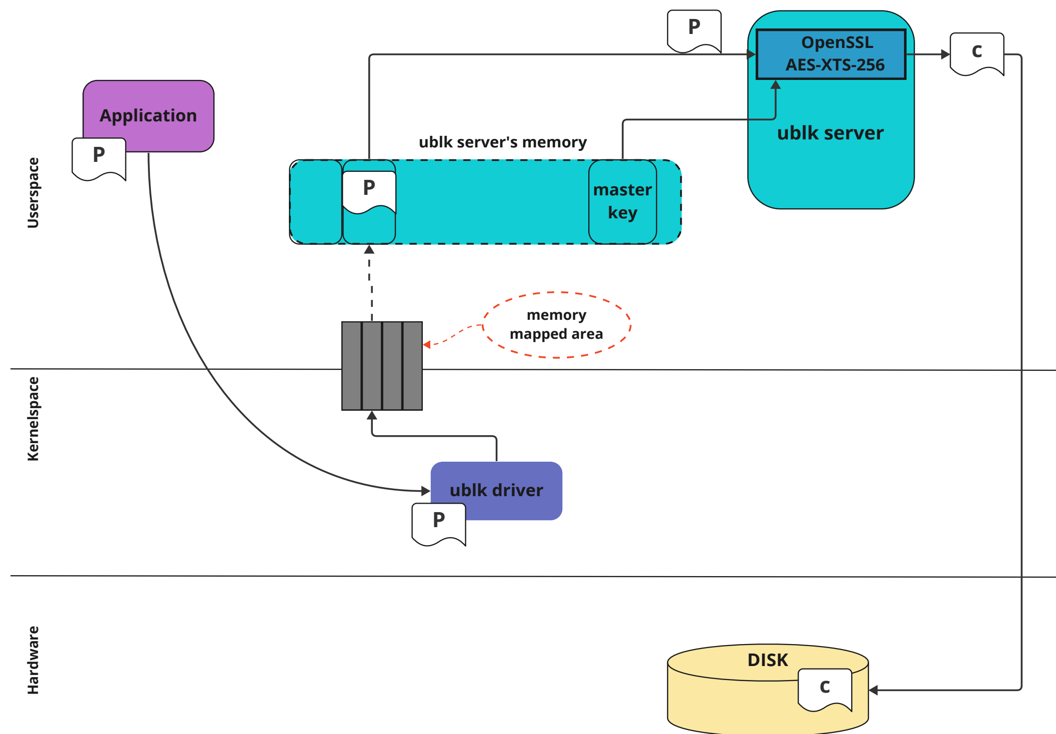

Figure 3.19: *General Overview of a Write Request in Encrypted Ublk*

unencrypted data from the server's memory to the application's buffer, completing the read request. Since AES is a symmetric encryption method, the key stays the same for both encryption and decryption.

With this general overview in mind, we are prepared to examine precisely how we manage the key setup and ultimately how we design each of the three encryption methods.

3.5.2 Key Setup design

After deciding on the type of encryption, we faced the question of how to handle the master key that is going to be used for data encryption. We wanted the master key to be saved in a file so that we can use it again. However, it wouldn't be a wise move to save the master key unencrypted in a file. It must be saved in an encrypted form. Additionally, we wanted to be able to add new keys and remove keys without changing the master key itself. This is because if the master key has to change in order to add or change a key, it would require re-encrypting the whole disk, which is not practical for data mediums that may store a large amount of data.

This idea led us to adopt a scheme of **key hierarchies**. We wanted another key that would

be used to encrypt the master-data key. This concept enables us to handle encryption with more flexibility (see more about key hierarchies in Section 3.7.1).

Also we wanted our key setup, to support the following high level operations:

- Create a new cryptographic disk. Initialize a master key, and save it encrypted in a file.
- Open an encrypted disk by recovering the master key, provided in encrypted form from a file.
- Add a new key.
- Remove a key.

We will follow each of these operations and see how exactly they function in our design. In order to be able to control some of the above high-level operations, we believed it would be better to allocate a range on the disk for metadata information. That's why we chose to allocate the first block of the disk (4096 bytes) for metadata reasons. In other words, the backing file that serves as a disk has the layout shown in Figure 3.20. The bulk data are saved after the "metadata area".

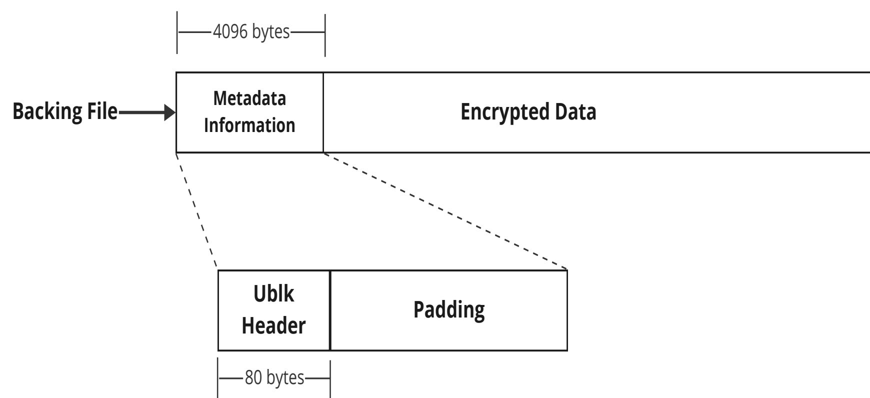

Figure 3.20: *Backing File Layout*

The header information is stored in a structure in this first version of Encrypted Ublk and it contains four fields:

1. The magic number for a ublk encryption disk.

2. The version of the encryption.
3. A flag that verifies if this disk is active (i.e. it has a valid master key) or not.
4. A hash of the master key.

Of course, this structure doesn't take up the whole metadata block. But we designed it this way in order to be extendable for later usages, and aligned with block size which is the basic operation unit from the OS's side.

Now, let's see how we respond to the high-level operations in our design, and what exactly is the role of the metadata `ublk_header` in each phase.

Create an encrypted disk

To initiate the ublk encryption, we must create an encrypted disk the first time we start the server. As discussed in Section 3.4.1, starting the ublk server necessitates specifying the file that will serve as a storage medium. This "disk" will house both metadata and the actual encrypted data.

We added an additional option to designate the name of the file that will contain the encrypted master key:

```
./ublk add -t loop -f backing_file -s <file_to_save_master_key>
```

Listing 3.10: *Start Encryption in Ublk*

Let's examine the basic steps undertaken when initiating our encryption scheme. Algorithm 1 outlines the process at a high level. The core design is as follows: We read `ublk_header` size from the beginning of the disk to check if it is an active disk. If it's active, it indicates that encryption is already in place, and to prevent user shooting his own leg, the operation is aborted. If the disk is not encrypted, we read 512 bits from a random source to serve as the master key. Subsequently, we: (a) Encrypt the master key, saving its encrypted form in the file passed via the `-s` option, and (b) Initialize the `ublk_header` struct with the relevant information, storing it at the beginning of the backing file.

A critical question centers around the method used to encrypt the master key. While the implementation details will be explored in the next chapter, for now, it suffices to say that

we leverage GnuPG for symmetric encryption of the master key before saving it to the file. By default, GnuPG adopts AES for symmetric encryption, requesting a password to secure the file contents in which it will be saved the encrypted key. We use the GPGME library [GNUa] to interface with the GPG agent, with more details on GPGME and the key encryption implementation to be provided in the subsequent chapter.

Now, with the master key loaded into memory, we can proceed with the remaining ublk server initializations. The encrypted master key, stored in the specified file, is now set aside for future use.

Algorithm 1 Initialize an encrypted disk

```
1: disk_fd ← open(backing_file)
2: master_file_fd ← open(file_to_save_master_key)
3: ublk_header ← read from disk_fd the first 80 bytes
4: if ublk_header.active is true then
5:     if ublk_header.magic == “UBLKEN” then
6:         ABORT. The disk is already encrypted.
7:     end if
8: end if
9: master_key ← read 64 bytes (512 bits) from /dev/random
10: master_file_fd ← Encrypt(master_key)
11: md_key ← SHA512(master_key)
12: ublk_header.magic ← “UBLKEN”
13: ublk_header.version ← “0.1”
14: ublk_header.active ← 1
15: ublk_header.master_key_hash ← md_key
16: Write ublk_header in disk_fd, starting from byte 0
```

Open an Encrypted Disk

What if we want to reuse a disk that's already encrypted? To do so, we need to recover the master key. That's why, when starting the ublk server, we pass a file containing the encrypted master key via the command line. Of course, we need to decrypt this file and verify if the result is indeed the valid master key.

To enable this functionality, we add the `-e <file>` option:

```
./ublk add -t loop -f backing_file -e <file_with_saved_enc_master_key>
```

Listing 3.11: *Open an Encrypted Ublk Disk*

The process is outlined in Algorithm 2, and, in short, the steps we follow are: (a) read the `ublk_header` from the metadata section of the disk, (b) check if it's a valid ublk encryption disk, (c) decrypt the file containing the candidate master key, (d) hash the decrypted candidate master key, and finally (e) compare it with the hash stored in the `ublk_header`. If the hashes match, this indicates that the candidate key is indeed the master key. If the hashes match, it means that the candidate key is indeed the master key, and we proceed with the master key saved in memory, ready to be used in any I/O encryption operation. If the hashes don't match, we abort.

Algorithm 2 Open an Encrypted Disk

1: disk_fd ← open(backing_file)
2: cand_file_fd ← open(file_with_saved_enc_master_key)
3: ublk_header ← read from disk_fd the first 80 bytes
4: **if** ublk_header.active is false || ublk_header.magic != "UBLKEN" **then**
5: ABORT
6: **end if**
7: candidate_master_key ← Decrypt(cand_file_fd)
8: md_candidate_key ← SHA512(candidate_master_key)
9: **if** md_candidate_key == ublk_header.master_key_hash **then**
10: PASS. The recovery of the master key was successful.
11: **else**
12: ABORT. The given key does not match with the master key.
13: **end if**

Add a New Key

To support this operation, we introduced a new command within the ublk framework. Adding a new key is an action independent from starting the ublk server. The previous two high-level operations involved starting the server, which is why they were options in the "`./ublk add`" command.

When a new key is added, the master key remains unchanged. What happens instead is that we re-encrypt the master key using a different password, and save the outcome in a new file. So, although the files are different, they house the same master key, the only difference is the encryption key applied on the master key.

The new command is called `add_new_key` and can be used as follows:

```
1 ./ublk add_new_key -f <backing_file> -s <file_with_saved_enc_master_key> -e <
    new_file>
```

Listing 3.12: *Add a New Key*

When we execute this command, it will re-encrypt the master key and save it in the `<new_file>`. From this point onwards, to start the ublk server and recover the master key, as demonstrated in Section 3.5.2, we can use either a previously added "master key file" or the `<new_file>` as the key. The result will be the same, and the decryption process will successfully yield the same master key.

In Algorithm 3, we outline the steps to follow when adding a new key. Practically, the entire procedure resembles Algorithm 2 because we choose to add a new key only if a valid encrypted master key is provided. Thus, we need to specify a valid file that contains the correct encrypted master key. This file is decrypted, hashed, and then checked against the saved hash of the master key in `ublk_header`. If they match, it indicates that we provided the correct master key. We then re-encrypt it and save the new encryption in the `<new_file>` passed as an input parameter. After adding the key, the user can use `<new_file>` to start the ublk framework.

Algorithm 3 Add new key

```
1:  disk_fd ← open(backing_file)
2:  cand_file_fd ← open(file_with_saved_enc_master_key)
3:  new_file_fd ← open(new_file)
4:  ublk_header ← read from disk_fd the first 80 bytes
5:  if ublk_header.active is false || ublk_header.magic != "UBLKEN" then
6:      ABORT
7:  end if
8:  candidate_master_key ← Decrypt(cand_file_fd)
9:  md_candidate_key ← SHA512(candidate_master_key)
10: if md_candidate_key == ublk_header.master_key_hash then
11:     new_file_fd ← Encrypt(candidate_master_key)
12: else
13:     ABORT. The given key does not match with the master key.
14: end if
```

The Figure 3.21 illustrates the connection between the first three high-level operations. After adding a new key, you can open the Encrypted Disk and run the ublk server with any file that stores a valid encrypted version of the master key.

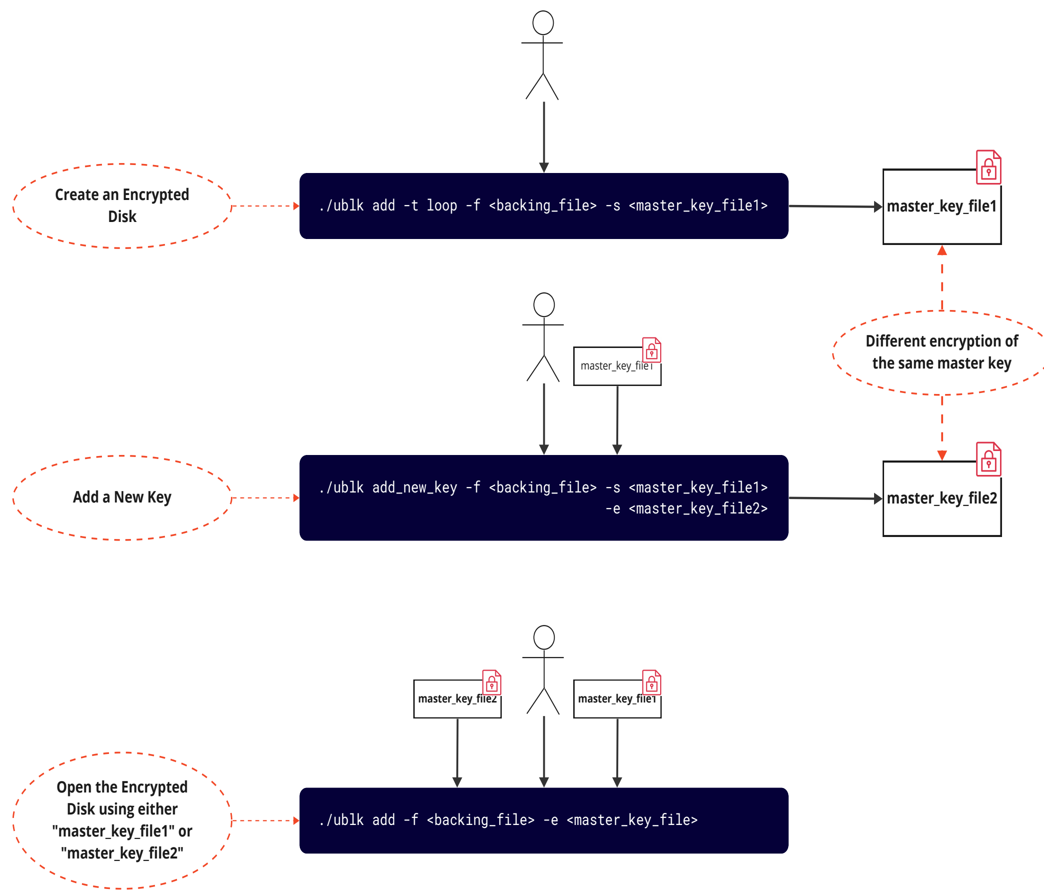

Figure 3.21: *Create, Add and Open Operations*

Remove a Key

Removing a key is a straightforward operation. To do so, we simply delete a file that contains the encrypted master key. We don't need to involve the ublk server at any point during this operation. There's no information stored about added keys in the metadata area. As long as we provide a valid file containing the encrypted master key, we can create a new encrypted master key file. Therefore, to remove a key, all we need to do is delete the file containing the encrypted master key.

3.5.3 Single-Thread Encryption

Having addressed the key setup problem, we can now discuss how we integrate encryption inside the ublk server, starting with our first solution: **single-thread encryption**.

By "single-thread", we mean that each ublk server thread responsible for communication with the driver and the target also performs encryption and decryption. As detailed in

Section 3.4, the ublk server consists of daemon threads, with each thread serving one queue. Each thread communicates via its io_uring instance with the ublk driver and the target.

In this design, each ublk server thread follows a loop where it submits all available SQEs in its io_uring and waits for at least one CQE. Since each thread uses the same io_uring instance for communication with both the ublk driver and the target, SQEs can be directed to either destination. Therefore, when a ublk server thread wakes up with a response in the form of a CQE, it can be attributed to one of four possible scenarios:

- **Ublk driver completes a CQE for a write request:** In this case, the server locates the data to be written in a specific buffer, and it must then submit a write SQE to the target to transmit the data.
- **Ublk driver completes a CQE for a read request:** The server needs to retrieve the data from the target (i.e. submit a read SQE to the target).
- **Target responds after a write request from the server:** The server's responsibility is to inform the driver that the request was successful (or not) submitting a SQE with `UBLK_IO_COMMIT_AND_FETCH_REQ`.
- **Target responds after a read request from the server:** In this scenario, the server needs to locate the data it requested to read from the target. The data can be found in a predefined buffer, and the server must then inform the driver submitting again a SQE with `UBLK_IO_COMMIT_AND_FETCH_REQ`.

As we discussed earlier (see Figure 3.19), the ublk server incorporates dedicated buffers to facilitate data transmission between the application and the target. When a write request arrives from the driver, the ublk server knows where to locate the data that needs to be written. Conversely, when a read request is received, the ublk server knows where to store the data after reading it from the target.

These buffers are preallocated during the ublk server's initialization, and their quantity matches the queue depth. Each buffer can store the intermediate result of one I/O operation, so the number of buffers must align with the outstanding requests.

In our design, we required an additional set of buffers equal to the queue depth to store the encrypted results. While we could allocate each buffer dynamically when needed in

order to save space, our primary concern was efficiency and speed. Allocating buffers on-the-fly in the critical path of an I/O request would have significantly impacted efficiency. For this reason, we decided to preallocate a set of buffers for each queue, similar to the buffers allocated by the ublk server. We'll refer to these preallocated buffers as **"temporary buffers"** to distinguish them from the **"original buffers"**.

The Figure 3.22 provides a clear depiction of some key concepts and illustrates the 1-1 correlation between the "original buffers" and our "temporary buffers". In this figure, we observe a running ublk server instance equipped with three queues. Each thread operates with its own io_uring instance and interacts with a distinct memory-mapped area. Additionally, every thread maintains its set of buffers, with each buffer having a corresponding temporary buffer designated for encryption purposes. Note that, the size of each ring aligns with the number of slots in the memory-mapped area, which, in turn, corresponds to the number of buffers and matches the quantity of outstanding requests that a queue can serve.

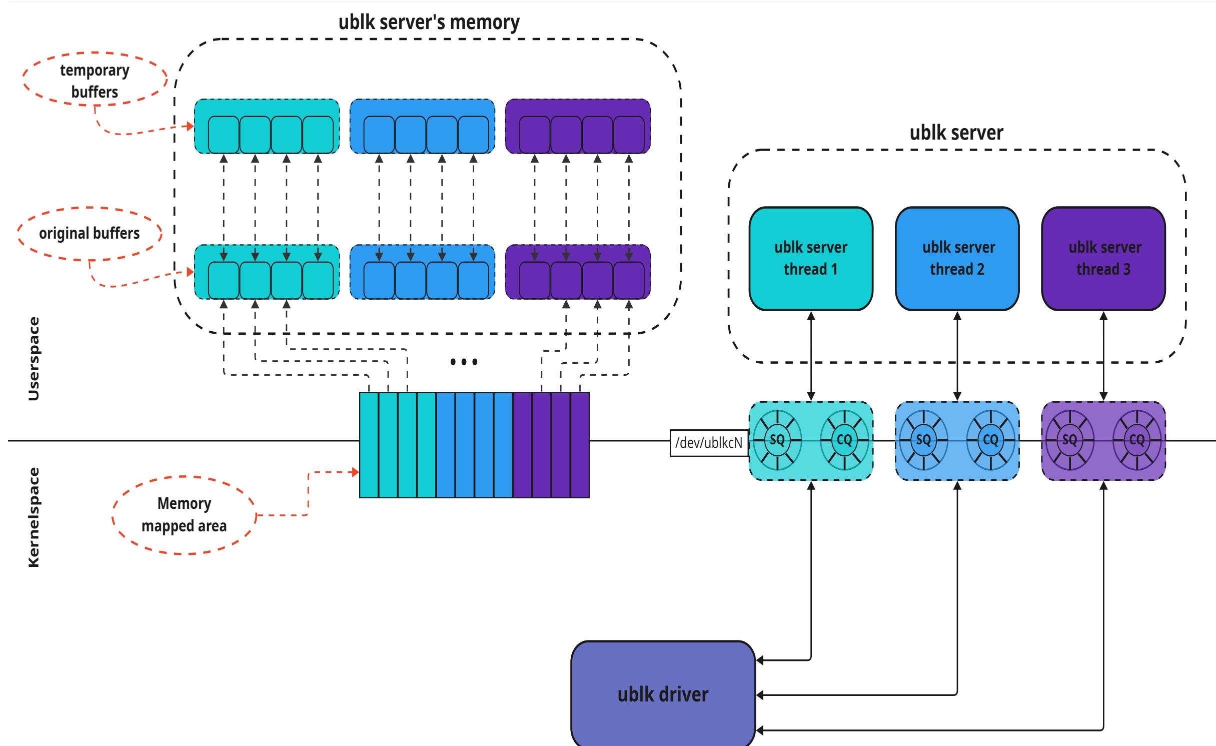

Figure 3.22: *Correlation Between Original Buffers and Temporary Buffers*

To enable encryption, we must extend the four previously mentioned actions when the server wakes up with a CQE to include the following:

- **Ublk driver completes a CQE for a write request:** The server locates the data to be written in a specific "original buffer". Then , it encrypts the data from this buffer, saves the encrypted results in the corresponding "temporary buffer", and finally writes the corresponding region from the "temporary buffer" to the target file.

- **Ublk driver completes a CQE for a read request:** The ublk server performs a read request to the target but doesn't save the results in the "original buffers". Instead, it initiates a read request and reads the (encrypted) data into the "temporary buffers".

- **Target responds after a write request from the server:** The server doesn't need to perform any additional steps. It simply submits an SQE with `UBLK_IO_COMMIT_AND_FETCH_REQ` to inform the driver.

- **Target responds after a read request:** When the target responds after a read request, it implies that the encrypted data resides in a "temporary buffer". At this point, the server must decrypt the data, save them in the "original buffer", and then submit an SQE with `UBLK_IO_COMMIT_AND_FETCH_REQ` to inform the driver.

Our intervention in the I/O path is crucial to ensure encryption occurs before saving data to the backing file, and decryption takes place before passing the results back to the driver and the application.

Each thread in the ublk server conducts encryption and decryption independently using AES in XTS mode with the key stored in memory from system initialization. We use the OpenSSL library for these cryptographic operations. While we will provide a detailed implementation overview in the next chapter, for now, consider the OpenSSL library calls as black boxes that take plaintext as input and produce ciphertext for encryption, and vice versa for decryption. The OpenSSL library operates in userspace, performing mathematical computations to achieve the desired results.

Encryption and decryption using the OpenSSL library occur at the sector level. Our buffers, both "original" and "temporary", can each store up to 0.5MB (2^{19} bytes). Therefore, the ublk server can handle requests that read or write up to 1024 sectors. So, we must repetitively invoke the encryption/decryption library function for each distinct

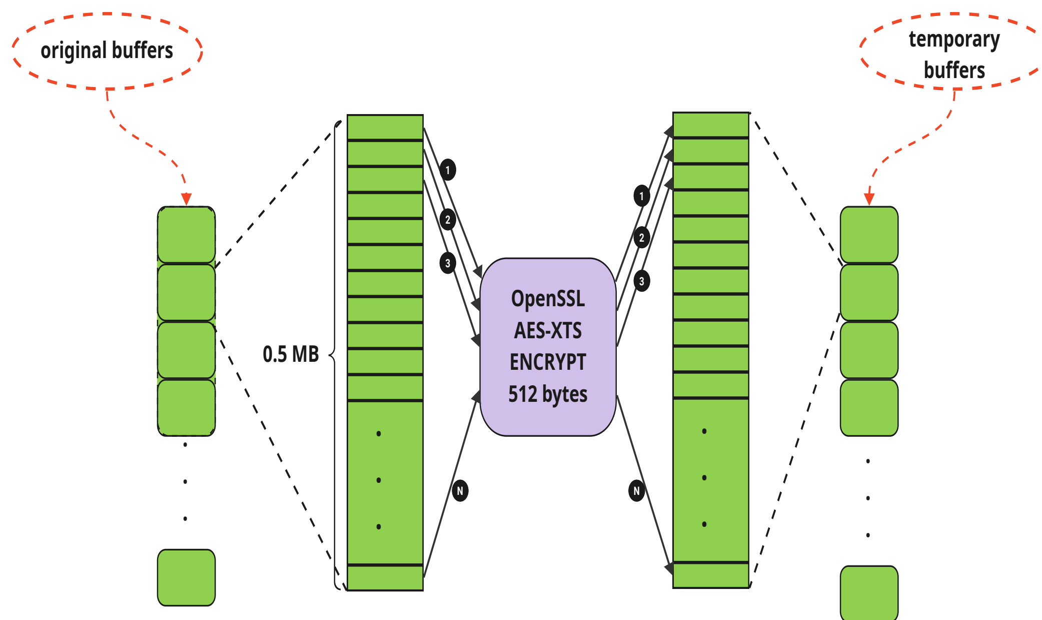

Figure 3.23: *Encryption Process on Each 512-byte Section*

512-byte chunk, which corresponds to a sector on the disk. Figure 3.23 illustrates this behavior in the encryption process of a request that is 0.5MB (N equals 1024 in this case). The decryption process is similar, with data being decrypted from the "temporary buffers" and stored in the "original buffers". Each encryption operation for every 512-byte chunk is carried out sequentially.

In Figures 3.24 and 3.25, we present the data path for a write and a read request, respectively. To enhance clarity, we depict a ublk server instance running with a single queue. We can compare this path with the original I/O path presented in 3.4.4 to identify the differences between read and write requests.

For a **write request**, the process begins with an application submitting a buffer for writing (1). The write request travels through the I/O stack and eventually reaches the ublk driver from the block layer. The driver identifies the request type (read or write), and copies the application's buffer to the ublk server's memory buffers (2). Then it completes a CQE to notify the server (3). Upon waking up (3), the server recognizes that a CQE for a write request arrived from the driver, locates the buffer, sends the data to the OpenSSL function for encryption, and stores the encrypted data in a temporary buffer (4). Next, it submits an SQE to the target for the encrypted buffer (5). After the disk write operation concludes, the target responds with a CQE (6). The server wakes up,

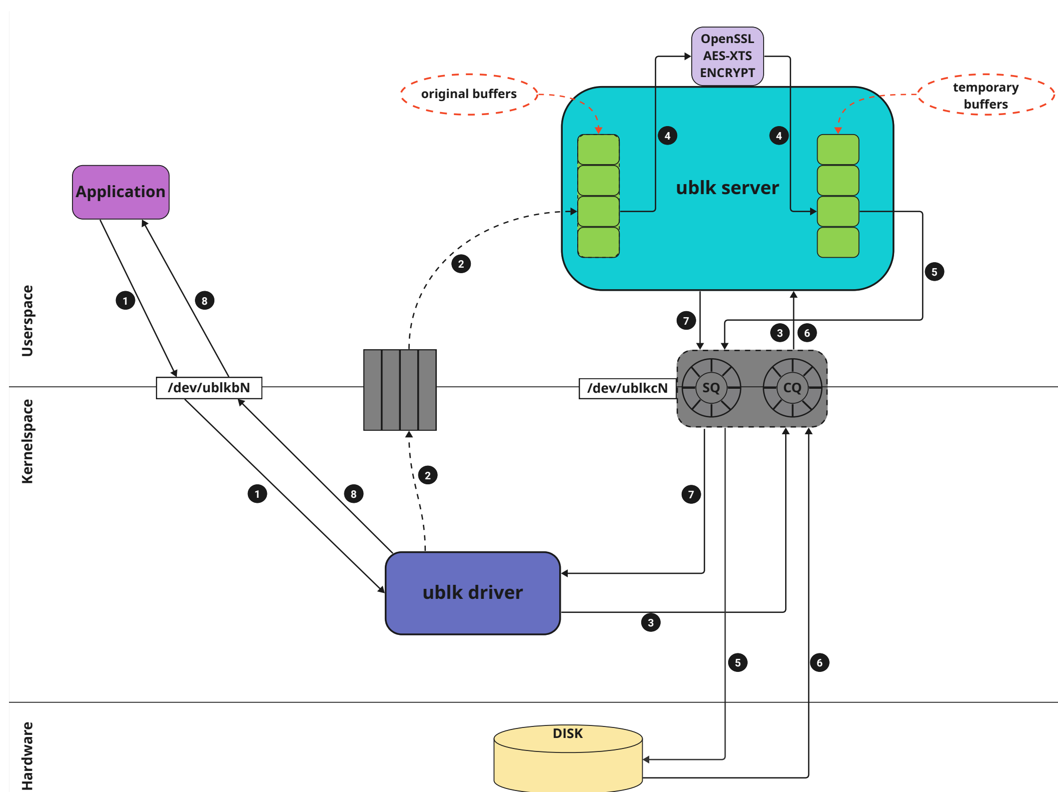

Figure 3.24: *Data Path for a Write Request in the Ublk Single-Thread*

identifies the response from the target, and submits a SQE to the driver to report the operation's result (7). The driver receives the result, completes the request, and passes the response (the number of bytes written or an error) to the upper layers (8).

For a **read request**, the driver doesn't need to perform any copying before submitting the CQE to notify the server (2). Upon waking up, the server identifies the read request from the driver and submits a SQE for a read request to the target, specifying the temporary buffer as the destination buffer of the request (3). Once the requested copy from the device to the temporary buffer is complete, the target notifies the server by submitting a CQE (4). The server wakes up, recognizes that the read request from the target has finished, indicating that the encrypted data is available in a temporary buffer. It then processes the buffer through the appropriate OpenSSL function for decryption, storing the decrypted data in the corresponding original buffer (5). Subsequently, the server submits an SQE with the driver as the recipient (6). The driver receives the SQE, identifies it as a response from the server for a read request, copies the server's buffer into the application's buffer and then completes the request and provides the response (the number of bytes read or an error) to the upper layers (7, 8).

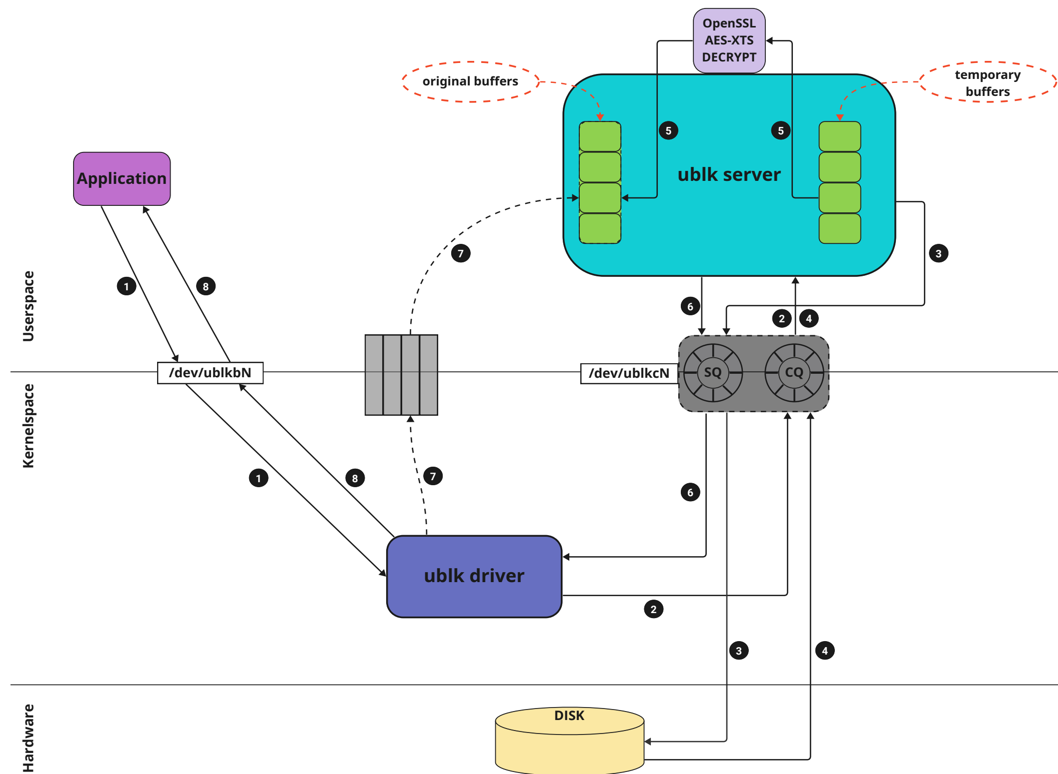

Figure 3.25: *Data Path for a Read Request in the Ublk Single-Thread*

3.5.4 Intra-Block Encryption

Our second solution to the encryption problem is based on the fact that AES-XTS encryption and decryption can be performed in parallel. As discussed in Section 3.6.3, the encryption and decryption of each sector do not depend on anything other than the sector number, which serves as a tweak value for a single sector's encryption/decryption. Consequently, there is no impediment to parallelizing the encryption of a buffer larger than 512 bytes (1 sector).

In the single-thread implementation, the ublk server encrypts/decrypts every 512-byte segment within each buffer sequentially. It invokes an OpenSSL library function as many times as necessary to complete the task, as depicted in Figure 3.23.

In this new implementation, each ublk server's thread creates a thread pool and **distributes** the work to the threads within the pool. For clarity, we will refer to the ublk server thread as the "main thread" and the threads within the pool as "working threads" or "workers". When the time comes to perform encryption or decryption, the main thread no longer handles these tasks directly. Instead, it informs a shared structure that

"instructs" the working threads about the type of work they should perform. It then waits for them to finish the encryption or decryption.

Introducing a thread pool raises several questions, such as how the pool will be created, how communication between the main thread and the working threads will be managed, how the pool will be destroyed etc.

Starting with the first question, we've introduced an option at the beginning of the ublk server that allows specifying the number of threads in the thread pool. The maximum allowed value is 64 threads, and the default setting is 8 threads. It's worth noting that each ublk server "queue-thread" maintains its own pool of working threads.

One key point to emphasize is that the points at which the encryption and decryption are performed remain unchanged compared to the single-thread solution. The main difference lies in who performs the encryption and decryption. For example, when a write request is received from the driver, the server distributes the work to the worker threads as follows: each thread takes responsibility for encrypting one 512-byte segment, according to its position. Once a segment is encrypted, the threads compute a stride and proceed to encrypt another segment, and so on until the requested size is processed. A similar distribution process occurs when a response from a read request arrives from the target. In this case, the server assigns the work to the worker threads, tasking them with decrypting the data based on their positions, following the same logic as before.

Figure 3.26 provides a visual representation of this work distribution.

In the core of our design, exists a shared object that facilitates the communication between the working threads and the main thread. This shared object stores essential information that enables the main thread to convey specific details about each incoming request to the working threads. This information includes the nature of the task to be performed (encryption or decryption), the source and destination buffers, the request size, the starting sector etc. This shared object represents the communication channel from the main thread to the working threads. Conversely, no explicit sign is needed for communication from the working threads to the main thread, as the main thread already knows what kind of task it has delegated to the working threads. Hence, upon waking from barrier2, it understands that the job has been completed, as will be examine below.

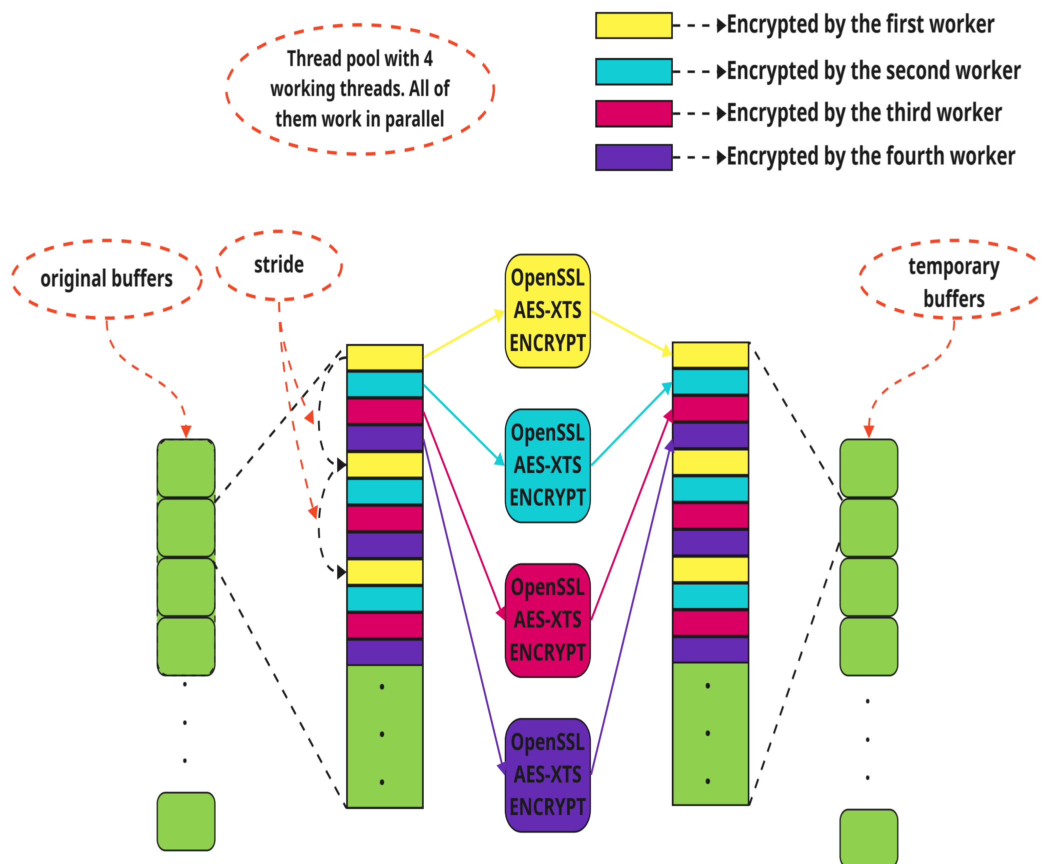

Figure 3.26: *Work Distribution Among Workers*

The second crucial aspect of our design concerns the **synchronization** between the main thread and the working threads. To achieve this, we use two barriers (`pthread_barrier_t`) within the shared object. A barrier works as follows: during the initialization of the barrier you denote a "count". The count argument specifies the number of threads that must call `pthread_barrier_wait()` before any of them successfully return from the call [mpf]. In other words, in a barrier you denote a number, and as long as the number of threads that have called this barrier (via `pthread_barrier_wait()`) has not reached the number you denoted, all threads remain blocked at the barrier. It is the last thread to "reach" the barrier that subsequently unblocks all other waiting threads.

Both barriers in our implementation were initialized with $N + 1$, where N is the number of working threads. Let's follow the Figure 3.27, in order to understand better the synchronization design. Initially, the working threads wait on the first barrier, referred to as barrier1, to receive a job (1). The main thread, wakes up by a CQE, and processes either a write request from the driver or a response to a read request from the target, re-

quiring encryption and decryption of data respectively. If such a CQE occurs, it records the request details into the shared object (2) and then triggers barrier1 (3). This indicates that $N+1$ threads have reached barrier1, which means that every thread unblocks from the barrier. The working threads inspect the shared object to recognize what job they had to perform (4), while the main thread waits on the second barrier (barrier2) for their work completion. Each worker executes its job repetitively as shown in Figure 3.26 (5). Upon completion, working threads "hit" barrier2 (6). When the last worker reaches barrier2, all threads wake up and continue. The main thread, now knowing that the requested job is completed, continues with its tasks (7), while the working threads loop back to barrier1 awaiting the next job (7).

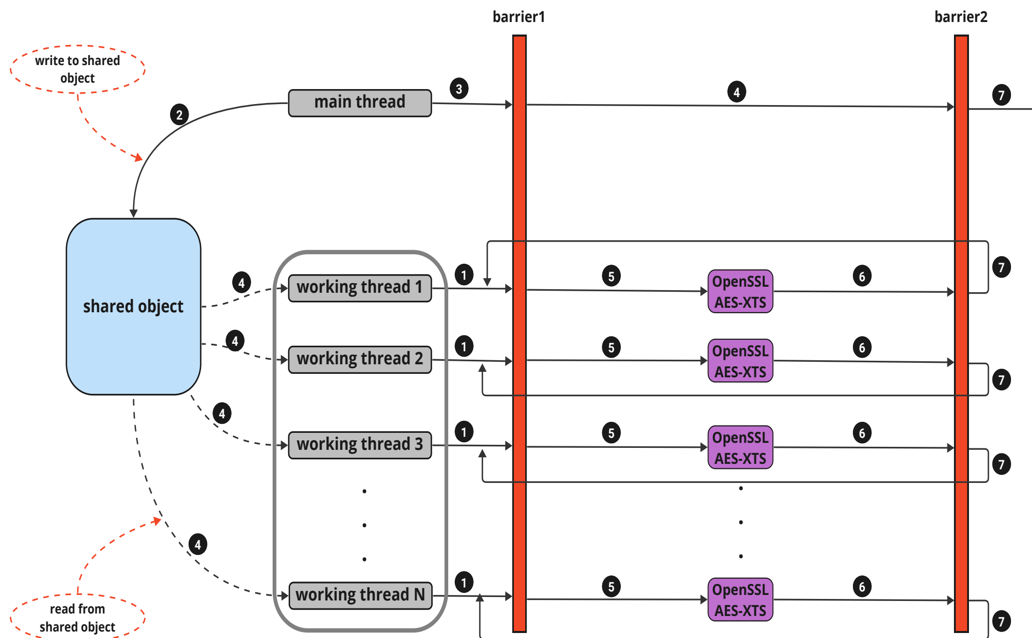

Figure 3.27: *Synchronization Between Main and Working Threads*

The primary objective during this design phase was to minimize the time spent on synchronization, avoiding unnecessary delays induced by waiting for locks. As illustrated in Figure 3.27, the working threads and the main thread do not wake up concurrently. The main thread activates the working threads at barrier1 and immediately awaits them at barrier2. Correspondingly, when the working threads awaken the main thread at barrier2, they loop back to wait at barrier1, while the main thread progresses. This ensures that there's no simultaneous manipulation of the shared object. If concurrent usage of the shared object between the main and working threads was allowed, a locking scheme would have been necessary, as the main thread writes to it and the working threads read

from it, which potentially would have reduced the efficiency of our solution. We should note, that the concurent access of the workers in the shared object doesn't need any synchronization because they are read-only entities.

Lastly, we take care of the elimination of the thread pool in a similar manner. Upon receiving a stop request, the main thread updates a specific field in the shared object, triggers barrier1 to wake up all threads, and invokes `pthread_join()` to join them. The working threads, upon waking and before they start to "work", they always check this specific variable. If set, it means that they need to stop, so they exit themselves.

3.5.5 Inter-Block Encryption

Both of the previous two implementations, namely **single-thread** and **intra-block parallelism**, had an issue: they altered the fundamental nature of the ublk server. This means that they modified the type of work the ublk server was initially designed to handle. Originally, the ublk server was architected as an **I/O bound** framework. The daemon queue threads comprising the ublk server submit requests to their Submission Queue and await a response. Upon receiving the response, they prepare the next request, whether it's for the driver or the target, and submit it again to the Submission Queue.

Our single-thread design transformed the *semantics* of each ublk server's thread into being **CPU bound**. The thread itself handled the necessary encryption or decryption, requiring CPU resources to perform the mathematical computations and generate results. Even in the case of intra-block parallelism, where we distributed this workload to worker threads, the main thread still waited for the results, so it remained a CPU bound implementation.

This issue made us think, design, and implement a solution that would allow the entire encryption process to occur without shifting the ublk server's nature from I/O-bound to CPU-bound.

To achieve this goal, we needed to relieve the main thread of any cryptographic operations or the need to wait for their completion, as it was the case in the previous two design solutions respectively. However, encryption and decryption tasks still needed to be performed, leading us to rely on a pool of worker threads once again. Each ublk

server's thread has a dedicated pool of workers, similarly to the intra-block implementation. However, in this case, instead of dividing the buffer and distributing the work to the workers, the main thread **offloads the entire buffer** to one worker and continues. Remember that each buffer can store up to 1024 sectors. This means that the working thread will execute the encryption or decryption on this buffer sequentially on each 512-byte segment, according to its size each time, exactly as it was the case in the single-thread solution. But, unlike the single-thread solution where this task was executed by the main thread itself, now it is managed by a worker.

Communication between main thread and workers

One of the key questions in this design was how to perform the communication between the main thread and the working threads. After offloading a request, the main thread no longer waits but continues its work, which, as mentioned earlier, involves submitting SQEs and waiting for CQEs. So, how do the working threads signal the main thread that a request is completed? We needed to integrate this communication (from working threads to the main thread) into the io_uring. To do so, we decided to allocate one more entry during the initialization of each io_uring Submission Queue, and this entry would be used to poll for readiness an **eventfd**.

The eventfd system call returns a file descriptor that refers to an "eventfd object" that can be used as an event wait/notify mechanism by userspace applications, and by the kernel to notify userspace applications of events [mpb]. The eventfd can be thought of as an empty pipe. One process waits at one end of the pipe until another process writes something to the other end to unblock it. The file descriptor returned by the eventfd system call can be used by the main thread to receive notifications from the worker threads. Every working thread, after finishing with a request, just needs to perform a write on the eventfd.

However, the main thread cannot directly wait on the eventfd because it also awaits requests from the driver and the target. For this reason, we integrated the waiting on the eventfd within the io_uring. The server submits a SQE with a poll request (operation `IORING_OP_POLL_ADD`) on the eventfd.

This means that each time the main thread wakes up with a CQE in hand, this CQE could originate from one of five sources:

- The ublk driver sent a write request.
- The ublk driver sent a read request.
- The target responded to a read request.
- The target responded to a write request.
- A worker finished a cryptographic job (encryption or decryption) and wrote to the eventfd.

The first four sources remain unchanged from the previous implementations. However, we now have an additional case to handle in this design. What should the main thread do when it receives notification that a requested job is complete? We will examine this in the data path section below.

Now, let's consider communication from the main thread to a working thread. We implemented this "communication flow" through a **condition variable**. The working threads are waiting on this condition variable, and the main thread signals them whenever there is work to be done.

Figure 3.28 focuses on this communication between the main thread and the working threads. The ublk server receives either a write request from the driver, or a read response from the target (1), prepares a request and submits it to the working threads (2), then signals the condition variable (3). A thread wakes up and carries out the encryption or decryption (4), then writes to the eventfd to notify the main thread that the requested job is completed (5). The kernel sees the write to the eventfd, indicating that the eventfd is ready to be read, so it submits a CQE for the previously submitted SQE with the `IORING_OP_POLL_ADD` operation (6).

Request manipulation

We've discussed the communication between the working threads and the main thread, but we haven't describe how exactly the requests from the main thread to working threads are managed and vice versa. What is happening in Step 2 of Figure 3.28? In this section, we'll try to provide answers to these questions.

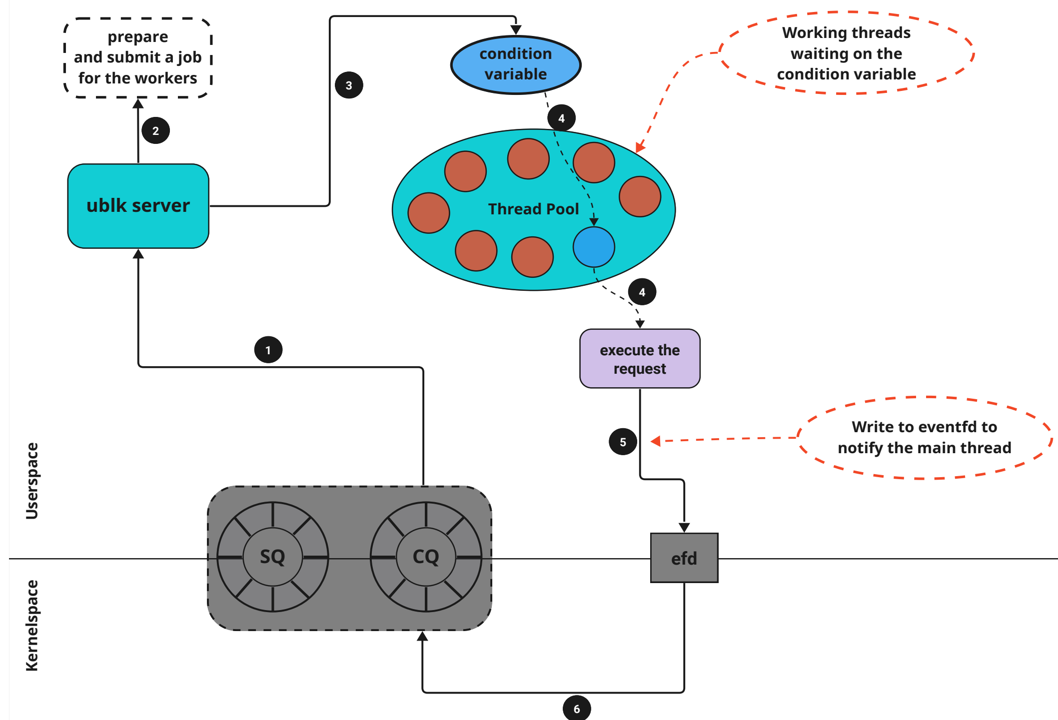

Figure 3.28: *Communication Between Main Thread and Working Threads*

As in the case of intra-block design, we have a shared object at the core of our design that helps communication between the main thread and the working threads. This object, in addition to the eventfd and the condition variable (whose roles we explained earlier), consists of three crucial fields: a "submit queue", a "complete queue", and an array of requests. Each request is a structure that represents a job and encapsulates all the information necessary for a worker to perform encryption or decryption.

Fig. 3.29 helps to clarify the relationships between these entities. Initially, we have queue-depth request structures preallocated during the initialization phase, for the same reason we have queue-depth "original" and "temporary" buffers. This is because each ublk server's thread can process a specific number of I/O requests on-the-fly, which is defined by the queue depth, and cannot handle more than that. Therefore, we have one request structure for every possible outstanding I/O request, establishing a 1-1 connection between the "original buffers", "temporary buffers", and the "request structures". The Nth request structure handles a cryptographic operation from or to the Nth "original buffer" and "temporary buffer".

The logic behind preallocating all the structures is the same as it was for the "temporary buffers" preallocation, which we analyzed in Section 3.5.3: to avoid invoking new system

calls for allocation during the hot path while serving each I/O operation. In the "time vs space" trade-off, we have chosen time whenever possible in our design.

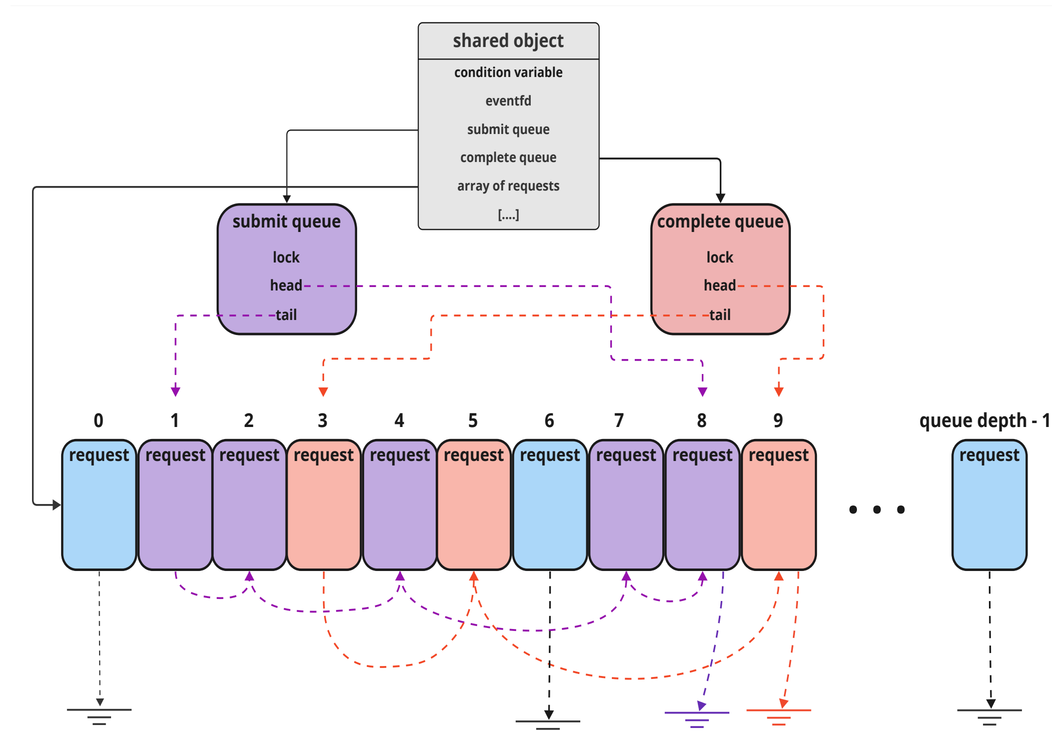

Figure 3.29: *Connections Between Important Structs*

Every time the ublk server needs to perform a cryptographic operation, it prepares the "request structure" based on the **tag** of the specific operation. As discussed in 3.4.3, each CQE sent by the driver to the server refers to a specific buffer using a value provided by the block layer, which is known as **tag**. This tag is the connecting factor that associates the "request structure" with the "original" and "temporary buffers".

We keep track of the submitted requests using a linked list called the "submit queue". The main thread adds requests at the head of the list, while the working threads extract requests from the tail of this list.

Similarly, there is a second list named "complete queue". This list is used by the working threads to submit their responses, and by the main thread to read the results. For these responses, we use the same "request structures" that we used for submissions.

It's important to note that these lists do not allocate or deallocate any request structure. We simply create a "chain" of requests either for the "submit queue"or for the "complete queue". This setup can be clearly seen in Figure 3.29, where the requests 1, 2, 4, 7, and

8 are in the "submit queue", and the requests 3, 5, and 9 are in the "complete queue". All other requests are not used at this moment.

We also need to consider synchronization among these components and protect against parallel access using locks in the "submit queue" and the "complete queue". We will explain how we managed to do this in the implementation chapter.

Data Path in Inter-Block Parallelism

Now let's try to fit all the pieces together and follow an I/O request to see how all components work together. We'll present both a write and a read request as sent by an application.

Let's begin by tracing a write request as illustrated in Figure 3.30. Up until Step 3, the process remains the same as discussed in Fig. 3.24. Now, after the server wakes up and checks the CQE in its hands (3) and realizes it has a write request, this indicates that it needs to encrypt the data before sending them to the disk. Thus, based on the tag number, it prepares a "request structure" which, among other information necessary for the working threads, also points to the buffer where the working thread will find the plaintext (the "original buffer") and the buffer where it needs to write the ciphertext (the "temporary buffer") (4). After populating this structure, the server adds this request to the "submit queue". It then signals the working threads that there is work to do (5). A thread wakes up, removes the job from the "submit queue" and then calls the necessary OpenSSL library functions to perform the encryption (6). At the same time, the main thread continues its job, by submitting any available SQEs (6). When the working thread completes its task, it adds the request to the "complete queue", and writes to the eventfd to notify the main thread (7). The main thread wakes up, processes the CQE and identifies that it was a "signal" from a working thread. This signifies that there is a request ready for further processing. So it retrieves the requests from the "complete queue" and finds that it has a response from a working thread regarding a write request. This means that the main thread has to send the encrypted buffer to the disk for writing, so it submits an SQE to write the buffer (8). Beyond this point the steps are again the same as the path we followed in Fig. 3.24 and have been analyzed.

A read request follows the same logic, but changes the order in which things are executed. Figure 3.31 demonstrates the flow. Upon receiving a read request (2), the server

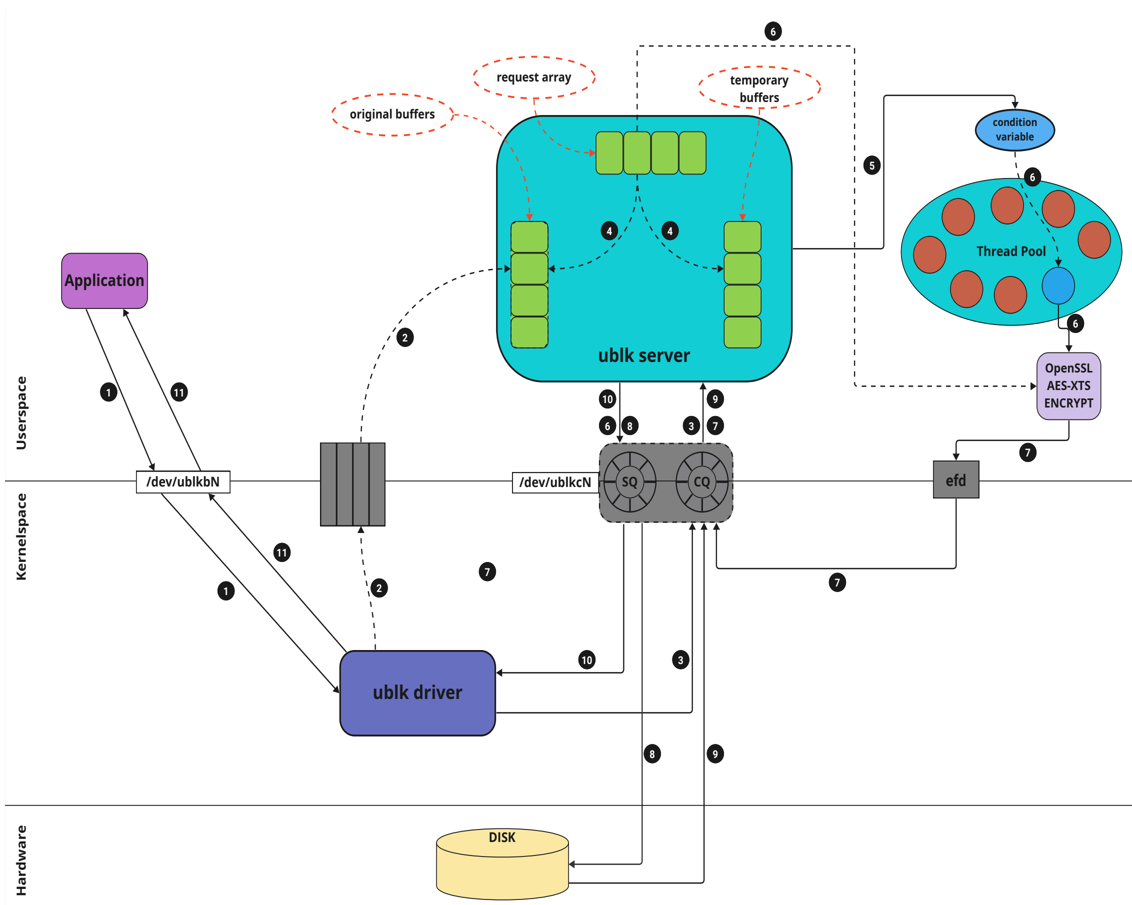

Figure 3.30: *Write Request from an Application in Inter-Block Encryption*

submits an SQE to the target to read the encrypted data into a "temporary buffer" (3). After the target replies (4), the server submits a "request structure" to the working threads to decrypt this buffer and place the plaintext in the original buffer (5, 6). A working thread carries out the decryption and "signals" the completion via the eventfd (7, 8). Lastly, the server notifies the driver as usual via a `UBLK_IO_COMMIT_AND_FETCH_REQ` (9), and the driver performs the necessary copies to the application's buffer, before completing the request to the block layer (10, 11).

This concludes the discussion on Inter-Block encryption. It is now clear how we managed to design and integrate encryption into the ublk framework, by offloading CPU-intensive tasks and keeping the server I/O bound.

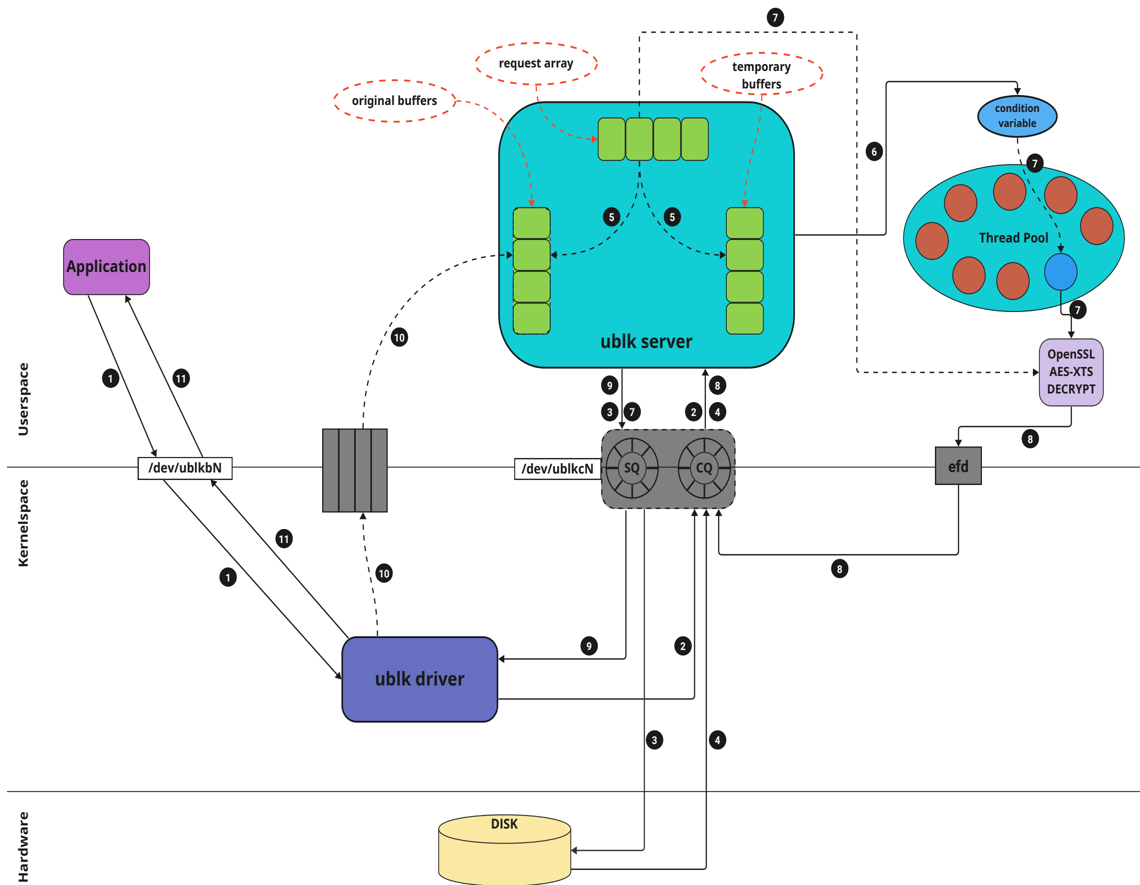

Figure 3.31: *Read Request from an Application in Inter-Block Encryption*

3.6 AES

Following the introduction to AES and the mathematical background we presented in Sections 2.5.3 and 2.5.4, we will now delve into its inner workings, examine how it functions, and consequently, comprehend why it is widely employed and how it eventually delivers the encryption we require.

3.6.1 Structure of AES

AES operates in rounds, depending on the size of the key provided, as indicated in Table 2.1. For input, this algorithm takes a 128-bit block. AES splits the input into 16-byte chunks, and these bytes form a 4x4 matrix known as the **state matrix**. All the operations in AES take place on this matrix.

Each round consists of 4 layers:

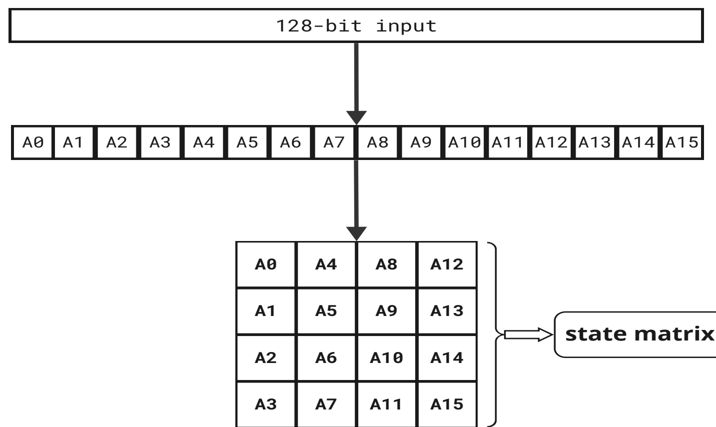

Figure 3.32: *AES Input*

1. Byte Substitution
2. Shift Row
3. Mix Columns
4. Key Addition

Byte Substitution

This layer is also known as the "**S-Box layer**". Each byte of the input, is passed in parallel into the same S-Box and produces another byte as output.

$$\boxed{S(A_i) = B_i}$$

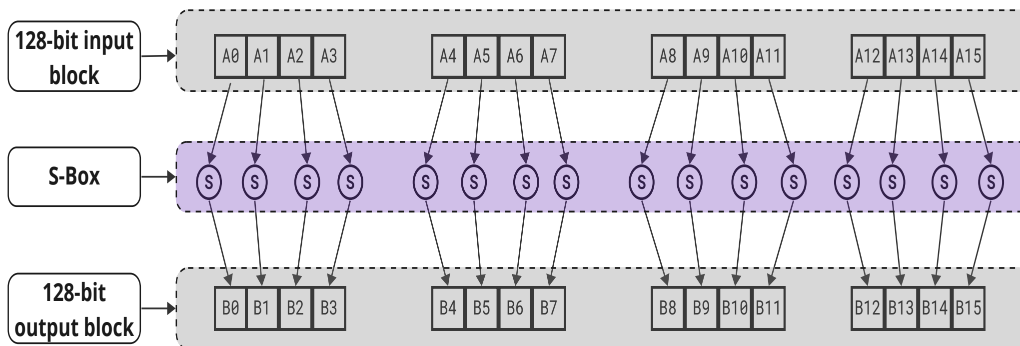

Figure 3.33: *Byte Substitution Layer*

An S-Box is essentially a function that takes as input a byte and produces another byte as output. Because both input and output are bytes, there are 256 different inputs, each of which is mapped to a unique output. This is why the AES S-Box can be represented as a lookup table (see Fig. 3.34).

	00	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0a**	**0b**	**0c**	**0d**	**0e**	**0f**
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3.34: *S-Box*

Example: Let's assume the input byte to the S-Box is $A_i = (B5)_{hex}$. Then the output value will be $S((B5)_{hex}) = (D5)_{hex}$.

On a bit level $(B5)_{hex} = 10110101$ and $(D5)_{hex} = 11010101$, so the substitution can be described as:

$$S((10110101)_b) = (11010101)_b$$

□

As explained by [CP10] «*The S-Box is the only nonlinear element of AES, i.e., it holds that* $ByteSub(A) + ByteSub(B) \neq ByteSub(A + B)$ *for two states A and B. The S-Box substitution is a bijective mapping, i.e., each of the* $2^8 = 256$ *possible input elements is one-to-one mapped to one output element. This allows us to uniquely reverse the S-Box, which is needed for decryption*".

Internals of the S-Box

The S-Box operates in two phases. In the first phase, it computes the inverse of the input byte, and in the second phase, each inverse byte is multiplied by a constant bit-matrix followed by the addition of a constant 8-bit vector.

The inverse of the input byte is done in the $GF(2^8)$ field (see more on 2.5.4). The irreducible polynomial used in AES, known as the "**AES irreducible polynomial**" is:

$$\boxed{P(x) = x^8 + x^4 + x^3 + x + 1}$$

The constant matrix is: $\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$

And the constant 8-bit vector is: $\begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$

Example: Let's examine how the S-Box matches $S((B5)_{hex})$ to $(D5)_{hex}$, as shown in the previous example.

Step 1: Compute the inverse of $(B5)_{hex}$ modulo $x^8 + x^4 + x^3 + x + 1$. This is explained in 2.5.4 and can be computed directly via a lookup table. It is $(B5)^{-1}_{hex} = (75)_{hex} = (01110101)_b$.

Step 2: Compute the multiplication of the 8x8 constant matrix, with the inverse $(75)_{hex}$. Note that it goes from the least significant bit to the most significant bit:

$$\begin{pmatrix} 1&0&0&0&1&1&1&1\\ 1&1&0&0&0&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&1&1&1&0&0&0&1\\ 1&1&1&1&1&0&0&0\\ 0&1&1&1&1&1&0&0\\ 0&0&1&1&1&1&1&0\\ 0&0&0&1&1&1&1&1 \end{pmatrix} \cdot \begin{pmatrix}1\\0\\1\\0\\1\\1\\1\\0\end{pmatrix} = \begin{pmatrix}4\\3\\3\\2\\3\\3\\4\\3\end{pmatrix} \equiv \begin{pmatrix}0\\1\\1\\0\\1\\1\\0\\1\end{pmatrix} \pmod 2$$

Step 3: Add the constant 8-bit vector: $\begin{pmatrix}0\\1\\1\\0\\1\\1\\0\\1\end{pmatrix} + \begin{pmatrix}1\\1\\0\\0\\0\\1\\1\\0\end{pmatrix} = \begin{pmatrix}1\\2\\1\\0\\1\\2\\1\\1\end{pmatrix} \equiv \begin{pmatrix}1\\0\\1\\0\\1\\0\\1\\1\end{pmatrix} \pmod 2$

And finally when we examine the result bottom-up we find that $(1101)_b = (D)_{hex}$ and $(0101)_b = (5)_{hex}$, which is what we expected.

This example demonstrates how the S-Box maps input bytes to output bytes.

□

Shift Rows

The state matrix after the "Byte Substitution" phase, looks like this:

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}

Table 3.2: *Input of Shift Rows Phase*

Where each $B_i = S(A_i)$.

In this phase, the bytes of each row in the state matrix shift cyclically to the left as follows: the first row remains unchanged, the second row shifts by one position to the left, the

third row shifts by two positions to the left, and the fourth row shifts by three positions to the left.

The output is the new state:

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}

Table 3.3: *Output of Shift Rows phase*

Mix Columns

In this step, each column of the state matrix, is multiplied by a predefined matrix. The predefined matrix is the following:

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

This computation results in another array, where each column of the new array consists of the multiplication of the predefined array with the corresponding column of the state matrix. For example, the second column of the new array is computed as:

$$\begin{pmatrix} C_4 \\ C_5 \\ C_6 \\ C_7 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} B_4 \\ B_9 \\ B_{14} \\ B_3 \end{pmatrix}$$

Every operation is performed in $GF(2^8)$ as described in 2.5.4. Therefore, the multiplication is modulo the AES polynomial $(x^8 + x^4 + x^3 + x + 1)$ and the addition is modulo 2, which is essentially an XOR operation.

For example, the calculation for C_4 is as follows: $\boxed{C_4 = 02 \cdot B4 \oplus 03 \cdot B_9 \oplus 01 \cdot B_{14} \oplus 01 \cdot B_3}$.

These two steps (Shift Rows and Mix Columns) add ***diffusion*** to the cipher. Shifting

diffuses the data horizontally, while mixing does so vertically. Diffusion is crucial operation in cryptographic algorithms that spreads the influence of one plaintext bit over many ciphertext bits, making it challenging to discern any patterns in the plaintext. In other words, if two plaintexts differ by just one bit, the corresponding ciphertexts must be completely unrelated. Both of them must look like random numbers.

Some intuition on diffusion in AES: If we flip one bit of the 128-bit plaintext, it will initially affect only one byte in the Byte Substitution step, resulting in approximately 4 bits being flipped compared to the original byte. In the Shift Rows step, this byte changes its position, but no other bytes are affected. However, in the Mix Columns step, this changed byte influences 4 bytes in the output, specifically the corresponding C_i column, which consists of 4 bytes, as we observed in the Mix Columns step. So, in just one round, a single-bit change in the plaintext can alter 32 bits in the output.

Key Addition

In this layer, the current state matrix is XORed (added in $GF(2)$) with a 128-bit subkey derived from the primary key. The number of subkeys equals the number of rounds plus one because there is an initial XOR operation with the input using the first subkey at the beginning of AES, before the 4-layer round we discussed. For instance, in AES with 256-bit keys and 14 rounds, there are 15 key derivations.

The derivation process involves passing through a nonlinear function for 32 bits of the key and then cyclically XORing each 32-bit part of the key with another 32-bit part. We won't delve further into the details of key derivation here, but anyone interested can refer to [CP10].

Decryption

The decryption process involves the inversion of the layers we discussed. Practically, each step we examined has an inverse, and the subkeys remain the same, but they are applied in the reverse order. So, the order of decryption in each round is as follows:

1. Key Addition
2. Inverse Mix Columns

3. Inverse Shift Row

4. Inverse Byte Substitution

3.6.2 Modes of Operation

The AES algorithm we presented encrypts chunks of 128 bits. However, in practical scenarios like secure communication, emails, and data storage, the information being transmitted is typically much larger than 128 bits. This raises the question of how we can encrypt long plaintexts using a block cipher.

The usual approach is to break the lengthy message into a series of sequentially fixed-sized blocks, and then apply encryption operations to each of these blocks. Essentially, a mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block [Wikc].

Many modes of operations have been defined. In this section we will cover the following:

- Electronic Codebook Mode (ECB)
- Cipher Block Chaining Mode (CBC)
- Cipher Feedback Mode (CFB)
- XEX Tweakable Block Ciphertext Stealing (XTS)

We'll pay special attention to the last one, the XTS mode of operation, as it was the mode we employed for implementing third-party encryption on ublk.

Note: The block operations are not tied to a specific block cipher. They simply describe how you can encrypt a lengthy message using a block cipher of your choosing. While we're using the AES block cipher, they are conceptually separate entities.

Electronic Codebook Mode (ECB)

ECB is the most straightforward way of encrypting a long message. If the message exceeds the block cipher's size, it's divided into the required fixed-sized blocks, and each

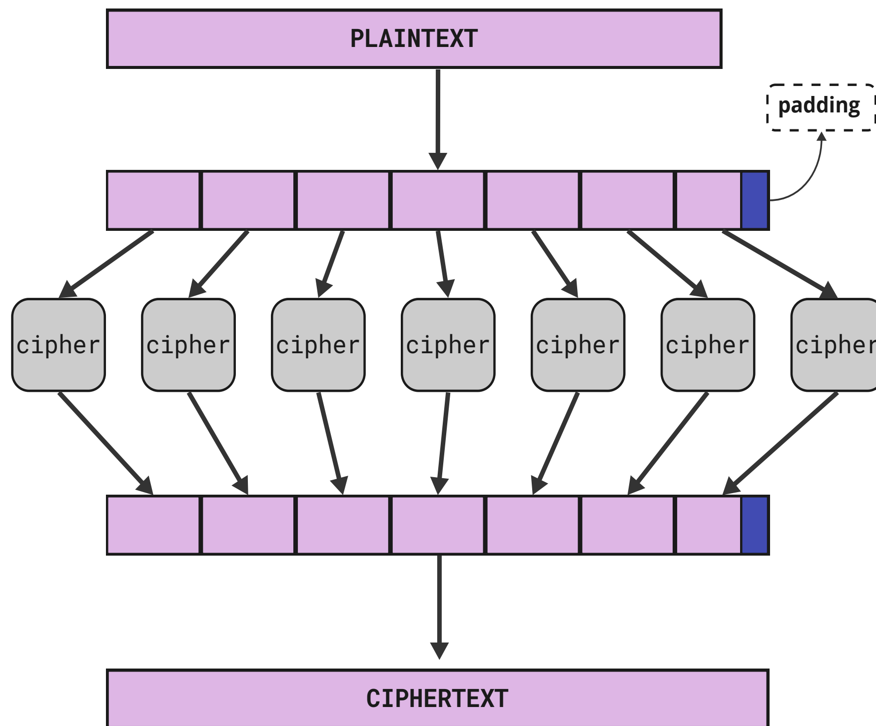

Figure 3.35: *ECB Mode*

block is encrypted independently. If the message length isn't a multiple of the block size, it must be padded before encryption. Figure 3.35 provides an illustration.

The decryption process follows similar principles. The ciphertext is split into fixed-sized blocks, and the decryption process is applied to each individual block.

This mode has several advantages: (a) It is parallelizable, allowing for the encryption and decryption of each chunk of code in parallel without requiring knowledge of anything else. (b) Bit errors only affect the specific fixed-size chunk they belong to and don't propagate. (c) There is no need for sender-receiver synchronization. The receiver can start the decryption process prior to receiving all of the encrypted blocks.

However, the disadvantages outweigh the advantages. The main drawback is that it doesn't conceal data patterns well. Identical plaintext blocks will always lead to identical ciphertext blocks, as long as the cipher key remains unchanged. Fig. 3.36 illustrates this issue. It reveals information about the underlying message. By examining the publicly accessible ciphertexts, a third party can identify identical blocks and potentially make educated guesses about the original plaintext.

We can differentiate between two types of encryption schemes:

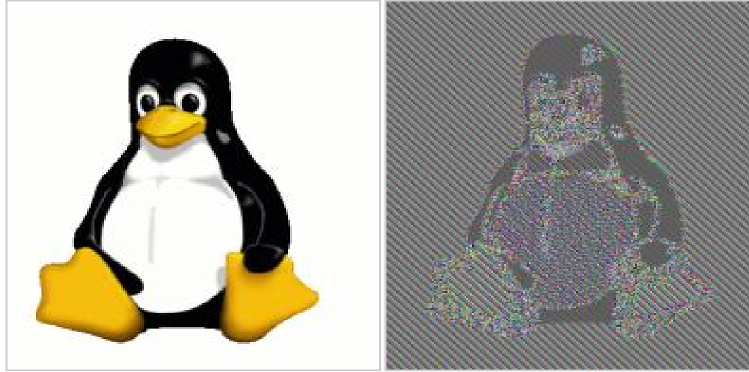

Figure 3.36: *Tux Encrypted Using AES in ECB Mode*

1. **Deterministic encryption:** This scheme always maps a particular plaintext to the same ciphertext if the key doesn't change.
2. **Probabilistic encryption:** This scheme uses randomness to achieve non-deterministic generation of ciphertext.

The problem with the ECB mode is that it is deterministic. The goal in general is to use probabilistic encryption. And this type of encryption is what CBC and CFB mode provide.

Cipher Block Chaining Mode (CBC)

The **Initialization Vector**, or more simply IV, plays the crucial role of the randomization factor in encryption. It is essentially a random number used in conjunction with the encryption key. Unlike the key in symmetric encryption, the IV doesn't need to be kept secret. However, it must be used only once during the encryption process. This unique characteristic gives rise to another term for IV: "*nonce*", which stands for "number used once".

IV serves as the randomizer, because if we encrypt a plaintext once with a first IV and a second time with a different IV, the two resulting ciphertexts look completely unrelated to each other for an attacker.

The CBC mode, both the encryption in the upper half, and the decryption in the lower half, is illustrated in Figure 3.37.

The IV is XORed with the first plaintext block (p_1) and the result is passed from the encryption cipher to produce the first ciphertext block (c_1), and then each plaintext

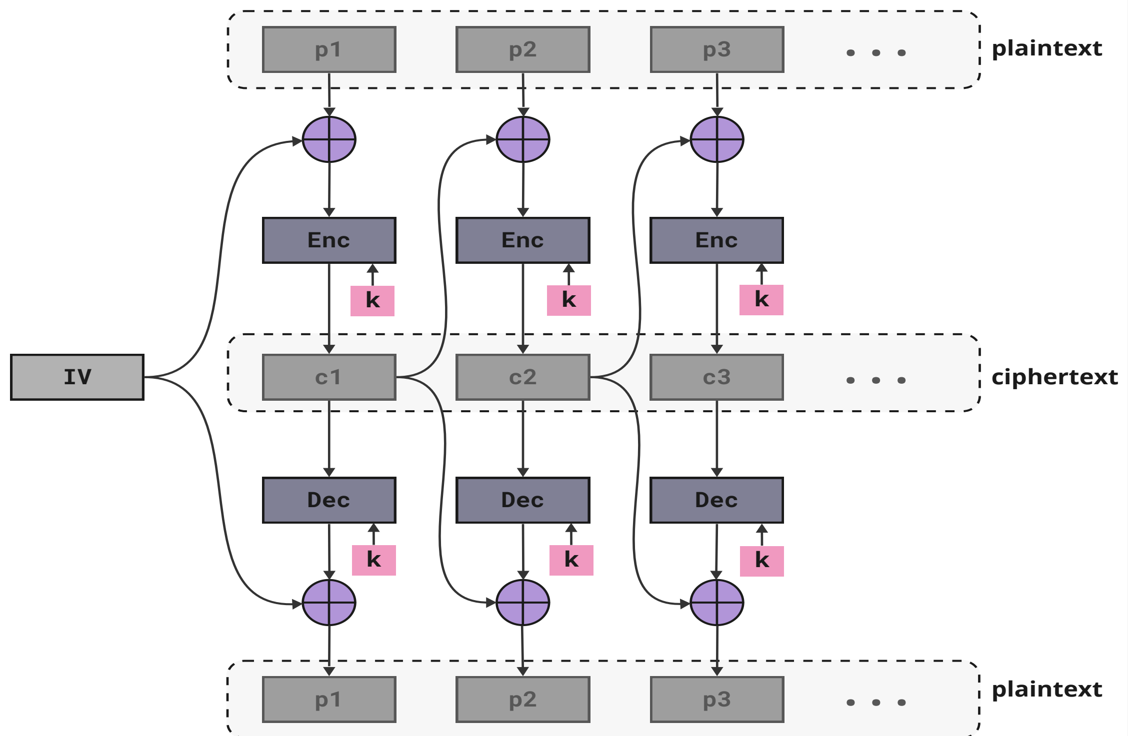

Figure 3.37: *CBC Mode*

block is XORed with the previous ciphertext block. Note that the first ciphertext c_1 depends on plaintext p_1 and the IV. The second ciphertext depends on the IV, p_1 and p_2. The third ciphertext c_3 depends on the IV, p_1, p_2, p_3, and so on. The last ciphertext block depends on all plaintext blocks and the IV.

This property makes CBC mode suitable for authenticating the original message, as even a small change in the plaintext will result in a different, unpredictable final block.

However, one disadvantage of this mode is that it cannot encrypt in parallel like ECB, because to compute a ciphertext block, you need to have computed the previous one first. This introduces a form of serialization in the ciphertext block computation. Nevertheless, it can decrypt in parallel, as only two ciphertext blocks are needed. Another drawback of CBC is that altering bits in the ciphertext block c_i results in corresponding alterations in the plaintext block p_{i+1} at exactly the same positions.

Cipher Feedback Mode (CFB)

The principle behind CFB is as follows: initially, we encrypt the IV in order to generate a key that will be XORed with the first plaintext block. For all subsequent “key blocks”,

the previous ciphertext is encrypted. This process is illustrated in Figure 3.38.

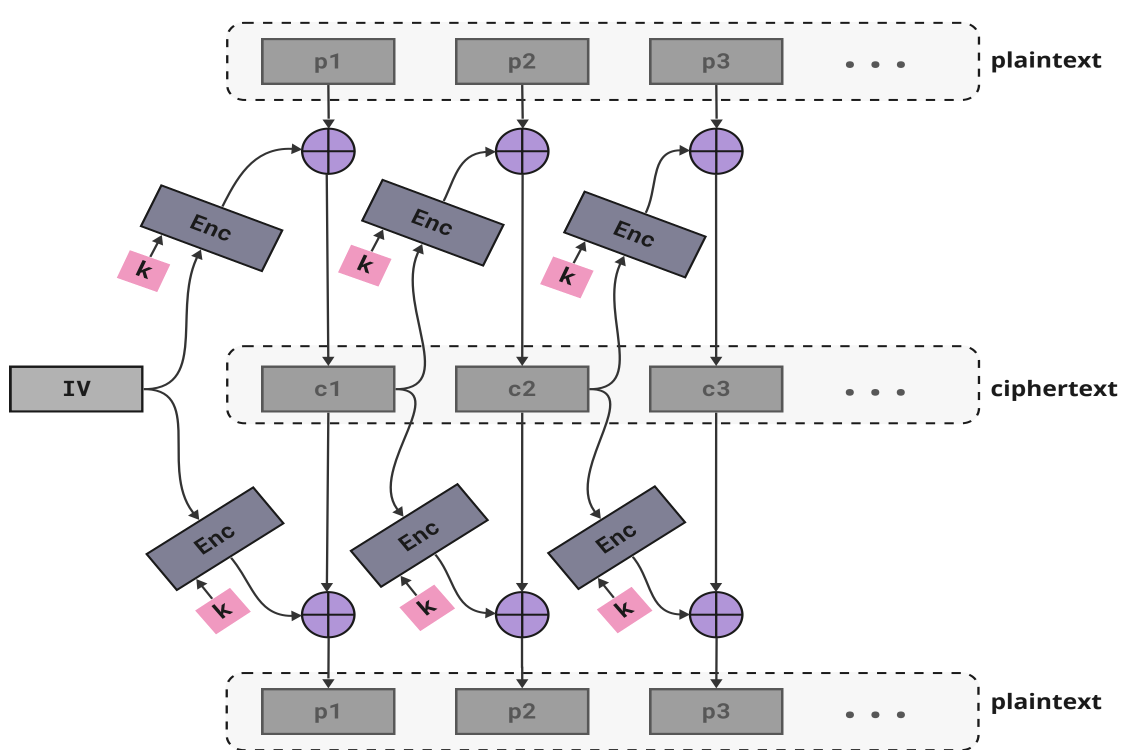

Figure 3.38: *CFB Mode*

The operations for encryption and decryption are identical. Like CBC mode, CFB also has parallel decryption, but serial encryption because the production of a ciphertext block depends on the previous one. It also shares the same issue as CBC regarding altering specific positions of the plaintext but with a slight difference. Altering bits in the ciphertext c_i affects specific positions in the corresponding plaintext p_i. In CBC mode, changing c_i impacts the corresponding positions in the subsequent plaintext block p_{i+1}.

3.6.3 XEX Tweakable Block Ciphertext Stealing (XTS)

The term "data-at-rest" refers to data that is stored on a storage medium, typically on a computer's or server's disk. Other forms of data, such as "data-in-transit" and "data-in-use", describe data that is in motion or loaded into memory, respectively [CLO].

AES-XTS is the default algorithm used for protecting data-at-rest on various storage mediums today. Many industry-standard disk encryption utilities, like VeraCrypt, BitLocker, and LUKS, employ AES-XTS as the default mode for data protection [Vera, Mic, Red].

Note: XTS mode is specifically designed for protecting data on storage devices and not intended for other purposes, such as encrypting data-in-transit.

Disk encryption methods are designed to achieve three key objectives:

1. Ensure confidentiality
2. Maintain fast data retrieval
3. Minimize disk wastage

The AES-XTS mode incorporates the concept of a **tweak**. Similar to the Initialization Vector discussed in Section 3.6.2, a tweakable cipher takes an additional argument, the tweak, alongside the key and input. The tweak introduces variability to the encryption scheme and, like the IV, does not need to remain secret.

XTS internals

XTS mode, like any other block cipher using AES as the basic encryption algorithm, operates on 128-bit blocks. Since XTS is used for encrypting disks, it also operates on sectors. This results in two basic units when referring to XTS: the 128-bit chunk, which serves as the working unit for each AES encryption (referred to as a *block*), and the sector, the minimum unit a storage device can handle (referred to as a *sector*). Thus, for the typical case we have 512-byte sectors they can be divided into 32 128-bit blocks. Understanding this division is important for comprehending how AES-XTS operates.

Let's examine how the encryption of an individual *block* is performed under XTS. Fig. 3.39 illustrates this operation.

AES-XTS employs a key with twice the size of AES encryption. This means for 128-bit AES, a 256-bit key is needed, and for 256-bit AES, a 512-bit key is required. XTS divides the key into two equal *subkeys*, referred to as key1 and key2, as shown in Figure 3.40.

The tweak plays the role of plaintext in the first encryption, using key2 as the key. The result is then multiplied by a^j, where a is an irreducible polynomial in $GF(2^{128})$ (specifically $x^{128} + x^7 + x^2 + x + 1$), and j represents the block's position within the sector (ranging from 0 to 31 for 512-byte sectors). The multiplication and computation occur in $GF(2^{128})$ as explained in Section 2.5.4. The result of this multiplication is then XORed with the plaintext block to be encrypted, followed by an AES operation. The

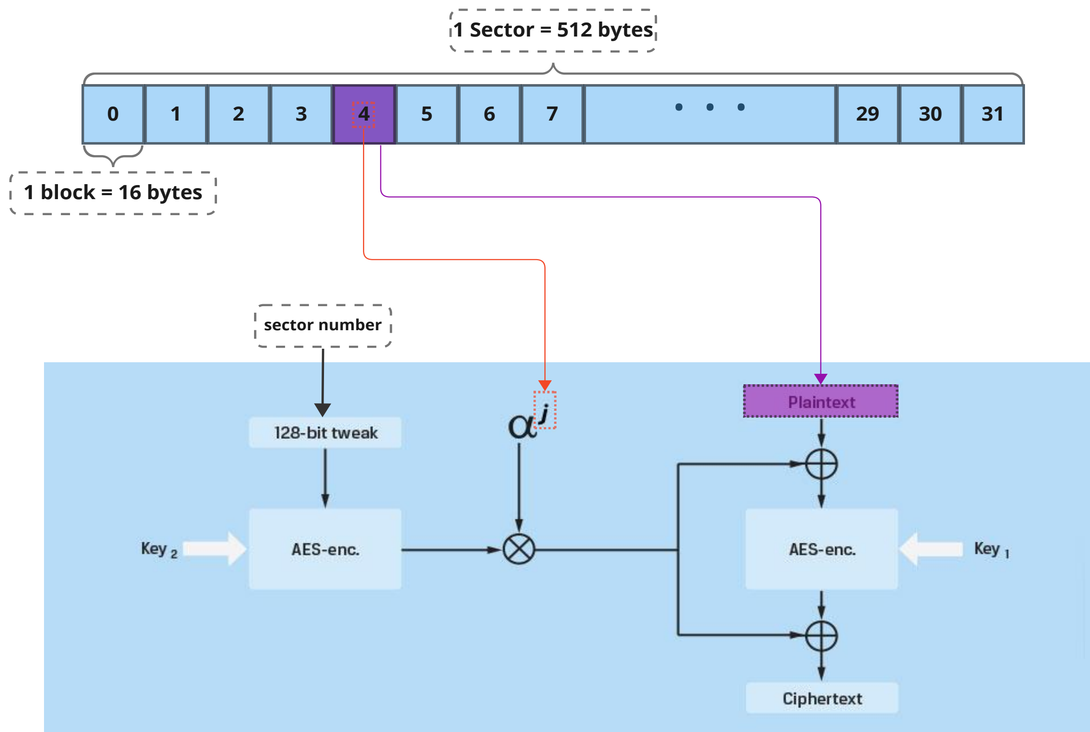

Figure 3.39: *Block Encryption in AES-XTS*

final ciphertext is obtained by XORing the result of this AES operation with the earlier multiplication result.

The tweak, is equal to the *sector number*. The sector number is a logical identifier denoting which sector is within the storage medium. Sectors start with 0, 1, and so forth, until reaching the storage medium's size divided by the sector size. Thus, the tweak is equal to the sector number: 0 for sector 0, 1 for sector 1, and so on.

Figure 3.39 describes the encryption of **one block** within a sector. Figure 3.41 illustrates the AES-XTS encryption within a whole sector. Notably, the tweak encryption occurs only once in a sector. Each time, the result of the encryption is multiplied in $GF(2^{128})$ by a^j, where j signifies the block's position within the sector. Consequently, for a 512-byte sector, the AES encryption algorithm must be executed 33 times: 32 times for each block within the sector and once for the tweak value.

Decryption follows a similar process, with the only difference being that each ciphertext block is passed through an AES decryption algorithm after XOR with the encrypted tweak value. The decryption procedure for a sector is depicted in Figure 3.42.

Note: The tweak value, is fed into the **AES encryption** algorithm again, just like in the

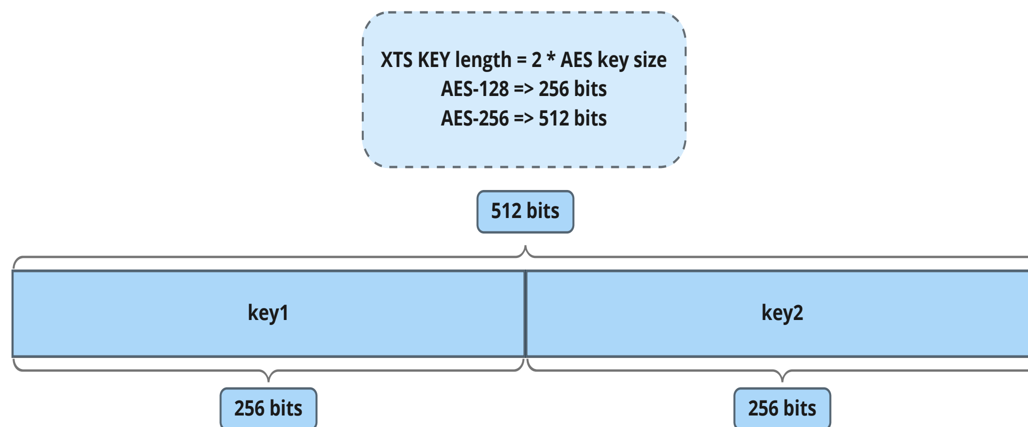

Figure 3.40: *Key Sizes in AES-XTS*

encryption procedure.

Ciphertext stealing

The last letter of the XTS acronym, stands for *ciphertext Stealing*. This is a feature that enables XTS to encrypt/decrypt sectors, whose size is not a multiple of the block size. For instance, what if the size of each sector was 520 bytes? This is not an XTS specific feature [Wikd].

Figure 3.43 showcases the ciphertext stealing technique by displaying the last two blocks of a sector in a scenario where the block size is not a multiple of the sector size. The last block consists of M bits, where $0 < M < 128$.

The concept is as follows: Suppose a sector that is divided into $k + 1$ blocks, with the first k blocks being 128 bits in size and the last one being M bits. All blocks except the last two are encrypted normally, as shown in Figure 3.43. The last two blocks are encrypted differently: the penultimate block is encrypted as usual, and its ciphertext is split into two parts. The first part consists of M bits and becomes the ciphertext of the last block, while the second part, comprising $128 - M$ bits, is used to pad the last M bits of plaintext. With this addition, the last plaintext block becomes 128 bits in size and is encrypted as usual. The resulting ciphertext block becomes the penultimate ciphertext block of the sector.

This concludes our discussion on the XTS mode. In summary, AES-XTS effectively achieves the three initial objectives we outlined for data-at-rest encryption, in the beginning of this section. It offers confidentiality, it doesn't add any overhead to access and process data, and due to ciphertext stealing the plaintext size and the ciphertext size

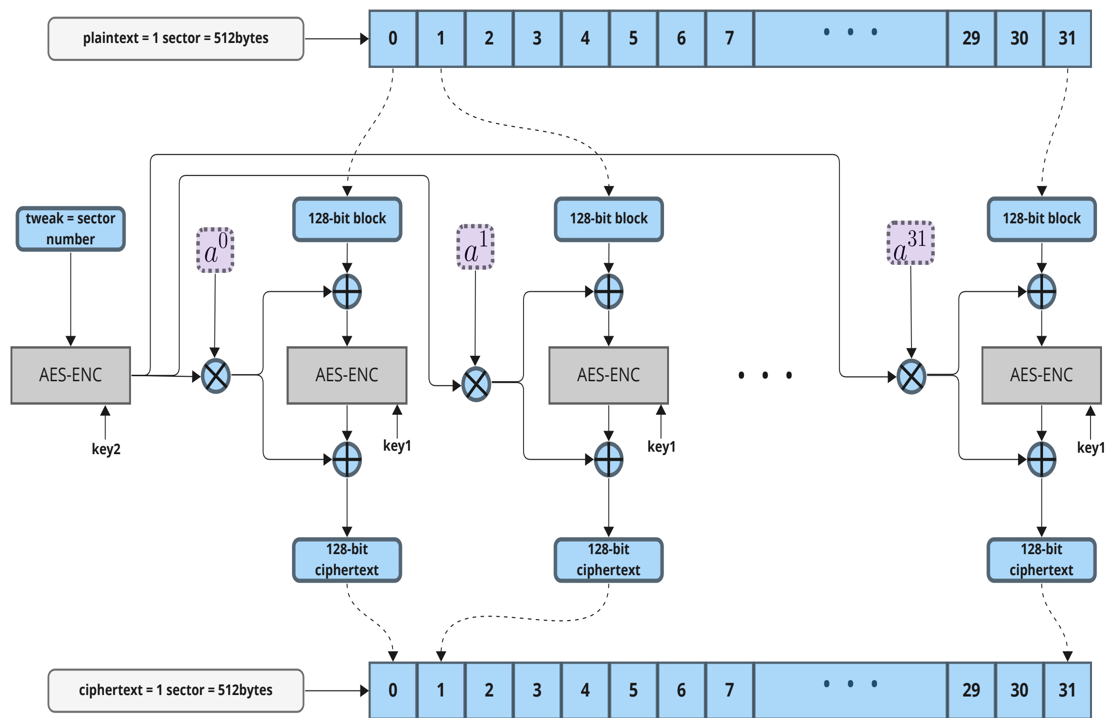

Figure 3.41: *Sector Encryption in AES-XTS*

are always equal. Also as it can be shown, both in encryption and in decryption the XTS mode can operate in parallel, for different sectors. However, it does not provide *data authentication*. This means that an adversary could potentially alter the data without detection. Achieving data authentication typically requires the inclusion of additional information, which may contradict the third objective. Disk encryption solutions usually aim to minimize the use of extra metadata space.

3.7 Linux Unified Key Setup (LUKS)

At this point, it will be useful to compare our key setup implementation with a standard system used in both professional and personal computing environments: **LUKS**, which is the acronym for Linux Unified Key Setup. This will allow us to assess what we may be missing and how we can improve our implementation in the near future.

What is LUKS?

LUKS is an encryption specification for block devices. It was developed by Clemens Fruhwirth, who released the version 1.0 in March 2005 [Fru]. Previous attempts at inte-

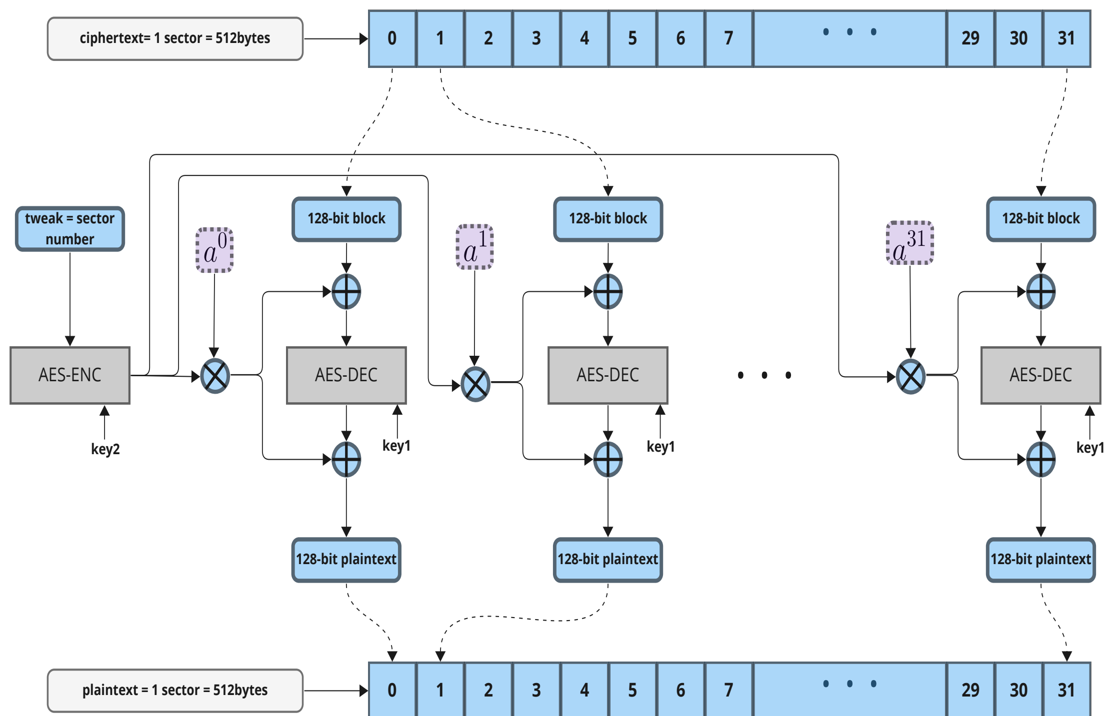

Figure 3.42: *Sector Decryption in AES-XTS*

grating cryptography into the Linux kernel faced challenges due to different approaches to key processing and parameter settings. The situation was resolved with the development of LUKS, which introduced a unified tool, resolving compatibility problems and providing a standardized approach to secure key setup for disk encryption in Linux. In a nutshell, LUKS defines how a block device is encrypted, which keys are used, where there are stored etc, along with various metadata information.

Today, the standard followed by most systems is an evolution of LUKS, known as LUKS2 [Bro]. We will analyze the initial implementation of LUKS, referred to as LUKS1, as the core concepts remain the same, and it provides a clearer understanding of the main ideas.

3.7.1 Key hierarchy

LUKS employs a concept known as ***key hierarchies***. To grasp the concept, consider the following scenario: If we were to encrypt a disk with a single key, changing this key would require re-encrypting the entire storage medium. This process can be particularly time-consuming, especially for larger disks. The idea behind "key hierarchies" is to treat

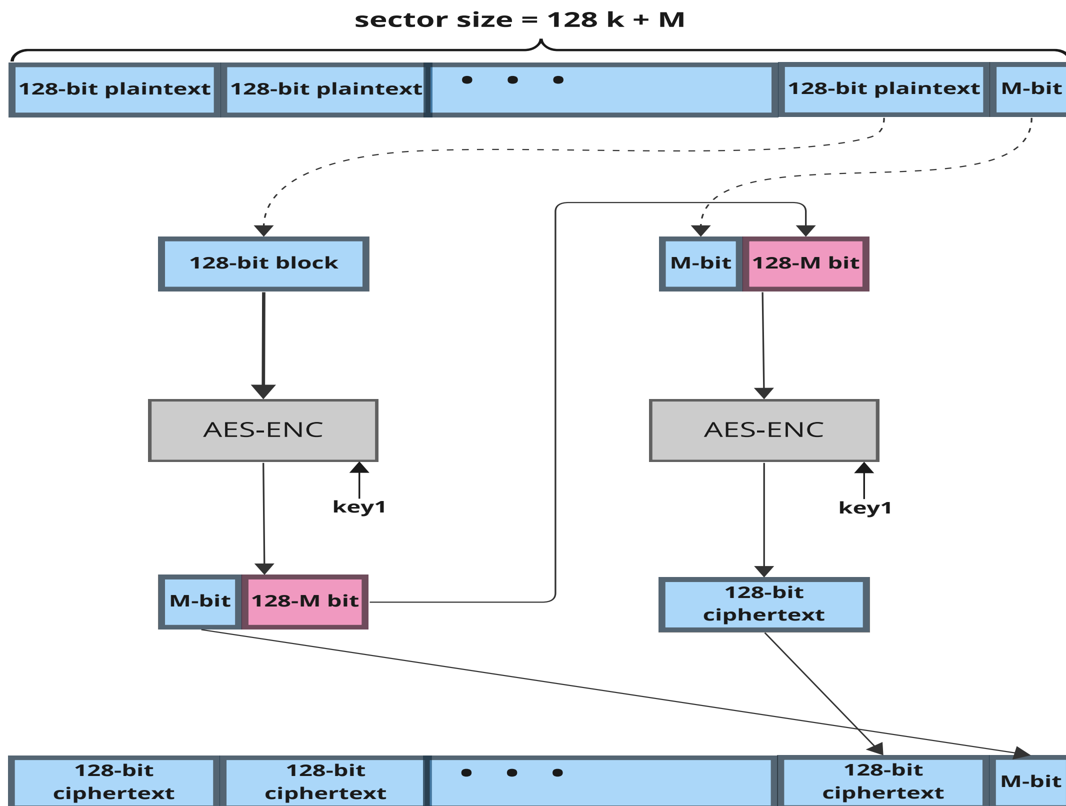

Figure 3.43: *Ciphertext Stealing Encryption*

the primary key as data. It is encrypted and stored as such. Consequently, a second key is necessary in order to decrypt the first.

We will refer to the initial key, which is responsible for encrypting the data, as ***data key*** or ***master key*** and to second key that is used to decrypt the initial key, as ***user key***.

This two-layer approach makes the procedure of changing user keys a lot easier. If a user wishes to change the key, the user decrypts the master key with the old user key, then re-encrypts the master key with the new user key, storing it in the same location as the previous one. As a result, the data encrypted by the master key remain unchanged. Only the encryption of the master key itself is modified.

Key hierarchies is a concept that is widely used in many cryptographic operations. For example Secure Shell protocol (SSH), uses key hierarchies [Phe]. A session key for a regular block cipher is generated on the fly and sent via public key encryption to another party. Upon successful decryption of the session key by the other party, it can then be used in a "symmetric way" by both parties for further communication. Generally, the practice of using a temporary key (often asymmetric) to encrypt a second data key (often

symmetric) for actual data transmission is a common technique in cryptography.

Key hierarchies also provide a way to allow more than one user keys to refer to the same master key. We just need to store the master key encrypted via each different user key.

Before we dive into the internal design of LUKS and examine how it works, it would be helpful to familiarize ourselves with two concepts that LUKS employs: (a) Secret splitting and (b) Key Derivation Functions.

3.7.2 Secret Splitting

This technique tries to split a secret message (e.g. a key) into equal parts. From these parts, we can reconstruct the secret key. However, if even one of these parts is altered the reconstruction will fail. This technique is commonly used to ensure the permanent erasure of sensitive information. Disks have a long memory and data can be recovered even if they seem to be gone.

By distributing crucial information into equally important parts, where even if one is altered the information can't be retrieved, we increase the probability that data erosion is permanent.

An easy to implement method for secret splitting is as follows: Imagine we have a secret message M of length K. We want to split M into N parts. We split the secret message by producing S_1, S_2, ..., S_{N-1} random messages of size K. Then we compute the last message as $S_N = S_1 \oplus S_2 \oplus S_3 \oplus ... \oplus S_{N-1} \oplus M$.

From the XOR semantics, this means:

$$\boxed{M = \bigoplus_{i=1}^{i=N} S_i} \tag{3.1}$$

Now, every S_i, carries important information for M . If an S_i changes, we will not be able to reconstruct M.

Usually, a more sophisticated concept is used for secret splitting. We typically want to insert some kind of diffusion in the process. With the process we described, each alteration on a specific position on an S_i will lead to a change in the corresponding

position on M . So, usually, during the computation of the S_i some kind of a hashing takes places to establish diffusion. But the concept remains the same.

The process of the splitting phase, from the original message M to the parts $S_1, S_2, ...S_N$ is called **AF splitting** (Anti-Forensic splitting). Conversely, the process from the $S_1, S_2, ..., S_N$ to the M as shown in equation 3.1, is called **AF merging**.

3.7.3 Key Derivation Functions (KDF)

People authenticate themselves on the Internet using passwords. Everyone who has used the Internet has come to a point where they needed to provide a password, whether it's for securing their accounts, accessing sensitive information, or conducting various online transactions. The biggest problem with this, is that people tend to use short passphrases that are easy to remember. In a mathematical sense, they provide passwords that lack *entropy*. This creates various problems because an adversary that can query the password can easily produce dictionary-like words and gain access to protected information.

Key Derivation Functions are functions that take a password as input, often along with additional data such as a *salt*, and generate a result that can be used as a cryptographic key for other operations. Typically, the computations performed by a KDF to derive the final key are **CPU intensive**. As a result, they do not complete quickly, demanding a significant amount of CPU cycles to produce the final result. The amount of time required is a factor that can often be adjusted. This characteristic makes KDFs particularly well-suited for deriving the user key protected by a passphrase, significantly increasing the level of difficulty for **brute force attacks**.

LUKS uses the *Password-Based Key Derivation Function, revision 2* (PBKDF2) as a key derivation function. PBKDF2 is based on a pseudo random function that is iterated many times, in order to consume a lot of CPU cycles for the final calculation. PBKDF2 takes not only the passphrase as input, but also the pseudo random function that computes the intermediate results, a salt, the desired number of iterations and the desired key length. However, we don't need to know the internals of PBKDF2. We can think of it like a black box that takes a password as input and instead of producing a result quickly, consumes a lot of CPU cycles until it produces the final key.

3.7.4 LUKS internal structure

The format of a disk that follows the LUKS standard, is shown in the Figure 3.44.

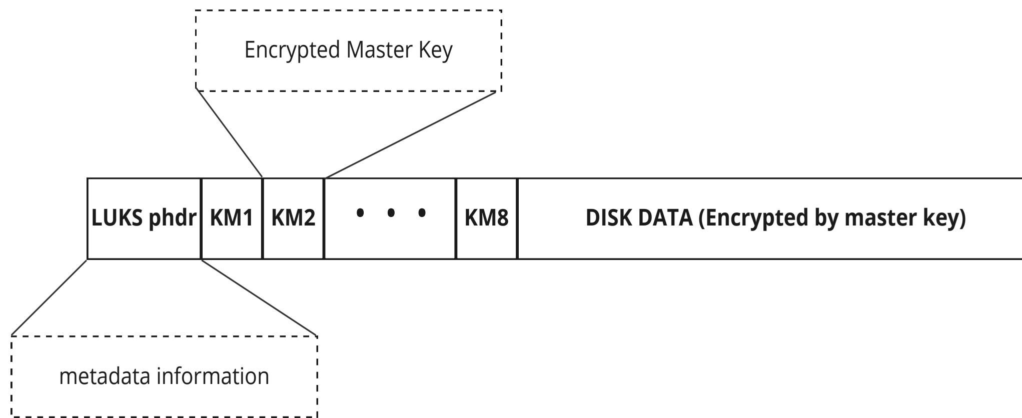

Figure 3.44: *LUKS Layout*

A LUKS partition starts with a partition header (the "LUKS phdr" in Fig. 3.44) that stores various metadata information for the partition. This includes the magic number that identifies it as a LUKS partition, the cipher name and cipher mode for cryptography, a digest of the master key, and more. At the end of the phdr, there are 8 slots, each referring to a KM field (Key Material). **Key material** refers to the actual bytes of the key used in a cryptographic operation. This distinction is important because sometimes "key" refers to both the actual key material and metadata about the key. Each of the eight KM regions stores the master key encrypted, and there are corresponding metadata slots in the phdr for each of these regions. Figure 3.45 provides a closer look at some of the LUKS header fields and visualizes the relationship between key-slots and key material.

A question that may arise is, "why do we need eight regions to save the same master key"? The answer lies in Section 3.7.1 where we discussed key hierarchies. Each of these regions saves the same master key encrypted by a ***different user key***. This means a user can choose as many passwords as there are key slots, allowing LUKS to operate with up to eight different user keys.

3.7.5 LUKS semantics

According to [Fru], LUKS can respond to 4 essential high-level commands:

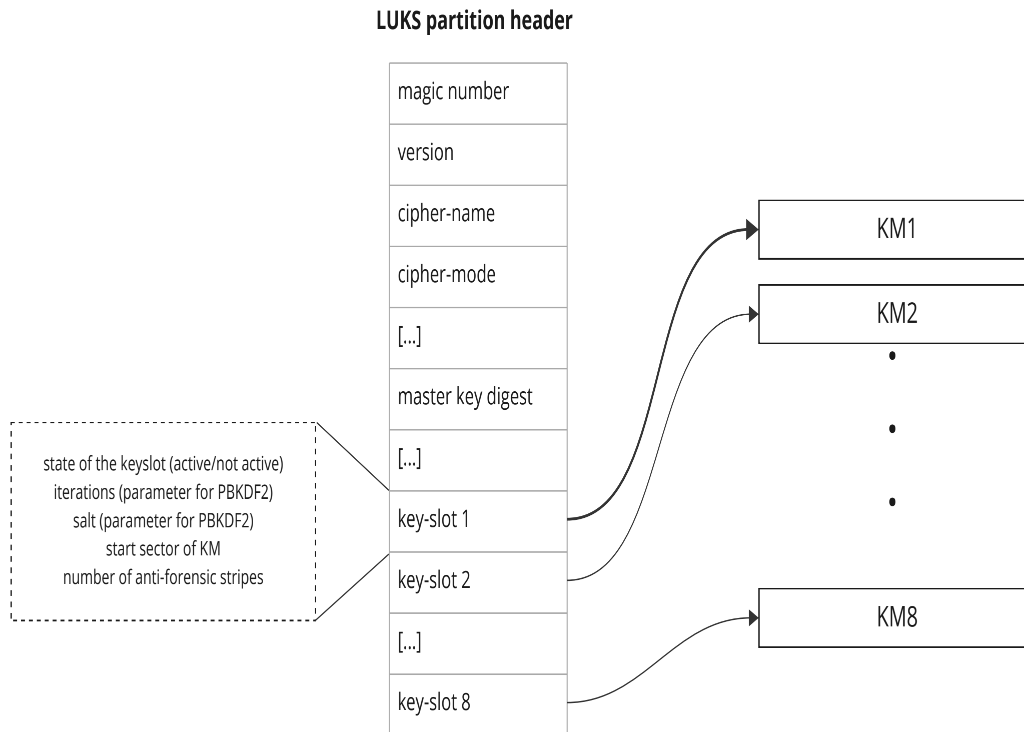

Figure 3.45: *A Zoom in LUKS Header*

1. **create partition:** Initialise an empty partition with a new master key, and set an initial passphrase.

2. **open partition:** Recover the master key with the help of a user supplied passphrase, and install a new virtual mapping for the backing device.

3. **add key:** Add a new passphrase to a key slot. A valid passphrase has to be supplied for this command.

4. **revoke key:** Disable an active passphrase.

Let's examine how these high-level commands are executed, and how the structures we presented fit into this process.

We will present two of the above operation, the "create partition" and the "open partition". The other two follow the same logic.

Create partition

To create a new LUKS partition, a series of steps are taken. Let's break down the most significant ones to enhance our understanding of the previously discussed topics. In

Figure 3.46, we present the initialization of a LUKS partition and the addition of a new key. While the figure may not capture all the details, it provides a clear representation of the core concepts. Let's dissect it.

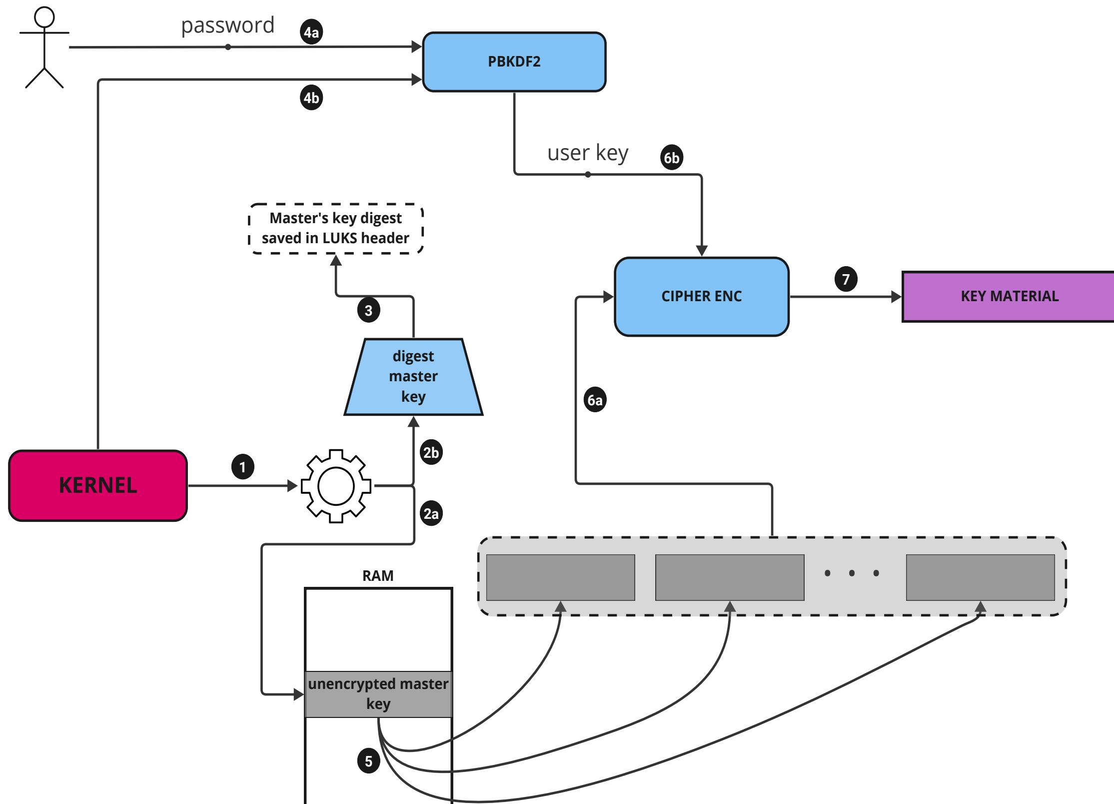

Figure 3.46: *Operation: Create Partition*

Initially, the kernel generates a random master key that will be employed later to encrypt the data (1). This key is temporarily stored unencrypted in memory (2a) and also digested in the "master key digest" field of the LUKS partition table (2b, 3). This digest is crucial for subsequent validation. With this step, the LUKS partition is initialized. The key activation process follows.

A user provides a password (4a), and other parameters for PBKDF2 are determined by the kernel. Subsequently, key derivation occurs, as detailed in Section 3.7.3. This yields a "user key" that will be used to encrypt the master key. In step 5, the AF splitting takes place, wherein the master key is divided into segments. This procedure is explained in Section 3.7.2. The encryption of the segmented key then occurs, using the user key derived from the user's password as the encryption key (6a, 6b). Finally, after encryption, the encrypted segmented master key is saved in one of the eight key material areas (7). The kernel then erases the memory where the master key was temporarily stored.

It's noteworthy that the master key is never saved in an unencrypted form on disk. Additionally, as we've observed, the master key isn't encrypted directly, instead a segmented master key is stored in the key material area.

Open partition

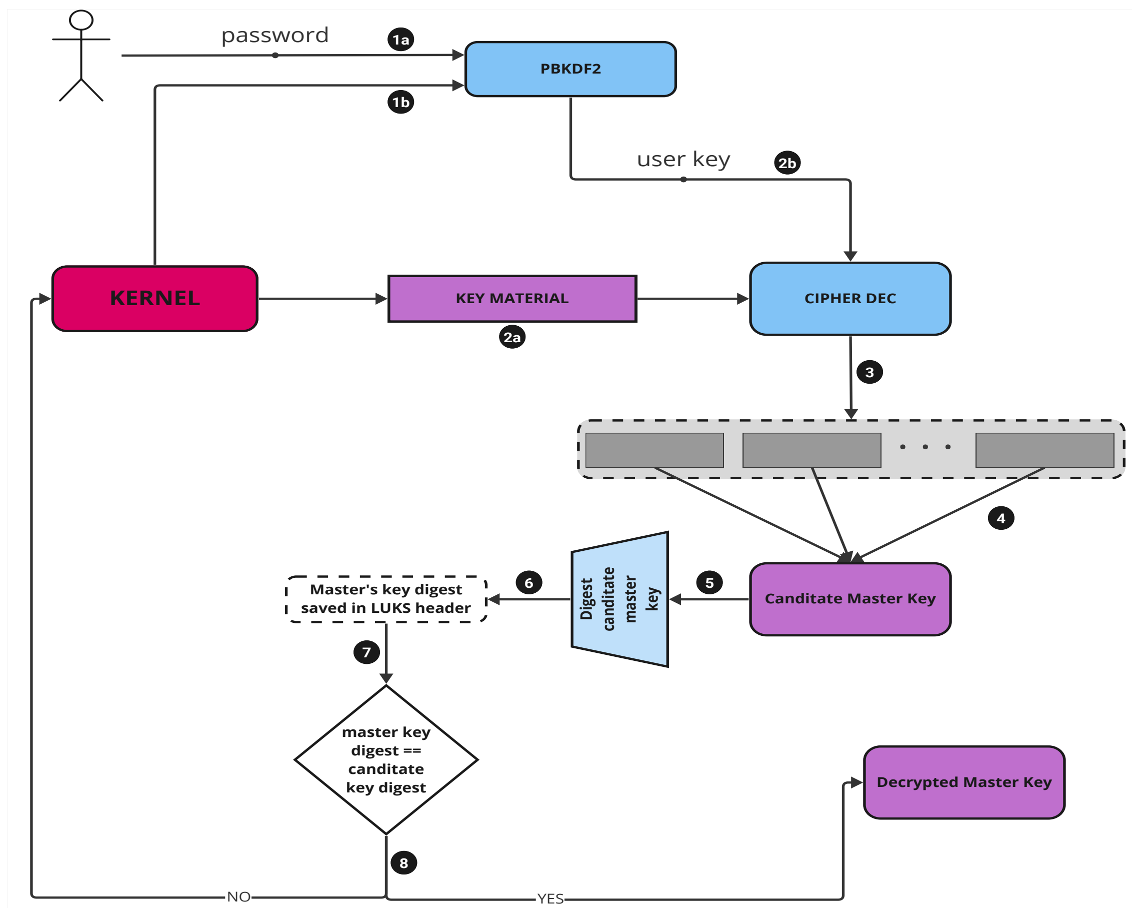

Figure 3.47: *Operation: Open Partition*

The procedure of recovering the master key, and decrypt the data is shown in Fig. 3.47. A user key, derived from a PBKDF2 s used to decrypt the master key. But the master key is AF-split stored in key material. Thus after decrypting it we must AF-merge it (4) to retrieve a *possible* master key. This candidate key then is passed from a digestion process (the same as in 2b of Fig. 3.46) and the result is checked against the saved master key digestion which lives in the LUKS phdr (6). If those two digests are equal, the validation process is passed successfully, and we have a master key decrypted, that can be used to decrypt/encrypt the whole disk. If this process fails, the kernel checks if there are other active key-slots and repeats the steps 2-8 for every active key-slot. If it achieves to match

one of them, then the master key is decrypted successfully and it is ready for usage. If it fails for every active key-slot, then the user is not validated and can't encrypt/decrypt the disk.

4 Implementation

4.1 Overview

In this chapter, we present our implementation of the three proposed solutions for integrating encryption within the ublk framework. Additionally, we describe the key setup phase, which is common across all implementations.

The ublk server is written in C and C++. Our implementation is compatible with both languages, though it adheres to the conventions of the C programming language to ensure compatibility and ease of integration with the existing codebase.

Throughout the upcoming sections, we'll highlight the key aspects of our solutions, discuss challenges encountered while translating our design choices into code, and reference the corresponding design decisions and diagrams to establish a stronger connection between implementation details and the chosen design.

Note: For brevity, we will focus on the key points, omitting some finer details, **and will avoid showcasing error handling** in most cases, to keep the presented code easy to follow and less noisy.

Libraries Used

One advantage of integrating the encryption mechanism directly into userspace is the ability to use ready-made libraries. Especially for encryption purposes, this is very important because implementing cryptographic operations from scratch is difficult and error-prone. These libraries are, of course, open-source and they have been tested and

integrated into numerous systems. This fact provides us with assurance (at least to some extent) that we won't introduce a security-related bug in our implementation.

We used two libraries: **GnuPG Made Easy (GPGME)** and **OpenSSL** [GNUa, Ope]. Both are C libraries and have been used for many years for various cryptographic tasks, in both industry and personal computing environments. These libraries are popular and have played a key role in making data transfer and storage secure in many applications and systems.

GPGME

GPGME is a library that doesn't implement by itself the cryptographic protocols and operations. It acts as a high-level interface, making it easier for developers to integrate cryptographic functions into their applications, and uses "backends" to do the real work. The default backend used by GPGME, and the one we also used in our operations, is the **GnuPG**, which according to [Gnub] is «*an universal crypto engine which can be used directly from a command line prompt, from shell scripts, or from other programs. Therefore GnuPG is often used as the actual **crypto backend** of other applications*».

Figure 4.1: *Application Using GPGME*

GPGME employs inter-process communication (pipes) to exchange data between the application and the backend. However, the specifics of the communication protocol and how the backend is accessed are completely hidden by the interface. Simply put, GPGME takes away all the complexity, allowing the application to interact with the backend effortlessly, without getting bogged down by the underlying technical details.

OpenSSL

OpenSSL is a full-featured open-source toolkit that implements the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. It also includes a cryptographic library that provides a vast array of cryptographic algorithms and functions, such as symmetric and asymmetric encryption, hashing, digital signatures, and key generation and management.

Unlike GPGME, which serves as a higher-level interface to backend cryptographic engines, OpenSSL itself is a cryptographic library that directly implements all the necessary cryptographic algorithms and protocols.

By using these libraries, we were able to take advantage of well-tested cryptographic solutions, saving a lot of development time. We avoided the challenges that come with building cryptographic algorithms and protocols from the ground up. This way, we could concentrate on adapting the encryption mechanism to work smoothly with the ublk framework, making sure our implementation is both safe and efficient.

4.2 Key Setup Implementation

An encrypted ublk disk begins with a metadata region that saves information about the current encryption (if any) on the disk.

The struct that holds the metadata information has the following form:

```
#define UBLK_MAGIC_LEN 6
#define UBLK_HEADER_VERSION_LEN 3
#define UBLK_MASTER_KEY_HASH_LEN 64
struct ublk_header {
  char magic[UBLK_MAGIC_LEN];
  char version[UBLK_HEADER_VERSION_LEN];
  int active;
  unsigned char master_key_hash[UBLK_MASTER_KEY_HASH_LEN];
};
```

Listing 4.1: *struct ublk_header*

The whole metadata region allocated is 4096 bytes, but the `ublk_header` structure takes up the first 80 bytes (see Fig. 3.20).

The basic functionality of this structure is to store the hash of the master key. When creating the encrypted ublk disk for the first time, we store the hash of the master key in the `master_key_hash` field. This enables us to validate users later on, who want to utilize the ublk framework.

Also, the `active` field along with the `magic`, inform if the current disk has already been encrypted or not.

Now, let's see how we implemented some of the high-level operations as presented in Section 3.5.2.

Create an Encrypted Disk Implementation

The following function, `start_enc()`, executes the Algorithm 1. This function is called during the creation of the encrypted ublk (when calling 3.10) before the ublk server starts, and takes the following three parameters:

- A pointer to a structure (`struct ublksrv_dev_data`), which encompasses information necessary to start the ublk server. The important field for us at this moment is a structure of type `encryption`, that stores an array of `unsigned char`, that will hold the master key during the usage of the server.
- The file descriptor of the backing file, which serves as the disk.
- The file descriptor of the file, where we will eventually store the encrypted master key.

```
int start_enc(struct ublksrv_dev_data *data, int fd_disk, int fd_master_key)
{
  gpgme_ctx_t ctx;
  gpgme_data_t plain, cipher;
  int fd_random;
  unsigned char md_buf[UBLK_MASTER_KEY_HASH_LEN];
  struct ublk_header header = {0};

  // Check that this is not an active disk
  pread(fd_disk, &header, sizeof(struct ublk_header), 0);
  if (header.active == 1) {
    if (!strncmp(header.magic, "UBLKEN", UBLK_MAGIC_LEN)) {
      fprintf(stderr, "This disk is already encrypted. ABORT!\n");
      return -1;
    }
  }
  ftruncate(fd_master_key, 0);

  // Allocate a structure to hold the master key
  data->enc = (struct encryption *)calloc(1, sizeof(struct encryption));

  // Read the master key from a random source
  fd_random = open("/dev/random", O_RDONLY);
  read(fd_random, data->enc->key, KEY_SIZE);
```

```
25
26      // Initialize a GPGME context
27      init_gpgme(GPGME_PROTOCOL_OpenPGP);
28      gpgme_new(&ctx);
29
30      gpgme_set_ctx_flag(ctx, "no-symkey-cache", "1");  // Disable caching of GPG daemon
31      gpgme_set_armor(ctx, 1);
32
33      // Save the encrypted master key to file fd_master_key
34      gpgme_data_new_from_mem(&plain, (const char *)data->enc->key, KEY_SIZE, 0);
35      gpgme_data_new_from_fd(&cipher, fd_master_key);
36      gpgme_op_encrypt(ctx, NULL, GPGME_ENCRYPT_SYMMETRIC, plain, cipher);
37
38      // Compute and store the hash of the key
39      SHA512(data->enc->key, UBLK_MASTER_KEY_HASH_LEN, md_buf);
40
41      memset(&header, 0, sizeof(struct ublk_header));
42      memcpy(header.magic, "UBLKEN", UBLK_MAGIC_LEN);
43      memcpy(header.version, "0.1", UBLK_HEADER_VERSION_LEN);
44      header.active = 1;
45      memcpy(header.master_key_hash, md_buf, UBLK_MASTER_KEY_HASH_LEN);
46
47      pwrite(fd_disk, &header, sizeof(struct ublk_header), 0);
48
49      return 1;
50  }
```

Listing 4.2: *Function start_enc()*

We can divide the functionality of `start_enc()` function into the following sections:

1. Checks whether the disk is already encrypted. If it is, it aborts.

2. Reads from `/dev/random` the master key. This key will be saved in a data structure (`struct encryption`) that we will be used later on to perform the "real" encryption and decryption. The `KEY_SIZE` is defined as 64 bytes (512 bits) because when encrypting in XTS mode, the key size required is twice the size of the standard AES key.

3. Starts a cryptographic context. The actual cryptographic operations are always set within a context in GPGME. A context provides configuration parameters that define the behaviour of all operations performed within it.

4. Encrypts the master key and saves the encrypted key in the file indicated by the `fd_master_key` argument.

5. Digests the master key using the Secure Hash Algorithm (SHA512).

6. Populates the `ublk_header` structure and saves it at the start of the disk for later usage. The information stored in this structure remains unchanged in subsequent interactions with the disk.

Every function prefixed with "`gpgme_*`" is actually a library call to GPGME. The encryption of the master key and the save to the file is performed via the `gpgme_op_encrypt()`. We used symmetric encryption to protect the file that is going to hold the master key, and by default, the GnuPG backend uses the AES algorithm. It also asks for a passphrase that will protect the encrypted master key from now on.

The `SHA512()` function that performs the digest of the master key, is defined in the OpenSSL library.

From this point onwards, the metadata section of the disk is populated, and the master key is stored in the `key` field of the encryption structure and from there it can be used for any cryptographic operation.

Open an Encrypted Disk Implementation

The function below is used when starting the ublk server with a pre-encrypted disk, and it is invoked as a consequence of starting the ublk server as mentioned in 3.11. Its main task is to recover the master key from the file passed as an argument. This implementation corresponds to Algorithm 2 shown in Section 3.5.2.

Just like `start_enc()`, the `extract_key()` function takes three arguments, with a slight difference. Here, the file descriptor referred to as `fd_cand_key` refers to a file containing a "candidate key", which needs validation before granting access.

```
1 int extract_key(struct ublksrv_dev_data *data, int fd_disk, int fd_cand_key)
2 {
3   gpgme_ctx_t ctx;
4   gpgme_error_t err;
5   gpgme_data_t plain, cipher;
6   size_t len;
7   unsigned char md_buf[UBLK_MASTER_KEY_HASH_LEN];
```

```
8     unsigned char *buf;
9     struct ublk_header header = {0};
10
11    pread(fd_disk, &header, sizeof(struct ublk_header), 0);
12
13    if (strncmp(header.magic, "UBLKEN", UBLK_MAGIC_LEN)) {
14      fprintf(stderr, "header.magic is wrong. ABORT!\n");
15      return -1;
16    }
17    if (header.active != 1) {
18      fprintf(stderr, "Disk must be active. ABORT!\n");
19      return -1;
20    }
21
22    data->enc = (struct encryption *)calloc(1, sizeof(struct encryption));
23
24    init_gpgme(GPGME_PROTOCOL_OpenPGP);
25    gpgme_new(&ctx);
26    gpgme_set_ctx_flag(ctx, "no-symkey-cache", "1"); // Disable caching of GPG daemon
27
28    gpgme_data_new_from_fd(&cipher, fd_cand_key);
29    gpgme_data_new(&plain);
30    gpgme_op_decrypt(ctx, cipher, plain);
31
32    buf = (unsigned char *)gpgme_data_release_and_get_mem(plain, &len);
33
34    SHA512(buf, UBLK_MASTER_KEY_HASH_LEN, md_buf);
35
36    if (strncmp((const char *)header.master_key_hash, (const char *)md_buf,
        UBLK_MASTER_KEY_HASH_LEN)) {
37      // Validation failed
38      fprintf(stderr, "The provided key was not correct\n");
39      return -1;
40    } else {
41      // Validation passed
42      memcpy (data->enc->key, buf, KEY_SIZE);
43    }
44
45    return 1;
46  }
```

Listing 4.3: *Function extract_key()*

The primary tasks of the `extract_key()` are listed below:

1. Retrieves the metadata section from the disk.

2. Checks that this is indeed a valid disk.

3. Allocates the `encryption` structure, which will house the master key upon successful validation.

4. Initializes a cryptographic context, essential for making any GPGME-related function calls.

5. Decrypts the context of the file containing the candidate master key.

6. Computes the hash of the candidate master key.

7. Compares the hash of the candidate master key with the stored master key hash in the `master_key_hash` field of the `ublk_header` structure. A mismatch in hashes denies access, whereas a match confirms the provided file contains the correct master key. The master key is then copied to the `enc->key` field, making it ready for subsequent cryptographic operations.

The next high-level operation of adding a new key, as presented in Section 3.5.2, follows the logic of the previous two, and thus, will not be detailed here.

Lastly, as discussed in 3.5.2, to remove a key we simply delete the file containing the encrypted master key. This action does not require any additional steps.

4.3 Single-Thread Encryption Implementation

In our initial solution, we mainly made three modifications to the existing ublk server code:

1. We initialized an OpenSSL cryptographic context inside of which every cryptographic operation is performed.

2. We allocated the "temporary buffers" and utilized them whenever interacting with the target. For a write request, the encrypted data resides in these "temporary buffers" and are sent to the target for storage. In contrast, for a read request, we retrieve the encrypted data from the target and place them in the "temporary buffers" before decryption.

3. We executed the appropriate calls to the OpenSSL library functions for performing the encryption or decryption tasks.

Initialize a Cryptographic Context

Every cryptographic operation must take place within a specific "cryptographic context" in OpenSSL. This context is represented by a structure of type `EVP_CIPHER_CTX`. Additionally, there is a separate structure for the specific cipher method implementation, denoted as `EVP_CIPHER`.

We allocate these two structures during the initialization phase of every "queue-thread" in the ublk server and maintain a reference to them. This approach allows us to reuse them for every encryption or decryption operation, avoiding unnecessary allocations during hot path.

These two structures are encapsulated within a common structure named `crypt_ctx`, and we refer to this structure whenever we need to perform an operation within the OpenSSL context.

Let's see how this initialization is performed.

```c
struct crypt_ctx {
  EVP_CIPHER_CTX *ctx;
  EVP_CIPHER *cipher;
};

struct crypt_ctx *init_crypt_ctx(void)
{
  struct crypt_ctx *c_ctx = (struct crypt_ctx *)malloc(sizeof(*c_ctx));

  c_ctx->ctx = NULL;
  c_ctx->cipher = NULL;
  c_ctx->ctx = EVP_CIPHER_CTX_new();
  c_ctx->cipher = EVP_CIPHER_fetch(NULL, "AES-256-XTS", NULL);

  return c_ctx;
}
```

Listing 4.4: *Function init_crypt_ctx*

Every `EVP_*` object and function, refers to OpenSSL-related features. The `EVP_CIPHER_CTX` and `EVP_CIPHER` structures are initialized using `EVP_CIPHER_CTX_new()` and `EVP_CIPHER_fetch()` respectively. The former function allocates and returns a cipher context while the latter retrieves the cipher implementation for the specified *algorithm*. We requested AES-256 in XTS mode as the algorithm.

Write Request

The following function is triggered during the data path for a **write request**, specifically at Step 4, as shown in Figure 3.24.

```c
int encrypt(const struct ublksrv_queue *q, const struct ublksrv_io_desc *iod, int tag)
{
  struct crypt_ctx *c_ctx = q->private_data; // Cryptographic context
  void *buf = (void *)iod->addr; // Original buffer
  __u64 start_sector = iod->start_sector;
  __u32 nr_sectors = iod->nr_sectors;
  __u64 relative_bytes;
  __u32 relative_sector = 0;
  union iv xts = {0};
  int encrypt_size;
  const struct ublksrv_ctrl_dev *ctrl_dev = ublksrv_get_ctrl_dev(q->dev);
  struct encryption *enc = ublksrv_get_encryption(ctrl_dev); // Master key
  struct _ublksrv_queue *queue = tq_to_local(q);
  unsigned char *t_buf = queue->tmp_buf[tag]; // Temporary buffer

  if (!EVP_EncryptInit_ex2(c_ctx->ctx, c_ctx->cipher, enc->key, NULL, NULL))
    goto err;

  while (nr_sectors) {
    xts.sector = start_sector;
    relative_bytes = relative_sector << 9;

    if (!EVP_EncryptInit_ex2(c_ctx->ctx, NULL, NULL,
          (const unsigned char *)xts.tweak, NULL))
      goto err;
    if (!EVP_EncryptUpdate(c_ctx->ctx, t_buf + relative_bytes, &encrypt_size,
          (const unsigned char *)buf + relative_bytes, 512))
      goto err;

    start_sector++;
    relative_sector++;
    nr_sectors--;
  }
  [...]
}
```

Listing 4.5: *Function encrypt()*

Let's decompose this code. Firstly, the function takes three parameters:

- A pointer to a structure containing information for a running ublk server queue.

- A pointer to a descriptor of the request, which has been explained thoroughly in Section 3.4.3.
- The tag of the request.

With the information from the second argument (`ublksrv_io_desc`), we can obtain three crucial values: (a) The pointer to the "original buffer" (i.e. the buffer containing the plaintext we wish to encrypt), (b) the starting sector that the first 512-byte segment of the "original buffer" refers to (this value is significant for encryption as the sector number is used as the tweak value), and (c) the total number of sectors included in this request.

Based on the `tag`, we can identify the specific "temporary buffer", referred to as `t_buf`, that will store the result of the encryption. This information along with the `crypt_ctx` structure which holds the cryptographic context in which the encryption operation will be performed, is extracted from the first parameter of the function, which contains information about the current queue.

The important computation occurs within the `while(nr_sectors)` loop. This loop, based on the request size, processes each 512-byte segment from the "original buffer", and stores the encrypted version of each segment in the corresponding location within the "temporary buffer". Here is a breakdown of the loop:

1. The loop continues as long as there are sectors left to process, as indicated by `nr_sectors`.
2. In each iteration:
   (a) The current sector number is updated in the `xts` structure.
   (b) The byte position within the buffers is calculated based on the current sector being processed.
   (c) Encryption initialization is performed with the current sector's tweak value via `EVP_EncryptInit_ex2()` function. This function is called in two different places. The first time it sets up the cipher context `ctx` for encryption with the XTS cipher, and inside the `while` loop it sets the current tweak value according to the current sector.

(d) The encryption process takes a 512-byte segment from the original buffer, encrypts it, and places the result in the temporary buffer at the corresponding position. This is performed by the `EVP_EncryptUpdate()` function, which encrypts successive blocks of data in every call.

(e) Sector indices are updated for the next iteration, moving on to the next 512-byte segment.

3. The loop advances to the next sector and repeats the process until all sectors have been encrypted.

As we can see, the XTS is applied individually to 512-byte chunks that corresponds to disk sectors with the tweak being the sector number.

To set appropriately the tweak value, we store it in a union as follows:

```
union iv {
  __u64 sector;
  unsigned char tweak[TWEAK_SIZE];
};
```

Listing 4.6: *Union Used to Set the Tweak Value*

So, in each loop iteration, we assign the number of the sector being encrypted to the `sector` field, and then **translate** this value to an `unsigned char` to properly set the tweak value expected by `EVP_EncryptInit_ex2()`. This is a "trick" we used in all implementations.

Read Request

In a read request, decryption occurs after the data has been retrieved from the target and stored in the "temporary buffer" (Step 5 in Fig. 3.25). The function logic mirrors that of `encrypt()` (4.5), with the primary difference being the source and destination buffers during the decryption process. Instead of encrypting data from the "original buffer" to the "temporary buffer", we decrypt data from the "temporary buffer" to the "original buffer".

The snippet below highlights the key differences:

```
int decrypt(const struct ublksrv_queue *q, const struct ublk_io_data *data, int tag)
{
  [...]
  if (!EVP_DecryptInit_ex2(c_ctx->ctx, c_ctx->cipher, enc->key, NULL, NULL))
    goto err;

  while (nr_sectors) {
    xts.sector = start_sector;
    relative_bytes = relative_sector << 9;
    if (!EVP_DecryptInit_ex2(c_ctx->ctx, NULL, NULL,
          (const unsigned char *)xts.tweak, NULL))
      goto err;
    if (!EVP_DecryptUpdate(c_ctx->ctx, buf + relative_bytes, &decrypt_size,
          t_buf + relative_bytes, 512))
      goto err;

    start_sector++;
    relative_sector++;
    nr_sectors--;
  }
  [...]
}
```

Listing 4.7: *Function decrypt()*

We can observe that the roles of the "temporary buffer" (`t_buf`) and "original buffer" (`buf`) have been switched and of course, unlike the encryption process, here we call the decryption-related functions from the OpenSSL library.

The term "single-thread" in the encryption solution is now clearer. It refers to the step-by-step encryption or decryption of each sector **by the main thread itself**, with the appropriate tweak value set sequentially for every sector involved.

4.4 Intra-Block Encryption Implementation

In our second solution, we manage a thread pool to handle the cryptographic tasks. The cornerstone of this solution is a shared object that facilitates communication between the main thread and the worker threads. Let's take a look into this object and explore the functionality of its various fields.

```
struct shared_obj {
  pthread_barrier_t barrier1;
```

```
3    pthread_barrier_t barrier2;
4
5    struct {
6      int pos;
7      pthread_mutex_t mutex;
8    };
9
10   struct {
11     void *buf;
12     void *tmp_buf;
13   };
14
15   unsigned long request_size;
16   __u64 start_sector;
17
18   struct encryption *enc;
19
20   int encrypt;  // Type of request
21   unsigned long num_threads;
22   int quit; //0 => don't quit, 1 => quit
23   pthread_t threads[];
24 };
```

Listing 4.8: *Shared Object Descriptor*

The fields of this structure are:

- `barrier1, barrier2`: These two barriers are necessary for synchronization between the main and the working threads.
- `pos, mutex`: The `pos` field is used by each thread to acquire a "position" within the pool during their initialization. Based on its position, each worker identifies the area of the buffer it must operate on. Since multiple threads may access the `pos` field concurrently while acquiring positions, a lock (`mutex`) is used to ensure safe access.
- `buf, tmp_buf`: Pointers to the "original buffer" and "temporary buffer" respectively, set by the main thread to guide workers on which buffers to operate on.
- `request_size`: Indicates the size of the request, enabling each worker to compute the number of iterations it has to perform.
- `start_sector`: Denotes the starting sector on disk of the I/O request, crucial for setting the appropriate "tweak" value.

- `enc`: A pointer to a structure holding the master key.
- `encrypt`: Specifies the type of request. A value of 1 indicates a write request requiring encryption from the "original buffer" to the "temporary buffer" before sending data to the backing file. A value of 0 indicates a read request requiring decryption from the "temporary buffer" to the "original buffer" after reading data from the backing file.
- `num_threads`: Indicates the total number of worker threads, which is essential for computing the stride.
- `quit`: Set by the main thread to instruct worker threads to exit.
- `threads`: This is a flexible array member and is used during the creation of the thread pool, storing unique identifiers for every thread, which in turn enables the main thread to await the completion of each worker before exiting.

Examining how this object is used, Figure 3.27 depicts that whenever the main thread needs to notify the worker threads of a request, it populates the fields of the shared object with relevant to this request information.

```
1  // q => struct ublksrv_queue *
2  // iod => struct ublksrv_io_desc *
3  [...]
4  struct shared_obj *so = ublksrv_queue_get_shared_object(q);
5
6  // Fill in the shared object
7  so->encrypt = 1;
8  so->buf = (void *)iod->addr;
9  so->tmp_buf = get_tmp_buf_from_queue(q, tag);
10 so->request_size = iod->nr_sectors << 9;
11 so->start_sector = iod->start_sector;
12 so->quit = 0; // it's not a quit request
13
14 // Let's start the encryption
15 pthread_barrier_wait(&so->barrier1);
16
17 // Wait for working threads to finish
18 pthread_barrier_wait(&so->barrier2);
19 [...]
```

Listing 4.9: *The Main Thread Prepares a Write Request*

The code snippet above demonstrates how the main thread prepares a **request for encryption**, alerts the workers, and waits for the task completion.

For a **decryption request**, the only change is setting `so->encrypt` to 0.

Now, let's inspect the actions of each worker. The following routine is carried out by each worker thread, with the shared object being passed as an argument during thread creation.

```
void *working_pool_fun(void *args)
{
  struct shared_obj *so = (struct shared_obj *)args;

  pthread_mutex_lock(&so->mutex);
  int pos = so->pos++;
  pthread_mutex_unlock(&so->mutex);

  unsigned long stride;
  __u64 sec;
  unsigned long num_threads = so->num_threads;
  struct encryption *enc = so->enc;
  int retval = 1;

  // Initialize encryption/decryption context and cipher
  EVP_CIPHER_CTX *ctx = NULL;
  EVP_CIPHER *cipher = NULL;
  ctx = EVP_CIPHER_CTX_new();
  cipher = EVP_CIPHER_fetch(NULL, "AES-256-XTS", NULL);

  union iv xts = {0};
  int encrypt_size, decrypt_size;

  for (;;) {
    pthread_barrier_wait(&so->barrier1); // Wait for a request from main thread
    // Check if we need to exit
    if (so->quit) {
      EVP_CIPHER_free(cipher);
      EVP_CIPHER_CTX_free(ctx);
      pthread_exit(&retval);
    }

    // Set the cipher and the key for the context
    if (so->encrypt) {
      EVP_EncryptInit_ex2(ctx, cipher, enc->key, NULL, NULL);
    } else {
      EVP_DecryptInit_ex2(ctx, cipher, enc->key, NULL, NULL);
    }
    stride = pos << 9;
    sec = so->start_sector + pos;
```

```
    while (stride < so->request_size) {
      xts.sector = sec;
      if (so->encrypt) {
        // Write request
        if (!EVP_EncryptInit_ex2(ctx, NULL, NULL, (const unsigned char *)xts.tweak, NULL))
          ublk_err("ERROR: EVP_EncryptInit_ex2() failed");

        if (!EVP_EncryptUpdate(ctx, (unsigned char *)so->tmp_buf + stride,
            &encrypt_size, (const unsigned char *)so->buf + stride, 512))
          ublk_err("ERROR: EVP_EncryptUpdate() failed");

      } else {
        // Read request
        if (!EVP_DecryptInit_ex2(ctx, NULL, NULL, (const unsigned char *)xts.tweak, NULL))
          ublk_err("ERROR: EVP_DecryptInit_ex2() failed");

        if (!EVP_DecryptUpdate(ctx, (unsigned char *)so->buf + stride,
          &decrypt_size, (const unsigned char *)so->tmp_buf + stride, 512))
          ublk_err("ERROR: EVP_DecryptUpdate() failed");
      }

      stride += (num_threads << 9);
      sec += num_threads;
    }
    pthread_barrier_wait(&so->barrier2);
  }
}
```

Listing 4.10: *Working Threads Running Function*

Here's a summary of the key activities within in this function. **Each thread**:

1. Determines its position, ensuring that there will be no concurrent access to the `so->pos` field.
2. Allocates the cryptographic context and fetches the cipher implementation for the AES-XTS algorithm, as seen in the single-thread solution.
3. Enters a loop, awaiting a request from the main thread at the first barrier.
4. Wakes up from the barrier when the request arrives, checks if it should terminate, and if not, it sets the context `ctx` with the specific cipher and key for encryption or decryption according to the type of the request it has to manage.

5. Calculates the necessary offsets based on its position, determining its "operational area" within the buffers. The `stride` variable indicates the starting point within the buffer, while the `sec` variable represents the specific sector number necessary to set the tweak value for the cryptographic operations. Since `pos` is a unique value for each thread, this calculation ensures that each thread starts at a different position within the buffers.

6. Starts a while loop, in which it either encrypts, or decrypts one sector at time. The distribution of work is set up such that each thread works on different, non-overlapping 512-byte sectors. After processing its current sector, the thread skips sectors equal to the total number of threads (`num_threads`) and processes the next one. This mechanism ensures that all threads operate simultaneously but on different parts of the buffer, achieving parallelism. Also the current sector is incremented in sync with `stride`, ensuring the correct sector number is used for each operation. For example, considering we have 4 threads (numbered 0 through 3) and a buffer of 16 sectors. Here's how the distribution will look:

   - Thread 0: processes sectors 0, 4, 8, 12
   - Thread 1: processes sectors 1, 5, 9, 13
   - Thread 2: processes sectors 2, 6, 10, 14
   - Thread 3: processes sectors 3, 7, 11, 15

7. Finishes the `while` loop, as soon as the stride exceeds the request size and "hits" the second barrier.

When every thread completes its task, it means the whole request has been processed, and the final thread to reach the second barrier will wake up the remaining worker threads and the main thread, as illustrated in Steps 6 and 7 of Fig. 3.27.

The `union iv` remains consistent with the previous implementation, serving the same purpose. When the sector number is set, the tweak value is also set due to the shared memory space of the union.

4.5 Inter-Block Encryption Implementation

In our third implementation, three key structures are employed to manage cryptographic tasks and help communication between the main thread and workers. These structures and their interconnections are depicted in Figure 3.29. Let's examine their components.

```
struct shared_obj {
  pthread_cond_t cond;
  int efd;
  struct encryption *enc;
  struct queue *submit;
  struct queue *complete;
  struct ublksrv_req *req_array;
  pthread_t threads[];
};

struct queue {
  pthread_mutex_t mutex_queue;
  int num;
  struct ublksrv_req *head;
  struct ublksrv_req *tail;
};

struct ublksrv_req {
  const struct ublksrv_io_desc *iod;
  void *tmp_buf;
  unsigned tag;
  int op; // 0=>read(decryption), 1=>write(encryption), set by main thread
  int quit; // Set by main thread if we need to quit
  struct ublksrv_req *next;
  struct io_uring_cqe cqe;
};
```

Listing 4.11: *Basic Structures in Inter-Block Implementation*

The `struct shared_obj` acts as a bridge between the main thread and worker threads. Similar to the intra-block solution, a shared object is required as the main and worker threads need to collaborate to perform any cryptographic task. Below is a breakdown of the fields in the `struct shared_obj`:

- `cond`: A condition variable used by the main thread to notify worker threads of a request.

- `efd`: An eventfd file descriptor used by worker threads to notify the main thread of job completion.
- `enc`: Holds the master key used in cryptographic operations.
- `submit`: A queue containing requests submitted to workers but not yet processed.
- `complete`: A queue containing requests from workers awaiting processing by the main thread.
- `req_array`: An array holding all potential on-the-fly requests.
- `threads`: A flexible array member holding threads, serving the same role as in the previous implementation.

The communication flow is explained in the Section 3.5.5. In short: the main thread signals the workers about a new job through the condition variable, and upon job completion, the workers notify the main thread via the eventfd file descriptor.

The queues `submit` and `complete` serve as holding areas for requests at **different stages** of processing. The `submit` queue holds the requests ready for processing, while the `complete` queue holds the requests that have been processed and are ready for the main thread to pick up.

The `struct ublksrv_req` plays a central role in managing a task's lifecycle, from the moment the main thread initiates it to its completion by a worker and subsequent processing by the main thread again. The fields of this structure are the following:

- `iod`: This is the task's descriptor, providing workers with necessary details. Its specific role and function is further discussed in Section 3.4.3.
- `tmp_buf`: A pointer to a "temporary buffer". Depending on the task's nature, this buffer is employed by workers either to store the encrypted result before sending it to the target, or to access encrypted content after fetching the data from the target.
- `tag`: A unique identifier for tasks, ensuring their tracking and management throughout their lifecycle.

- op: An indicator the main thread fills out to specify the task type. A “0” signifies a decryption task, whereas a “1” indicates an encryption operation.
- quit: Acts as an exit signal. It informs the worker threads to conclude and terminate their current activities when set by the main thread.
- next: A pointer used to link requests in the queues.
- cqe: Stores a Completion Queue Entry before submitting a read request to a worker. We will clear out its purpose below, after examining exactly what the main thread does when submitting a request.

Working Threads Execution Flow

The following function is the routine each working thread executes. The shared object, given as an argument during thread creation, acts as the synchronization and communication bridge between the main thread and worker threads.

```
void *working_pool_fun(void *args)
{
  struct shared_obj *so = (struct shared_obj *)args;
  uint64_t data = 1;
  int cnt = sizeof(uint64_t);
  int retval = 1;
  struct crypt_ctx *c_ctx = init_crypt_ctx();
  for (;;) {
    struct ublksrv_req *req;

    // Wait for a request
    pthread_mutex_lock(&so->submit->mutex_queue);
    while (so->submit->num == 0) {
      pthread_cond_wait(&so->cond, &so->submit->mutex_queue);
    }
    req = pop_req_from_list(so->submit);
    pthread_mutex_unlock(&so->submit->mutex_queue);

     // Check if we are done
    if (req->quit) {
      pthread_exit(&retval);
    }

    // Execute the request
    execute_req(c_ctx, req, so->enc);

    // Add the request to the completion
```

```
28        pthread_mutex_lock(&so->complete->mutex_queue);
29        add_req_to_list(so->complete, req);
30        pthread_mutex_unlock(&so->complete->mutex_queue);
31
32        // Notify main thread
33        write(so->efd, &data, cnt);
34    }
35 }
```

Listing 4.12: *Working Threads Running Function*

The main workflow of this function can be broken down as:

1. **Initialize a cryptographic context:** Each worker has to perform the encryption or decryption inside of a cryptographic context, as we have already mentioned. To do so it calls the function `init_crypt_ctx()`, which has been shown in Listing 4.4.

2. **Wait for new requests on a condition variable:** Each worker thread enters a loop and first locks the "submit queue mutex" to safely check if there is any available requests. If no request is present, the thread waits on a condition variable, freeing up the mutex, until the main thread signals the arrival of new tasks.

3. **Retrieve a request:** After getting a `pthread_cond_signal()`, a worker awakens, pops out a request from the `submit` queue, and then releases the mutex.

4. **Process the request:** After passing the termination check, the thread moves forward to handle the request using the `execute_req` function. This function completes the request by invoking the necessary encryption or decryption functions, as discussed further below.

5. **Mark the request's completion:** After processing, the thread locks the "complete queue mutex", adds the finished request to the complete queue for the main thread to pick up, and then unlocks the mutex.

6. **Inform the main thread:** At the end, the worker thread alerts the main thread about the task's completion by writing to the eventfd.

Through this infinite loop, each worker thread handles requests, processes them, and communicates their completion to the main thread.

The two helper functions for queue manipulation, `add_req_to_list()` and `pop_req_from_list()`, are always called with the specific queue's lock held, guarding against simultaneous access. Requests are added to the head of a queue and taken from the tail.

```
void add_req_to_list(struct queue *q, struct ublksrv_req *req)
{
  req->next = NULL;
  if (q->head != NULL) {
    //assert(q->num != 0);
    q->head->next = req;
  } else {
    //assert(q->num == 0);
    q->tail = req;
  }
  //assert(q->tail != NULL);
  q->head = req;
  q->num++;
}

struct ublksrv_req *pop_req_from_list(struct queue *q)
{
  //assert(q->tail != NULL && q->num > 0);
  struct ublksrv_req *req;
  req = q->tail;
  q->tail = q->tail->next;
  q->num--;
  if (q->tail == NULL) {
    //assert(q->num==0);
    q->head = NULL;
  }
  return req;
}
```

Listing 4.13: *Addition and Retrieval of a Request*

To wrap up our exploration of the working threads, let's see how request execution occurs. As mentioned before, each request can be either for encryption or decryption, with the op field of `ublksrv_req` indicating which.

The `execute_req` function determines the type of cryptographic task the worker should carry out and calls the relevant function.

```
void execute_req(struct crypt_ctx *c_ctx, struct ublksrv_req *req, struct encryption *enc)
{
  if (req->op) {
    encrypt(c_ctx, req, enc);
```

```
5   } else {
6     decrypt(c_ctx, req, enc);
7   }
8 }
```

Listing 4.14: *Distinguish the Type of the Request*

The `encrypt` and `decrypt` functions follow the same logic as the functions in the single-thread implementation (4.5, 4.7). Every worker computes the results by repeatedly invoking the relevant OpenSSL function to encrypt or decrypt the entire buffer.

Main Thread Execution Flow

While we have explored the operations of worker threads, the role of the main thread in this implementation remains to be discussed. How is the offloading of requests managed?

The main thread submits SQEs and awaits CQEs. In this design, the main thread may be awakened by a CQE originating from one of three sources:

1. A read or write request from the driver.
2. A response to a read or write request from the target.
3. A write to the eventfd from a working thread.

Upon awakening, the server examines two specific bits of the `user_data` field of CQE to acknowledge the type of request it needs to handle.

With that context, let's proceed to closely analyze the implementation for each of these three cases.

Case 1: The Driver Sent a Write or a Read Request

Upon receiving a request from the driver, the server takes action based on whether the request is a "write" or "read" operation.

```
1 [...]
2   if (cqe->res == UBLK_IO_RES_OK) {
3     __u8 real_cmd_op = ublksrv_get_op(io->data.iod);
4     if (real_cmd_op == UBLK_IO_OP_WRITE) {
```

```
5        // A write request came from ublk driver.
6        // Forward the request to a working thread for encryption.
7        // First of all prepare the request.
8       struct ublksrv_req *req = &so->req_array[tag];
9       req->iod = io->data.iod;
10      req->tmp_buf = q->tmp_buf[tag];
11      req->op = 1; //encrypt operation
12      req->quit = 0; // This is not a quit request
13
14      // Now put the request in the ”submit” queue and signal a working thread
15      pthread_mutex_lock(&so->submit->mutex_queue);
16      add_req_to_list(so->submit, req);
17      pthread_cond_signal(&so->cond);
18      pthread_mutex_unlock(&so->submit->mutex_queue);
19    } else {
20      // A read request came from ublk driver. Call this target callback
21      // to prepare a read SQE for the target.
22      q->tgt_ops->handle_io_async(local_to_tq(q), &io->data);
23    }
24  [...]
```

Listing 4.15: *Serving a Request from the Driver*

From the above code, it's evident that in the case of a write request, the server prepares an appropriate `ublksrv_req` structure. It adds the request to the `submit` queue, and subsequently, it signals a worker thread to handle the encryption.

On the other hand, when handling a read request, the server must interact with the target, not the workers. The reason being, the server must first retrieve the encrypted data from the target. To achieve this, it invokes a "target-specific" callback, which prepares a read request directed to the target.

Case 2: Response from the Target to a Write or Read Request

```
1  [...]
2    struct ublk_io *io;
3    io = &q->ios[tag];
4    __u8 real_cmd_op = ublksrv_get_op(io->data.iod);
5    if (real_cmd_op == UBLK_IO_OP_READ) {
6
7      // This is a respond to a read request.
8      // A worker must decrypt the data.
9      // Prepare the request
10     struct shared_obj *so = q->so;
11     unsigned tag = user_data_to_tag(cqe->user_data);
```

```
12      struct ublksrv_req *req = &so->req_array[tag];
13      //assert(tag == req->tag);
14      req->iod = io->data.iod;
15      req->tmp_buf = q->tmp_buf[tag];
16      req->op = 0; // read request
17      req->quit = 0;
18      req->cqe.user_data = cqe->user_data;
19      req->cqe.res = cqe->res;
20      req->cqe.flags = cqe->flags;
21
22      // Put the request in submit list and signal a working thread
23      pthread_mutex_lock(&so->submit->mutex_queue);
24      add_req_to_list(so->submit, req);
25      pthread_cond_signal(&so->cond);
26      pthread_mutex_unlock(&so->submit->mutex_queue);
27    } else {
28      // This is a response to a write request.
29      // Call this target callback to prepare a UBLK_IO_COMMIT_AND_FETCH_REQ SQE for the driver
30      if (q->tgt_ops->tgt_io_done)
31        q->tgt_ops->tgt_io_done(local_to_tq(q),
32            &q->ios[tag].data, cqe);
33    }
34
35 [...]
```

Listing 4.16: *Serving a Response from the Target*

The server again, must choose between different paths based on whether it receives a write or read response from the target.

For write responses, since the encrypted data is already written to the target, the server only invokes a target-specific callback to prepare the response for the driver.

For read responses, data decryption is required. To achieve this, the server prepares a `ublksrv_req`, places it into the `submit` queue, and signals a worker. Notably, in this scenario, the server saves the target's CQE response within the `ublksrv_req` structure. This step is absent when preparing a `ublksrv_req` for encryption, as shown in Listing 4.15. This distinction arises because omitting this step would cause the server to lose track of the target's specific request response.

To understand this nuance, let's review the stages for both write and read requests. The request flows are:

- **Ublk server receives a write request from the ublk driver:**

1. The server adds a ublksrv_req and notifies a worker. The worker does the encryption and responds.
2. The server sends to the target the encrypted data. The target responds.
3. The Server submits an UBLK_IO_COMMIT_AND_FETCH_REQ SQE to the driver to inform about the request.

- **Ublk server receives a read request from ublk driver:**

1. The server submits a read SQE to the target to fetch the encrypted data. The target responds.
2. The server adds a ublksrv_req to the submit queue and notifies a worker. A worker does the decryption and responds.
3. The server submits an UBLK_IO_COMMIT_AND_FETCH_REQ SQE to the driver to inform about the request.

Each "response" in the sequences above stands for a CQE. Thus, in Step 3 of the write request, the server possesses the target's response, allowing it to prepare the SQE that will be submitted to the driver, reflecting the result of the request. In contrast, during the read process in Step 3, the only response the server has is from the worker. This distinction highlights the necessity for a field in the ublksrv_req structure to "hold" the target's CQE for read requests.

Case 3: A worker notifies the server via eventfd

In this third scenario, the server awakens to find a CQE that corresponds to the IORING_OP_POLL_ADD SQE. The server had previously submitted this SQE to receive notifications from the workers.

```
[...]
  struct req_list rl;
  struct shared_obj *so = q->so;
  unsigned long long d;
  const int cnt = sizeof(uint64_t);
  req_list_init(&rl);

  pthread_mutex_lock(&so->complete->mutex_queue);
  req_list_splice(so->complete, &rl);
  // Consume the eventfd.
```

```
11  read(so->efd, &d, cnt);
12  __ublksrv_queue_event_for_main_thread(local_to_tq(q));
13  io_uring_submit_and_wait(&q->ring, 0);
14  pthread_mutex_unlock(&so->complete->mutex_queue);
15
16  /*
17   * Now we have in rl the pointers to the beginning and
18   * the end of the requests that we have to satisfy
19   */
20  struct ublksrv_req *req = rl.tail;
21  while (req != NULL) {
22    struct ublksrv_req *tmp;
23    tmp = req;
24    /*
25     * manipulate the responds.
26     * If it is a write request, we need to handle it,
27     * but if it is a read request, which we have already
28     * handled, we need to proceed to tgt_io_done
29     */
30    struct ublk_io *io;
31    io = &q->ios[req->tag];
32
33    if (req->op) {
34      // Write case
35      q->tgt_ops->handle_io_async(local_to_tq(q), &io->data);
36    } else {
37      // Read case
38      if (q->tgt_ops->tgt_io_done) {
39        q->tgt_ops->tgt_io_done(local_to_tq(q),
40          &q->ios[req->tag].data, &req->cqe);
41      }
42    }
43    req = req->next;
44    tmp->next = NULL;
45  }
46
47 [...]
```

Listing 4.17: *Ublk Server Handles a CQE from eventfd*

Let's dissect the code snippet to understand the actions the main thread performs:

- The main thread first acquires the lock for the `complete` queue. It then copies the `head` and `tail` fields of it to a temporary structure, named "`rl`". This structure is made up of two pointers to `ublksrv_req` and will hold these pointers as the main thread begins processing each completed request. Holding onto these

pointers in a temporary structure prevents the server from retaining the lock on the `complete` queue while processing the completed requests. The function `req_list_splice` also sets the `num` field of the `complete` queue structure to zero.

- While still possessing the lock, the server:
  1. Clears any pending notifications by reading from the eventfd.
  2. Prepares a `IORING_OP_POLL_ADD` SQE for the eventfd.
  3. Submits the SQE and returns immediately without waiting.
  4. Releases the mutex of the `complete` queue.
- After releasing the lock, the main thread now has the pointers to the start and end of the `complete` queue. It must now address each completed request. The `while()` loop serves this purpose. The loop traverses each request, moving from `tail` to `head`, and takes actions based on its type. If it was a completed write request, the server needs to submit a write request to the target and thus calls the `handle_io_async` callback. If it was a read request, the server must finalize the response to the driver and invokes another target callback to achieve this.

Locking Schema

Our design has two primary "communication directions": from the main thread to the worker threads, and vice versa. We'll explain how our locking system keeps the data safe.

From Main Thread to Working Threads
At the heart of this direction is the `submit` queue. The crucial element here is the mutex that safeguards this queue, coming into play when the main thread adds a request and when worker threads retrieve requests from it.

This locking mechanism ensures only one worker manipulates an event. But how can we be certain no "orphaned" job will arise? There are two potential scenarios after the main thread submits a request to the submit queue:

1. At least one worker is already waiting because of `pthread_cond_wait`. In this case, the `pthread_cond_signal` just wakes up one of the waiting threads. This means the request will be taken care of.

2. All workers are active and currently processing jobs. Here, the first worker that concludes its task and loops back into the `for(;;)` loop will find the `while(so->submit->num == 0)` predicate unsatisfied, meaning it won't wait and will instead retrieve the request from the list.

Hence, we can confidently state that we won't encounter unprocessed requests.

From Working Threads to Main Thread

The `complete` queue is central to this communication direction. We've designed our solution with an emphasis on reducing the duration for which locks are held. As a result, when the main thread manipulates the `complete` list, it doesn't maintain any lock. This design choice allows workers to continue if they need to add jobs to the list.

As detailed in Listing 4.17, the main thread acquires the lock of the `complete` queue during the "splicing" process, i.e. while taking the reference of its `head` and `tail`. Furthermore, it continues to hold this lock while reading the eventfd and resubmitting the poll request. This decision is crucial. Imagine a scenario where the main thread only acquires the lock during the "splice" operation for the `complete` queue. If a worker completes a request and writes to the eventfd just before the main thread submits the next poll request, the notification would be missed. To sidestep such pitfalls, the main thread maintains the lock until the poll request for the eventfd is resubmitted. However, it ensures that the lock is released before **processing** each individual request, allowing this way the workers to submit any completed request to the list.

5 Evaluation

In this chapter, we present the evaluation of our implementations. We compare our three cryptographic solutions with the original ublk and measure the overhead each adds to it. This allows us to draw conclusions about the potential benefits or pitfalls of our parallel solutions which will help us find ways to improve them in the future. Our presented metrics were done using fio [Axbc], a configurable program that tests workloads and measures the performance of various components in the I/O stack.

5.1 Machine Specification

We conducted our measurements on an Amazon Web Services (AWS) machine, specifically a "c5d.2xlarge" model [Ama]. This machine uses the x86_64 architecture. Tables 5.1 and 5.2 present its software and hardware specifications respectively.

Operating System	Debian GNU/Linux 12 (bookworm)
Kernel	6.1.0-17-cloud-amd64
Shell	bash 5.2.15
Compiler	gcc (Debian 12.2.0-14) 12.2.0

Table 5.1: *Software Specification*

5.2 fio

Fio is a benchmarking tool that is used mainly for storage performance benchmarking. It can be configured to model nearly any storage model, making it the standard tool

CPU	Intel Xeon Platinum 8275CL
Frequency	3.000GHz
Architecture	x86_64
Socket(s)	1
Cores per socket	4
Threads per core	2
Level 1 Data Cache	128 KiB (4 instances)
Level 1 Instruction Cache	128 KiB (4 instances)
Level 2 Unified Cache	4 MiB (4 instances)
Level 3 Unified Cache	35.8 MiB (1 instance)
Memory	16GiB DIMM DDR4
Drive 1	Amazon Elastic Block Store (15 GiB)
Drive 2	Amazon EC2 NVMe Instance Storage (186.26 GiB)

Table 5.2: *Hardware Specification*

for testing various workloads against disk devices and measuring their performance. It spawns processes (or threads if explicitly set) that perform the specified I/O actions.

And why do we need fio?

We aim to measure the overhead caused by the encryption we've implemented. More specifically, we want to see how **bandwidth** and **latency** are affected. We want to understand the burden our encryption/decryption implementation adds to the ublk framework and compare the different implementations.

In this section, we will analyze the fio script we wrote for the testing purposes. Fio runs either via a job file (i.e. a file script) or through the command line.

To simulate and test our implementations under different environments, we measured four different types of workloads, issued from both a synchronous and asynchronous I/O engines for different request sizes. More specifically, we tested both random and sequential reads and writes issued both synchronously and asynchronously for six different request block sizes (4k, 8k, 16k, 32k, 64k and 1m).

Below, we present an indicative segment of the fio job file for the case of random read. The other three options (sequential read and random/sequential writes) follow the same logic and for the sake of brevity we omit their display here.

```
1 [global]
2 filename=/dev/ublkb0
3 direct=1
4 ioengine=sync
```

```
ramp_time=5s
runtime=1m
time_based
numjobs=1
iodepth=1
rw=randread

#### RANDOM READ ######
[4k_randread]
bs=4k
[8k_randread]
stonewall
bs=8k
[16k_randread]
stonewall
bs=16k
[32k_randread]
stonewall
bs=32k
[64k_randread]
stonewall
bs=64k
[1m_randread]
stonewall
bs=1m
```

Listing 5.1: *fio job file*

The job file typically contains a global section defining shared parameters which are common to the subsequent job sections. In the example above, we have one global section and six job sections, each for a different block size. For this workload, fio will spawn six processes (six jobs in fio terminology, "4k_randread", "8k_randread", "16k_-randread", "32k_randread", 64k_randread" and "1m_randread"), each performing random reads with a different block size.

Let's analyze the parameters of the fio segment 5.1 one-by-one:

- **filename=/dev/ublkb0:** Specifies the device or the file that is the "target" of the I/O. Of course, in our case we specified the emulated block device that ublk ex-

poses. The backing file that ublk uses for any read or write is the NVMe disk, displayed as "Drive 2"in Table 5.2.

- **direct=1:** Specifies whether the I/O will be buffered or not. In our case we set it to non-buffered, meaning that the I/O will be O_DIRECT. This ensures the I/O performed on the block device exposed by ublk will not use the operating system's page cache, providing a more accurate measurement of the "real" device's performance.
- **ioengine=sync:** Defines how the job issues I/O to the file. In our case we have "sync", which stands for "synchronous". For asynchronous metrics, this parameter is set to **io_uring**.
- **ramp_time=5s:** Specifies the time to pass without logging measurements in between workloads. This is useful to let the performance settle before the logging takes place.
- **runtime=1m:** Specifies the duration of each metric.
- **time_based:** Ensures fio runs for the specified runtime duration (one minute in our case), even if the file is completely read or written. It will simply loop over the same workload as many times as the runtime allows.
- **numjobs=1:** The number of processes that will run each workload. In our case each workload is executed by one process. It can be used to setup a larger number of processes do the same thing.
- **iodepth=1:** The number of I/O units to keep in flight against the file. Having iodepth above 1 doesn't make sense in synchronous engine, but for asynchronous engines like io_uring, this parameter can be set higher.
- **stonewall:** Serializes the jobs so that each job starts after the previous one finishes.
- **rw=randread:** Specifies the type of I/O. In this case "randread" stands for random read. For the other I/O types (sequential read and random/sequential write), this parameter is set to "read", "randwrite" or "write" accordingly.
- **bs=4k,8k,16k,32k,64k,1m:** The block size in bytes for each I/O, indicating the "chunk" size for each workload.

So to wrap up our workload has six jobs, each job runs after the previous job ends (due to "stonewall"), and each job runs with the same parameters except the block size. Thus, **each job performs random, synchronous reads, on /dev/ublkb0, for 1 minute with block size equal to 4k, 8k, 32, 64k and 1m each time**.

5.3 Experimental Evaluation

As explained in Section 3.6, AES is the most widely used encryption algorithm today across various domains. However, cryptography comes at a cost. The AES algorithm operates in rounds, with different computations occurring in each round. To address this, Intel initially, designed and integrated a new set of instructions into their processors, specifically to accelerate the AES algorithm.

The Advanced Encryption Standard New Instructions (AES-NI) introduce six new instructions to support AES execution at hardware level [Int].

The processor in the machine where we took the measurements (see Table 5.2) supports AES-NI. The OpenSSL library we used for encryption and decryption also supports the new instructions if the running processor does so.

For this reason, to gain a comprehensive view of our implementations, we ran the workloads with the new instructions both enabled and disabled. Disabling AES-NI means that cryptographic operations are performed in software by the OpenSSL crypto libraries, instead of hardware.

The difference between software and hardware implementations of AES can be tested using the `speed` command that OpenSSL provides. By querying our machine with this command, with AES-NI enabled and disabled, we found that decryption with AES-NI is approximately 11 times faster than without it, while encryption is 9.5 times faster.

Note: To force OpenSSL to disable the new instructions and run the workload via its software implementation, we used the environmental variable `OPENSSL_ia32cap`.

In the next two sections, we present the **bandwidth** and **latency** metrics for our implementations without and with AES-NI support, respectively. After each section there are comments and conclusions on the results. To conserve space in the presentation, we will only display metrics for the **synchronous engine** and for **io_uring with iodepth**

equal to 4, focusing on the random read and write cases. The sequential read and write follow the same logic and are therefore omitted.

Note: In every workload, ublk server was parametrized with the default settings (one queue with 128 requests depth). Also our parallel implementations (ublk-intra and ublk-inter) were operating with **4 workers**.

5.3.1 Metrics Without AES-NI Support

Bandwidth

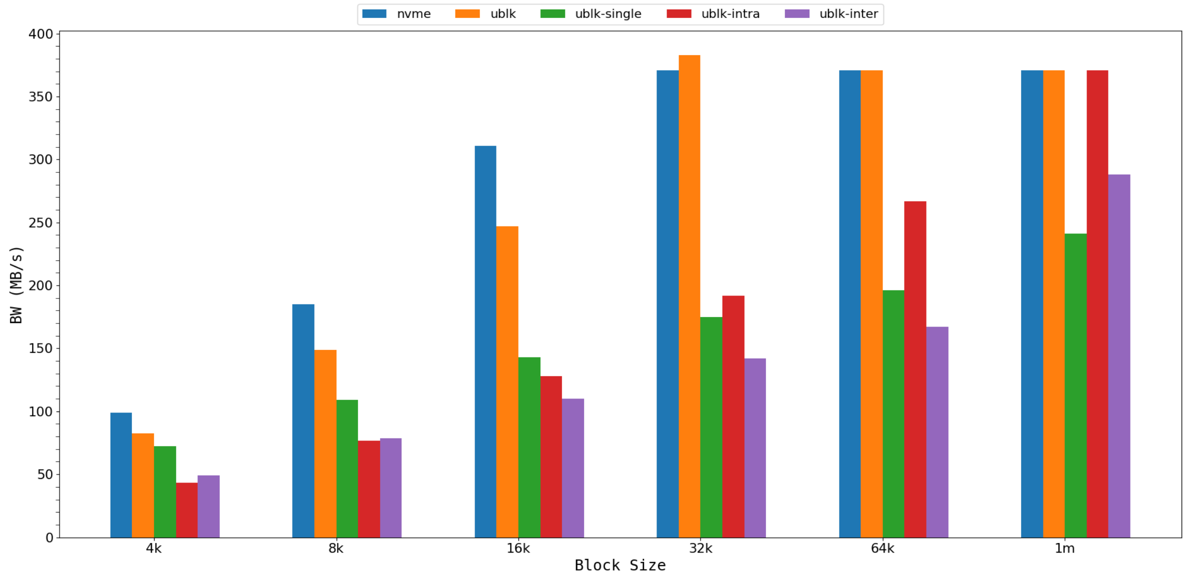

Figure 5.1: *Synchronous Random Read*

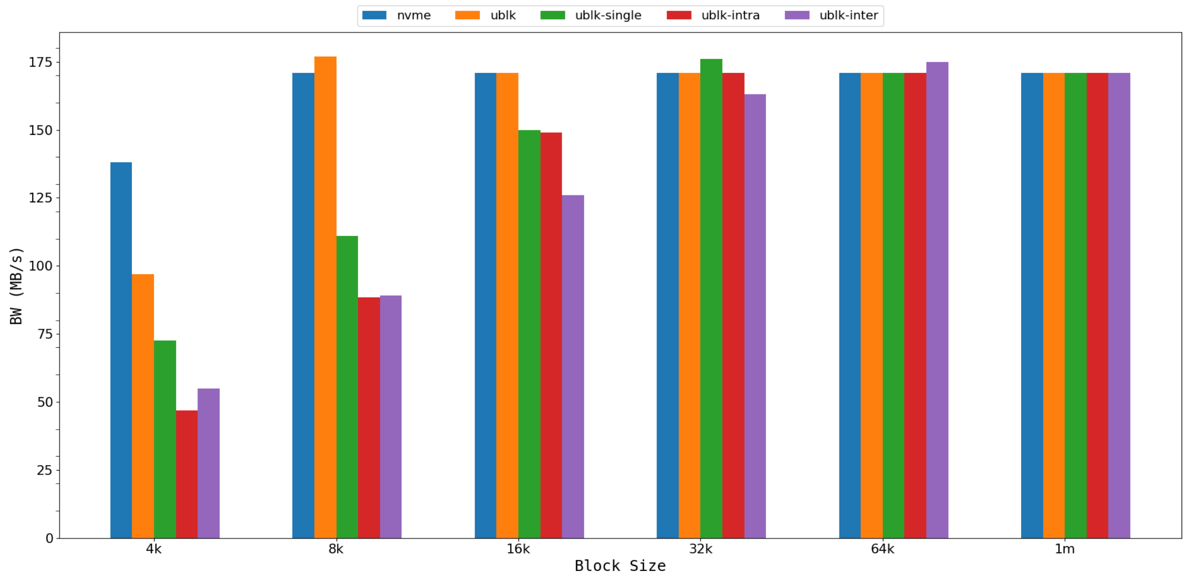

Figure 5.2: *Synchronous Random Write*

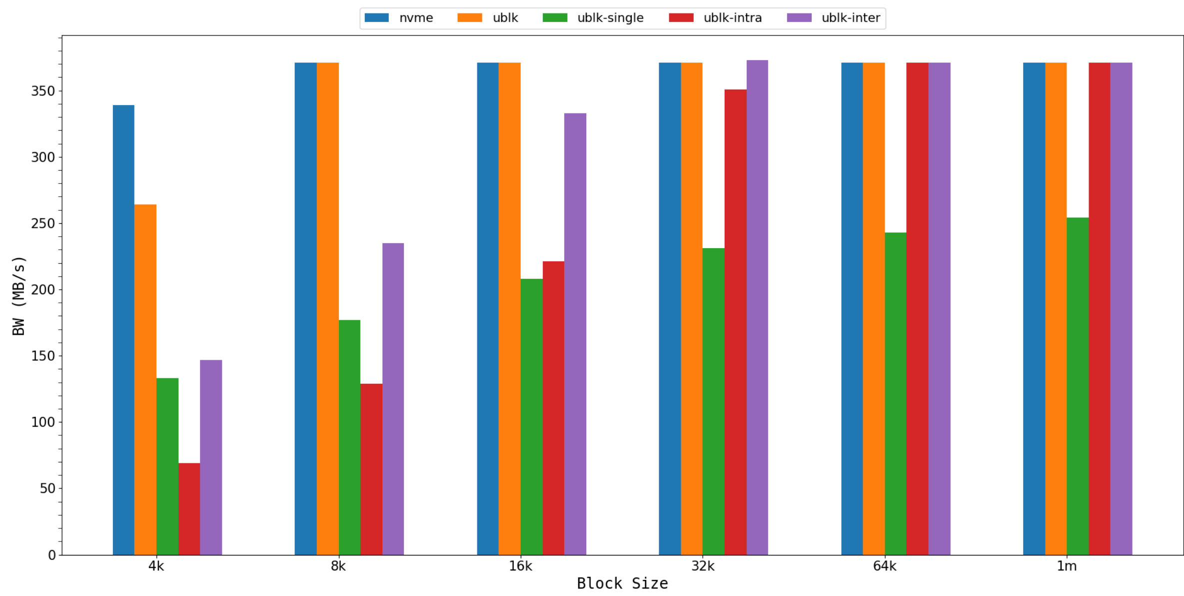

Figure 5.3: *io_uring Random Read (iodepth = 4)*

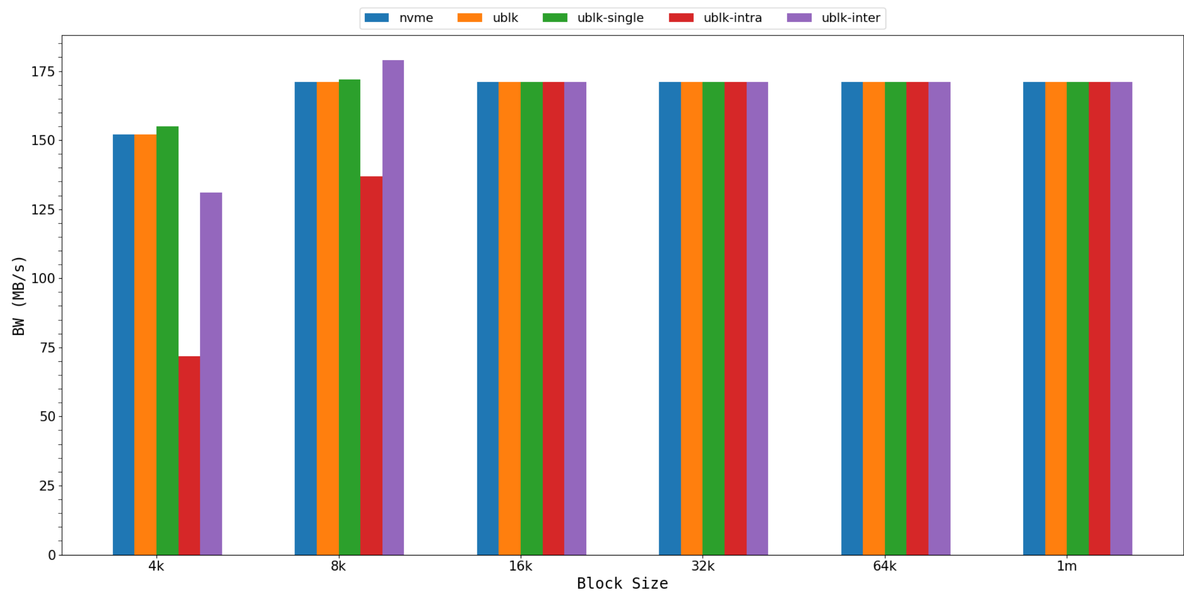

Figure 5.4: *io_uring Random Write (iodepth = 4)*

Latency

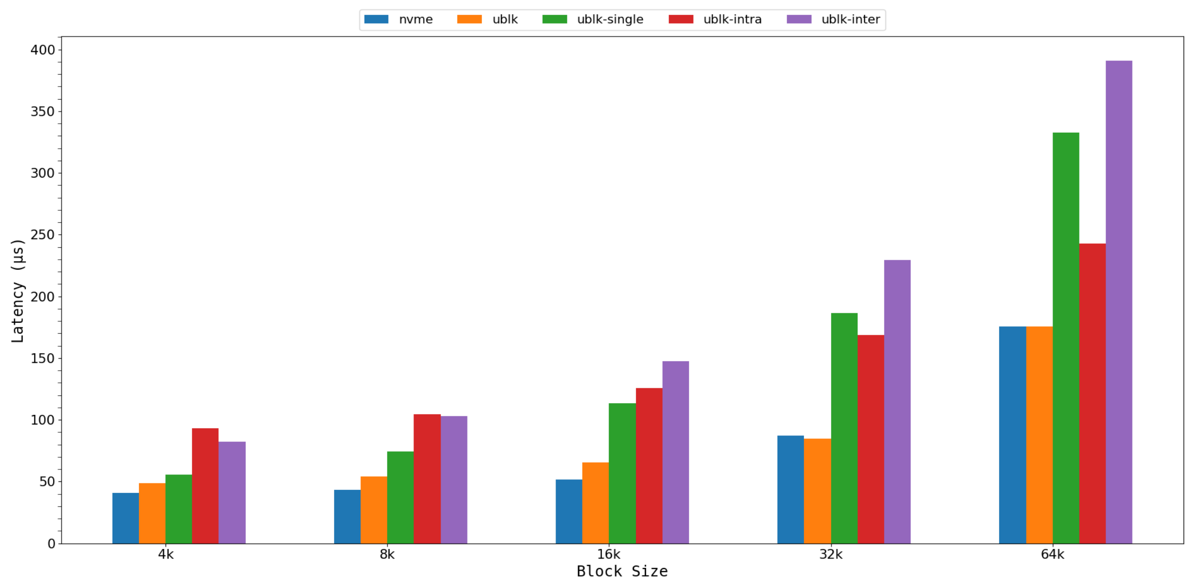

Figure 5.5: *Synchronous Random Read*

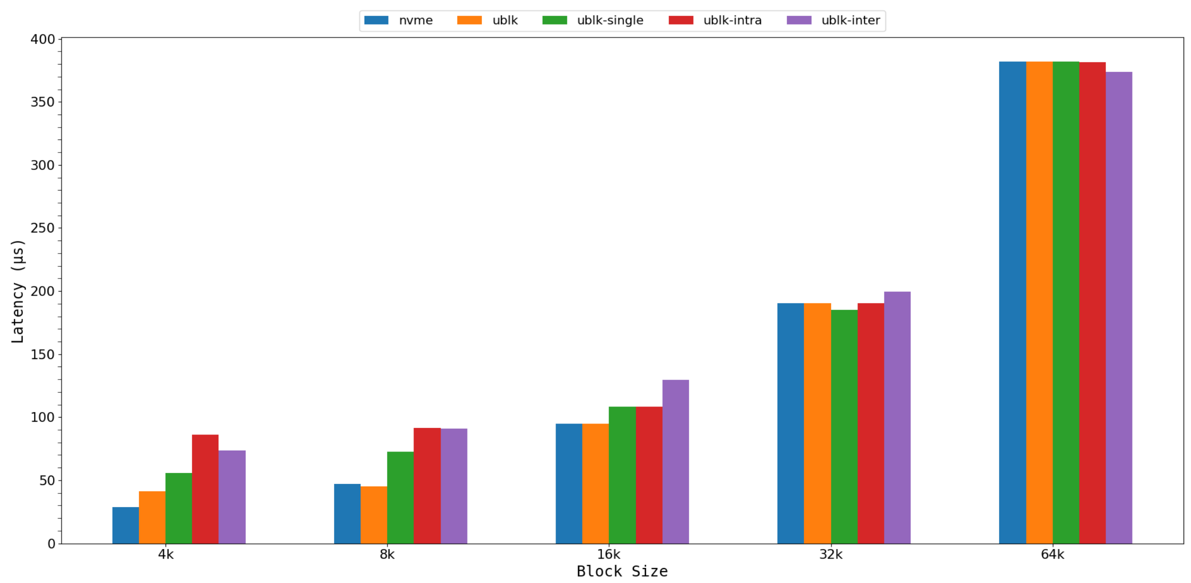

Figure 5.6: *Synchronous Random Write*

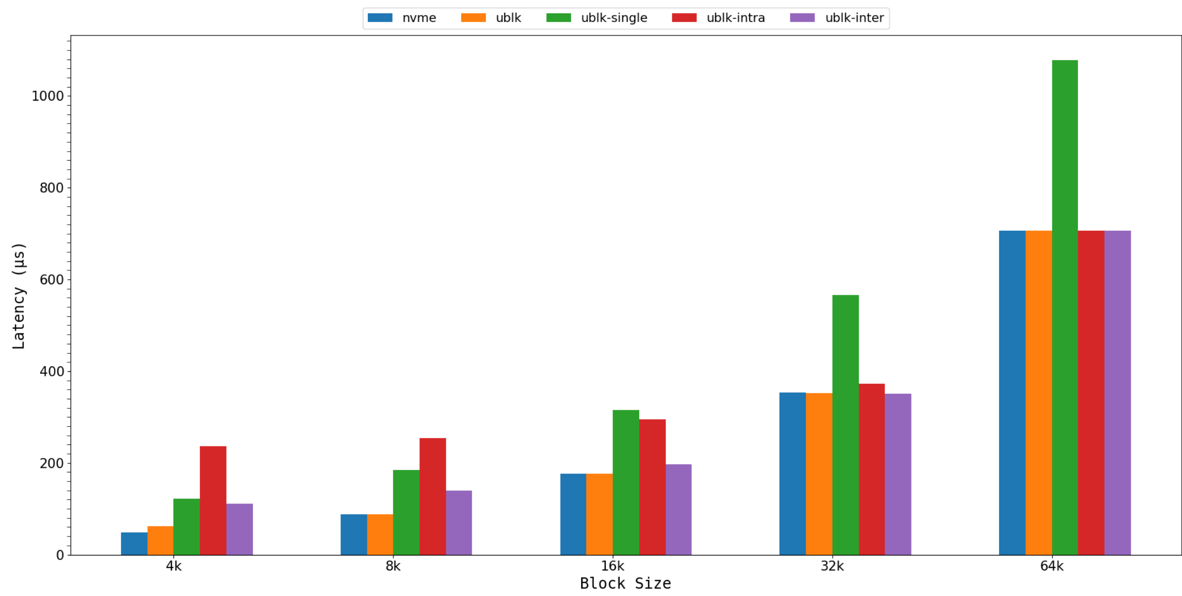

Figure 5.7: *io_uring Random Read (iodepth = 4)*

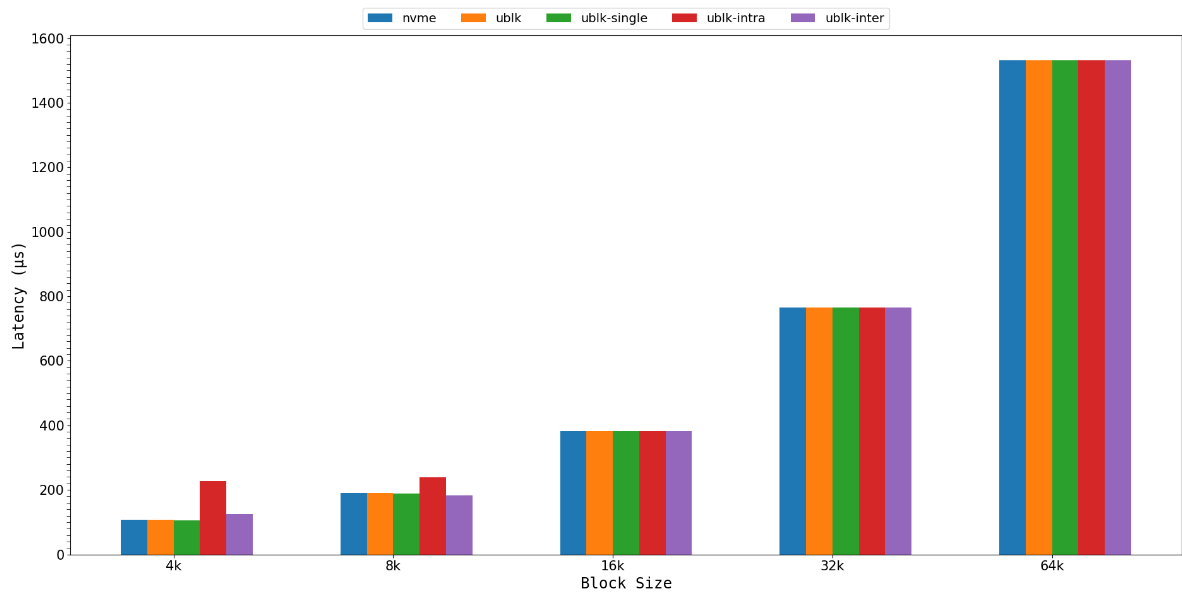

Figure 5.8: *io_uring Random Write (iodepth = 4)*

5.3.2 Comments on Results (no AES-NI support)

To begin with, we can observe from the fio test results that there appears to be a saturation point in the bandwidth of our system for both reads and writes. For the read tests, no test surpassed the 371MB/s limit, while for the write tests, this ceiling was at 171MB/s. Therefore, some results may not be particularly representative, especially those with larger block sizes that reach higher bandwidth. This may indicate a bottleneck in the I/O path at a different spot than our implementations, or it could simply mean that AWS limits the bandwidth when a certain size is reached.

That said, we clearly see that the `nvme` workload records the best performance in all test cases below the saturation point, which we anticipated since this workload runs directly on the disk without any overhead caused by the ublk framework.

Following the raw results from the `nvme` workload, the next best overall workload is the `ublk` one, without any cryptographic operation. This was also expected because cryptographic operations add functionality to the framework, which, of course, impacts overall performance.

Synchronous operations:

- **Write requests:** For both random and sequential writes, the ublk-single implementation performs better for 4k and 8k requests. The ublk-intra has the worst performance for these two block sizes but scales better than the other two, reaching the ublk-single performance for 16k block size. The ublk-inter starts between the other two for 4k requests but scales less than ublk-intra as request size increases.

- **Read requests:** The results are clearer here. Again, the ublk-intra implementation starts as the worst compared to the other two but scales better, recording the best performance for sequential reads after 16k requests and for random reads after 32k. The ublk-inter implementation performs worse than ublk-single for all request sizes except for the 1Mb requests.

Conclusions:

- Write request results for large sizes (32k, 64k, and 1m) seem to reach our system's limit (171MB/s), so we cannot derive meaningful insights from them.
- The read request results were as expected. The ublk-intra implementation is not suited for small requests due to the overhead of worker synchronization plus the inherent parallel implementation overheads (communication, caching). However, as request size increases, the sequential encryption/decryption performed by both ublk-single and ublk-inter becomes more costly than managing workers in the ublk-intra pool, making ublk-intra the better solution for requests of 32k and above.
- Ublk-inter performs worse than ublk-single for all request sizes except 1Mb. In synchronous cases, there is only one request that fio submits and waits for completion, limiting the advantage of parallelism offered by ublk-inter. The main thread simply offloads encryption/decryption to a worker, who performs it sequentially, resulting in worse performance than ublk-single, which does the same without the overhead of thread communication. In the case of 1Mb requests, the block layer sends two requests for execution (due to the 0.5Mb internal buffer limit in the ublk server), allowing parallel execution in ublk-inter and resulting in better performance compared to ublk-single.

io_uring operations:

- **Write requests:** Again, our results in this case aren't very helpful. For request sizes of 16k and above, all our implementations reach the upper limit (171 MB/s). For smaller requests, the results reflect the logic we encountered in the synchronous case, where the ublk-intra implementation performs the worst, and ublk-single performs the best.
- **Read requests:** The ublk-intra implementation starts as the worst compared to the others for both iodepth equal to 2 and 4. In the first case (iodepth = 2), it scales better than ublk-inter and surpasses it for block sizes of 64k. In the second case (iodepth = 4), both hit the bandwidth ceiling for 64k requests.

Conclusions:

- The results of these measurements match our expectations. The ublk-single implementation, though it performs better than in the synchronous case, it does not improve as much as the other two implementations as iodepth increases. Even for the smallest block size (4k), where ublk-single recorded its biggest difference from the others in the synchronous case, in io_uring with iodepth equal to two the difference is smaller, and for iodepth equal to four, ublk-inter performs better. This can be attributed to the fact that even for small requests, when iodepth increases and multiple requests are on the fly, working on them in parallel can make a difference.

- As block size increases, ublk-single cannot scale well, which prevents it from fully taking advantage of iodepth. The time spent on sequentially decrypting larger buffers impacts its performance, which was anticipated.

- Ublk-inter shows better performance in these asynchronous workloads for iodepth equal to 4. We observed a progressive improvement in performance from synchronous to io_uring with iodepth equal to 4. This was expected, as this scenario can fully leverage iodepth by offloading requests to different threads. However, since both ublk-intra and ublk-inter reach the performance limit (371MB/s) for large requests, we cannot be certain if ublk-intra would perform better as size increases, as was the case in synchronous tests.

5.3.3 Metrics With AES-NI Support

Bandwidth

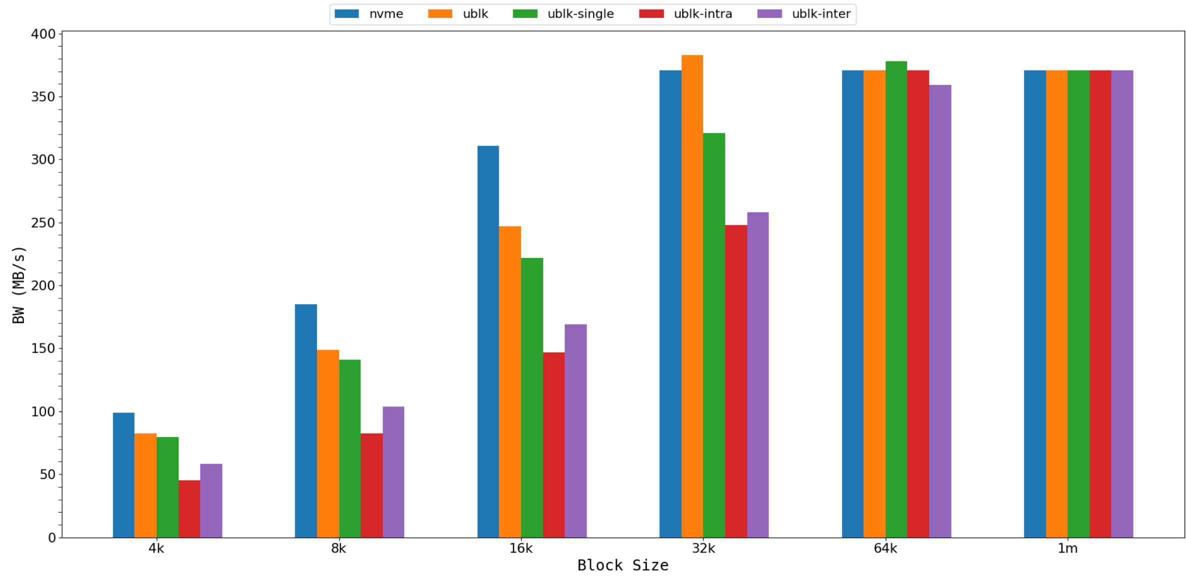

Figure 5.9: *Synchronous Random Read*

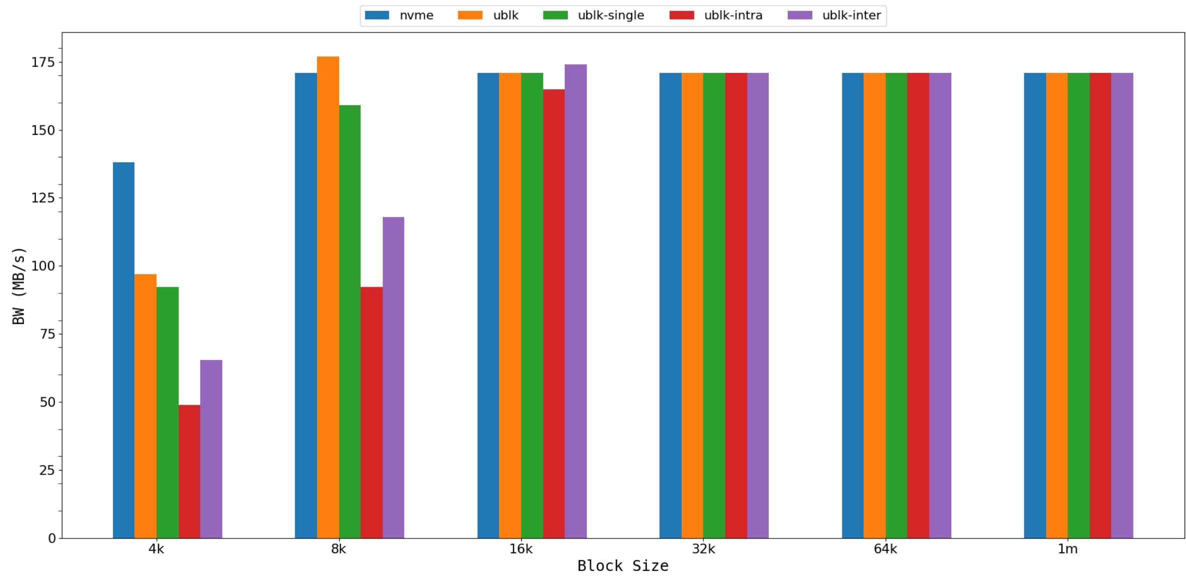

Figure 5.10: *Synchronous Random Write*

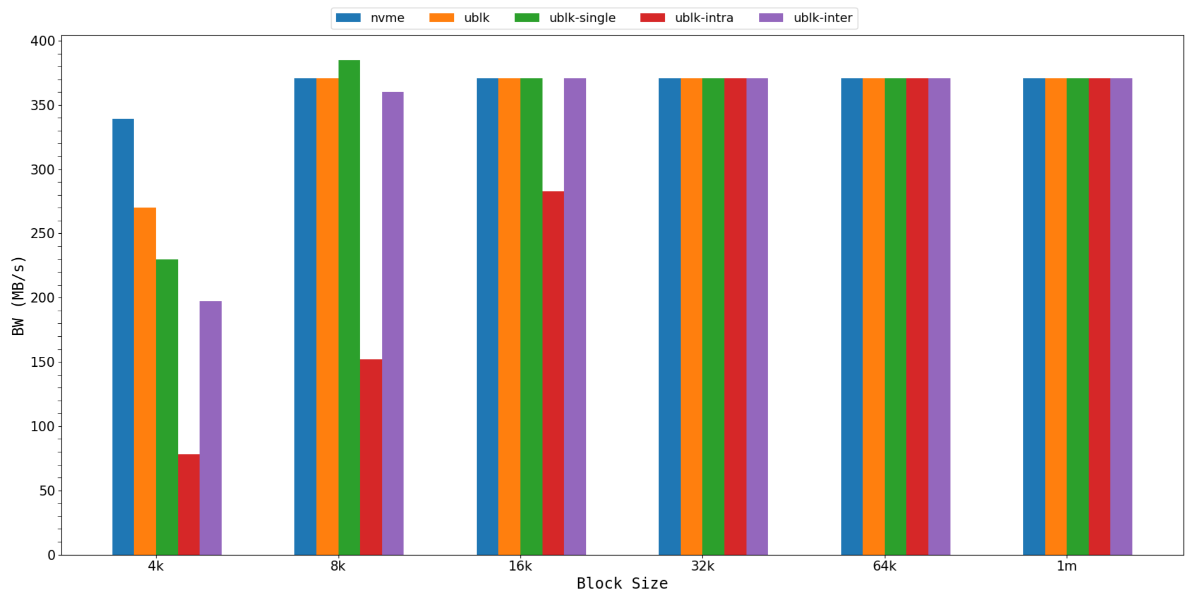

Figure 5.11: *io_uring Random Read (iodepth = 4)*

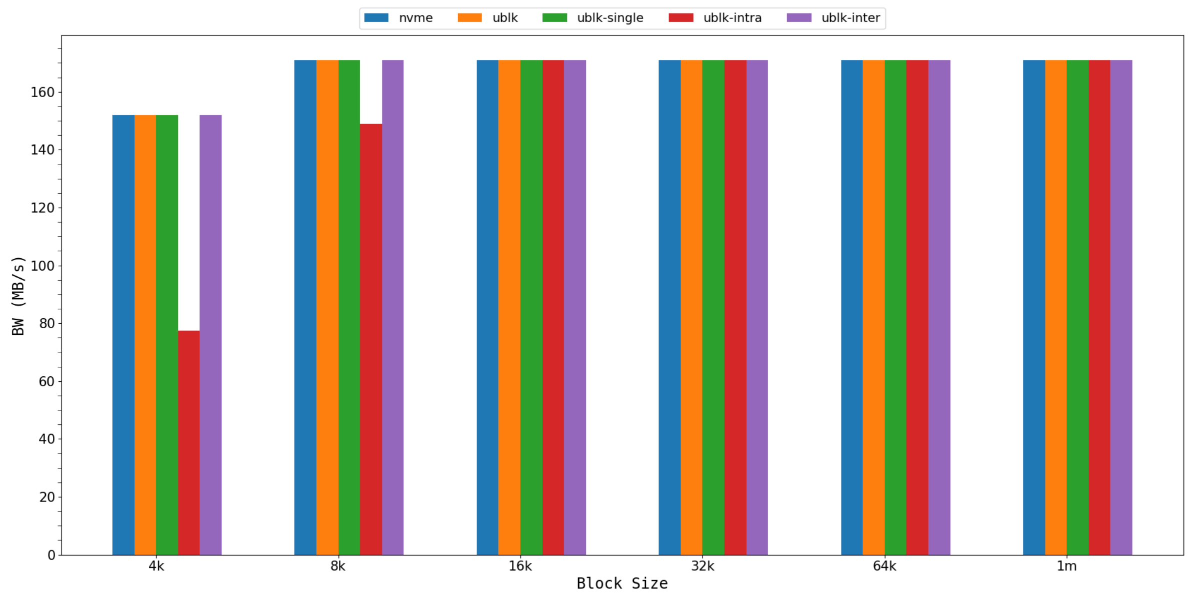

Figure 5.12: *io_uring Random Write (iodepth = 4)*

Latency

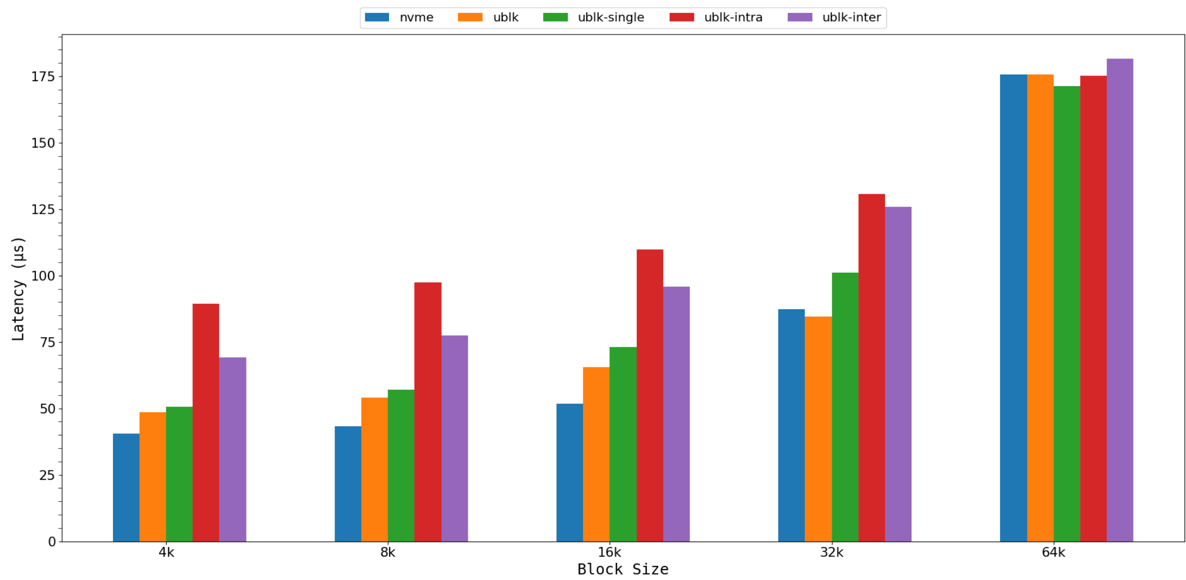

Figure 5.13: *Synchronous Random Read*

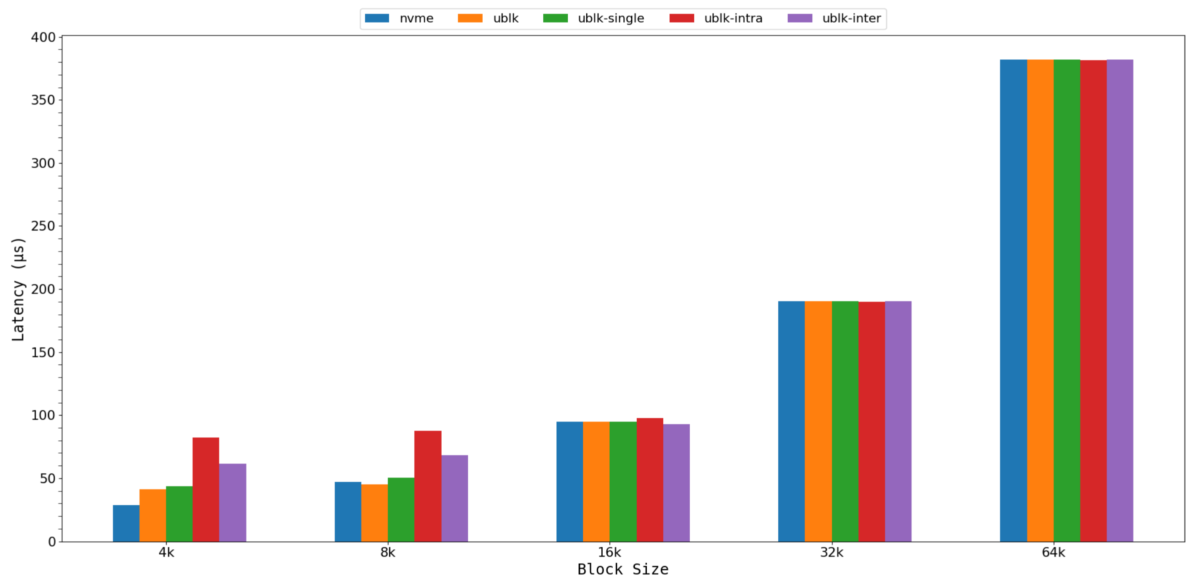

Figure 5.14: *Synchronous Random Write*

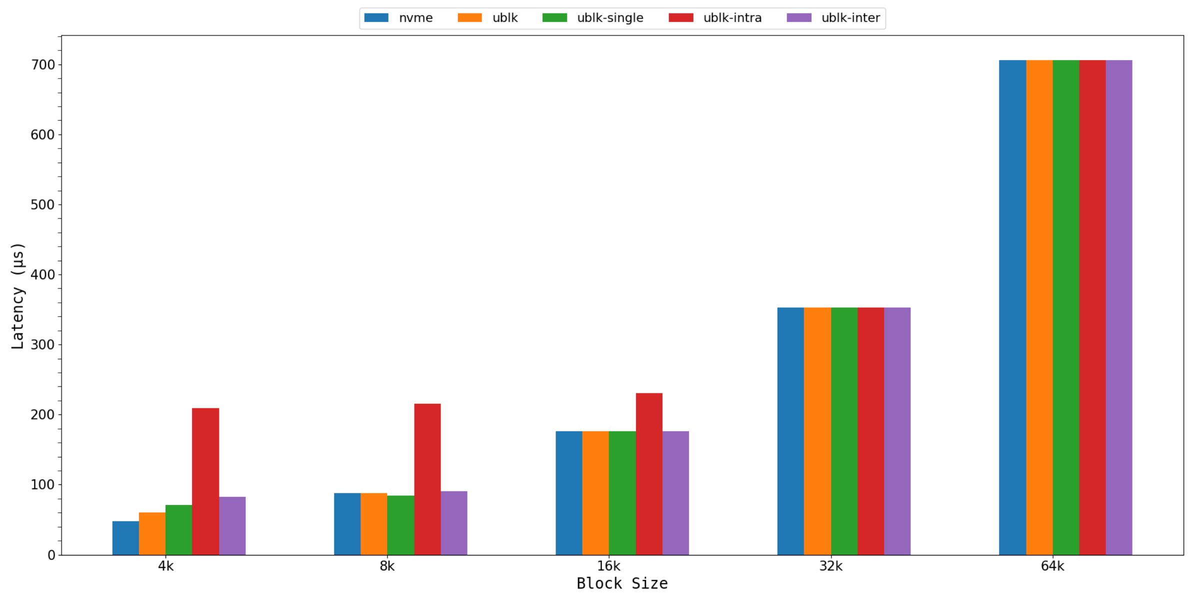

Figure 5.15: *io_uring Random Read (iodepth = 4)*

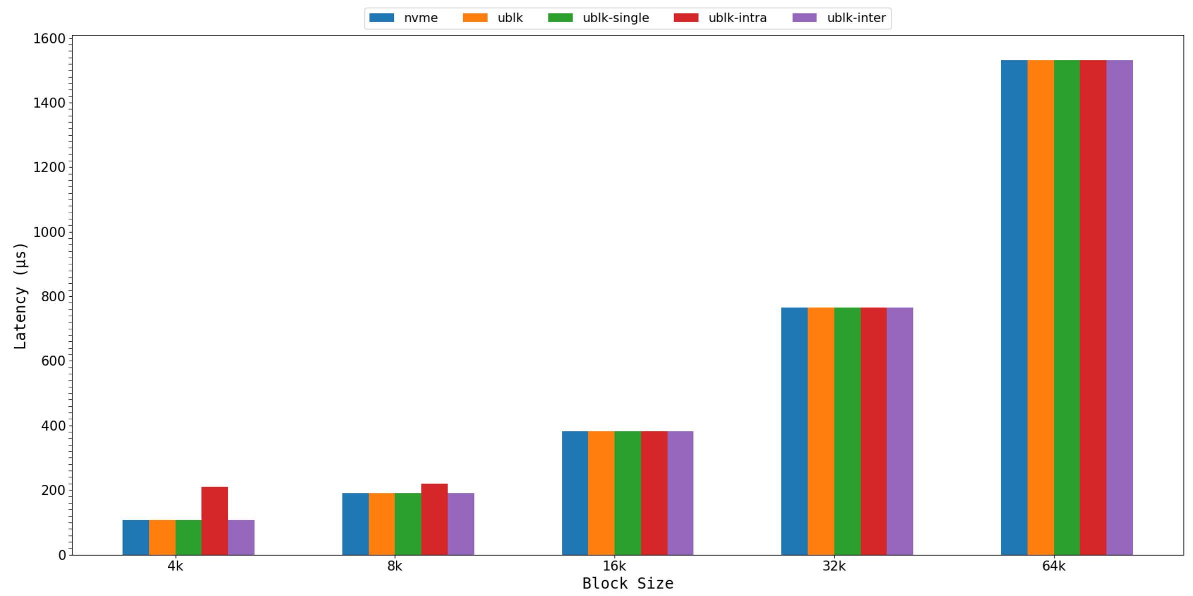

Figure 5.16: *io_uring Random Write (iodepth = 4)*

5.3.4 Comments on Results (with AES-NI support)

Running our workload with AES-NI enabled, our implementations, as expected, perform better than without it. Surprisingly, the ublk-single implementation records the best overall performance, even in asynchronous cases with iodepth equal to 4, which we did not anticipate. With hardware support for AES, which accelerates encryption and decryption by approximately 10 times, ublk-single scales well even for large requests, unlike when AES-NI is disabled. However, all implementations eventually reach a performance ceiling for larger requests, not helping us make conclusions on their scaling capabilities.

This suggests that the overhead of thread communication and scheduling in parallel implementations outweighs the cost of executing AES instructions in hardware. The superior performance of ublk-single could be attributed to two factors: (a) the overhead of scheduling threads, which appears to be more significant than performing AES operations in hardware, and (b) potential cache invalidations in the ublk-intra implementation, where threads working on different CPUs and accessing the same buffers may lead to frequent RAM accesses, reducing efficiency.

Further investigation is needed on a system with higher bandwidth to better understand the scaling factor of each implementation.

6 Conclusion

In this final chapter, we present a brief summary of our work, our conclusions, and potential directions for future research.

6.1 Concluding Remarks

Our journey began over a year ago with a keen interest in Operating Systems and a curiosity about the low-level mechanisms used by the Linux kernel. This curiosity led us to io_uring. After grasping its core concepts, we wanted to apply our knowledge practically, which brought us to the ublk framework.

Initially, the lack of documentation for ublk required us to dive into its source code to unravel its functionalities. This process involved using various tools in both userspace and kernelspace, gradually clarifying the workings of ublk's components. This experience was invaluable, teaching us not only new concepts but also how to approach unfamiliar code bases systematically, which is a crucial skill, especially when dealing with complex systems like the Linux kernel.

Our focus then became more defined: contribute to the ublk project by integrating cryptographic functionality directly into the ublk server. This allowed us to contribute meaningfully to the project while gaining practical experience with industry standards like LUKS and AES-XTS.

Finally, this thesis provided us with the opportunity to explore different aspects of parallelism, evaluate its benefits and limitations, and gain practical insights into its appli-

cation.

6.2 Future Work

Although we have achieved our initial goal, there is still plenty of room for improvement and further research. Below are some directions for future work related to this thesis:

- Explore alternative cryptographic libraries (e.g. libgcrypt [Gnuc]) and compare their performance with the results obtained using OpenSSL.
- Further investigate the optimal configuration for the number of workers in our parallel solutions. We aim to test and measure the performance of ublk-intra and ublk-inter under various workloads and environments with different numbers of worker threads to understand their impact on performance. Additionally, we plan to examine whether binding workers to specific CPUs makes a practical difference in these implementations.
- Develop a more efficient communication method for the ublk-inter encryption schema that minimizes the need for locks. We have begun to implement a concept similar to *plugging* (see more on 2.4.4) in our approach, where the main thread submits a batch of requests to the workers, allowing them to process multiple requests. This approach reduces the need for acquiring a lock with each request submission.
- Contribute and push our work to the upstream ublk project.

Bibliography

[989] iso 9899, *The standard*, https://www.iso-9899.info/wiki/The_Standard#C99, Accessed: 2023/08/30.

[Ama] Amazon, *Amazon EC2 C5 Instances*, https://aws.amazon.com/ec2/instance-types/c5/, Accessed: 2024/1/31.

[Arc] Archlinux, *Archwiki*, https://wiki.archlinux.org/title/Self-encrypting_drives, Accessed: 2023/08/27.

[Axba] Jens Axboe, *blk-mq: new multi-queue block IO queueing mechanism*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=320ae51feed5c2f13664aa05a76bec198967e04d, Accessed: 2023/9/4.

[Axbb] Jens Axboe, *Efficient IO with io_uring*, https://kernel.dk/io_uring.pdf, Accessed: 2023/08/27.

[Axbc] Jens Axboe, *fio - Flexible I/O tester*, https://fio.readthedocs.io/en/latest/fio_doc.html, Accessed: 2024/1/31.

[Axbd] Jens Axboe, *io-wq: provide a way to limit max number of workers*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2e480058ddc21ec53a10e8b41623e245e908bdbc, Accessed: 2023/9/15.

[Axbe] Jens Axboe, *liburing*, `https://github.com/axboe/liburing`, Accessed: 2023/9/15.

[Axbf] Jens Axboe, *polling mode using liburing example*, `https://github.com/axboe/liburing/issues/385`, Accessed: 2023/9/15.

[Bak] Lewiss Baker, *Coroutine Theory*, `https://lewissbaker.github.io/2017/09/25/coroutine-theory`, Accessed: 2023/9/17.

[BC06] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*, O'REILLY, 2006.

[Bro] Milan Broz, *LUKS2 On-Disk Format Specification*, `https://gitlab.com/cryptsetup/cryptsetup/blob/master/docs/on-disk-format-luks2.pdf`, Accessed: 2023/9/13.

[CLO] CLOUDFLARE, *What is data at rest?*, `https://www.cloudflare.com/learning/security/glossary/data-at-rest/`, Accessed: 2023/9/9.

[Cora] Jonathan Corbet, *Descriptorless files for io_uring*, `https://lwn.net/Articles/863071/`, Accessed: 2023/08/27.

[Corb] Jonathan Corbet, *The new way of ioctl()*, `https://lwn.net/Articles/119652/`, Accessed: 2023/9/18.

[CP10] Jan Pelzl Cristof Paar, *Understanding Cryptography: A Textbook for Students and Practitioners*, Springer, 2010.

[Dia] The Geek Diary, *Understanding the /proc/mounts, /etc/mtab and /proc/partitions files*, `https://www.thegeekdiary.com/understanding-the-proc-mounts-etc-mtab-and-proc-partitions-files/`, Accessed: 2023/08/30.

[Doca] Kernel Docs, *Network Block Device (TCP version)*, `https://docs.kernel.org/admin-guide/blockdev/nbd.html`, Accessed: 2023/9/18.

[docb] The Linux Kernel documentation, *Multi-Queue Block IO Queueing Mechanism (blk-mq)*, `https://docs.kernel.org/block/blk-mq.html`, Accessed: 2023/9/4.

[Dwe] Hiba Dweib, *Kernel Uevent*, `https://issuu.com/hibadweib/docs/open_source_for_you_-_october_2012/s/13663276`, Accessed: 2023/08/30.

[F5] F5, *What is SSL/TLS Encryption?*, `https://www.f5.com/glossary/ssl-tls-encryption`, Accessed: 2023/9/3.

[Fru] Clemens Fruhwirth, *New Methods in Hard Disk Encryption*, `https://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf`, Accessed: 2023/9/13.

[GNUa] GNU, *GPGME*, `https://gnupg.org/software/gpgme/index.html`, Accessed: 2023/08/27.

[Gnub] GnuPG, *GnuPG - The Universal Crypto Engine*, `https://www.gnupg.org/related_software/`, Accessed: 2023/9/29.

[Gnuc] GnuPG, *LIBGCRYPT*, `https://gnupg.org/software/libgcrypt/index.html`, Accessed: 2024/2/18.

[Gre20] Brendan Gregg, *Systems Performance*, Mark L. Taub, 2020.

[HS] Sandra Henry-Stocker, *Linux dominates supercomputing*, `https://www.networkworld.com/article/3568616/linux-dominates-supercomputing.html`, Accessed: 2023/08/27.

[Hut] Lee Hutchinson, *Solid-state revolution: in-depth on how SSDs really work*, `https://arstechnica.com/information-technology/2012/06/inside-the-ssd-revolution-how-solid-state-disks-really-work/3/`, Accessed: 2023/08/31.

[Int] Intel, *Advanced Encryption Standard Instructions (AES-NI)*, `https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html`, Accessed: 2024/2/1.

[kD] Linux kernel Documentation, *ublk.rst*, `https://elixir.bootlin.com/linux/v6.3/source/Documentation/block/ublk.rst`, Accessed: 2023/9/19.

[Kou] Vangelis Koukis, *I/O and Scheduling Techniques for the Efficient Utilization of Shared Architectural Resources on Clusters of SMPs*, `http://artemis.cslab.ece.ntua.gr:8080/jspui/bitstream/123456789/8802/1/PD2010-0052.pdf`, Accessed: 2023/9/4.

[Lei] Ming Lei, *Userspace block driver(ublk)*, `https://github.com/ming1/ubdsrv`, Accessed: 2023/08/27.

[Lov13] Robert Love, *Linux System Programming: Talking directly to the kernel and C library*, O'Reilly Media, 2013.

[LWNa] LWN, *driver-core: devtmpfs - driver core maintained /dev tmpfs*, `https://lwn.net/Articles/330985/`, Accessed: 2023/08/30.

[LWNb] LWN, *Driver-Core: devtmpfs - remove EXPERIMENTAL and enable it by default*, `https://lwn.net/Articles/370422/`, Accessed: 2023/08/30.

[LWNc] LWN, *Kernel development*, `https://lwn.net/Articles/369883/`, Accessed: 2023/08/30.

[MAAR] Khairulmizam Samsudin Mohammad Ahmed Alomari and Abdul Rahman Ramli, *Implementation of a Parallel XTS Encryption Mode of Operation*, `https://sciresol.s3.us-east-2.amazonaws.com/IJST/Articles/2014/Issue-11/Article13.pdf`, Accessed: 2023/08/27.

[MBB] David Nellans Matias Bjørling, Jens Axboe and Philippe Bonnet, *Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems*, `https://kernel.dk/blk-mq.pdf`, Accessed: 2023/9/5.

[Mic] Microsoft, *BitLocker settings reference* , `https://learn.microsoft.com/en-us/mem/configmgr/protect/tech-ref/bitlocker/settings`, Accessed: 2023/9/9.

[mpa] Linux manual page, *aio (7)*, `https://man7.org/linux/man-pages/man7/aio.7.html`, Accessed: 2023/9/13.

[mpb] Linux manual page, *eventfd(2)*, `https://man7.org/linux/man-pages/man2/eventfd.2.html`, Accessed: 2023/9/27.

[mpc] Linux manual page, *io_uring_register(2)*, https://manpages.debian.org/unstable/liburing-dev/io_uring_register.2.en.html, Accessed: 2023/9/14.

[mpd] Linux manual page, *loop(4)*, https://man7.org/linux/man-pages/man4/loop.4.html, Accessed: 2023/08/27.

[mpe] Linux manual page, *open(2)*, https://man7.org/linux/man-pages/man2/open.2.html, Accessed: 2023/9/2.

[mpf] Linux manual page, *pthread_barrier_init(3)*, https://linux.die.net/man/3/pthread_barrier_init, Accessed: 2023/9/26.

[mpg] Linux manual page, *udev(7)*, https://man7.org/linux/man-pages/man7/udev.7.html, Accessed: 2023/08/30.

[mph] Linux manual pages, *readv(2)*, https://man7.org/linux/man-pages/man2/readv.2.html, Accessed: 2023/9/15.

[NIS] NIST, *The XTS-AES Mode for Confidentiality on Storage Devices*, https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38e.pdf, Accessed: 2023/08/27.

[Ope] OpenSSL, *OpenSSL - Cryptography and SSL/TLS Toolkit*, https://www.openssl.org/, Accessed: 2023/9/29.

[Phe] PheonixNAP, *How Does SSH Work*, https://phoenixnap.com/kb/how-does-ssh-work, Accessed: 2023/9/9.

[Red] RedHat, *Chapter 11. Encrypting block devices using LUKS*, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/security_hardening/encrypting-block-devices-using-luks_security-hardening, Accessed: 2023/9/9.

[RKH05] Jonathan Corbet Alessandro Rubini and Greg Kroah-Hartman, *Linux Device Drivers*, O'REILLY, 2005.

[Sit] Jakub Sitnicki, *Missing Manuals - io_uring worker pool*, https://blog.cloudflare.com/missing-manuals-io_uring-worker-pool/, Accessed: 2023/9/15.

[Sta] StackExchange, *Is it necessary to mount devtmpfs with /etc/fstab?*, https://unix.stackexchange.com/questions/619589/is-it-necessary-to-mount-devtmpfs-with-etc-fstab, Accessed: 2023/08/30.

[Teaa] Linux Kernel Teaching, *Character Device Drivers*, https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html, Accessed: 2023/08/29.

[Teab] Linux Kernel Teaching, *Linux Kernel Labs*, https://linux-kernel-labs.github.io/refs/heads/master/lectures/intro.html, Accessed: 2023/08/29.

[Tec] Kinsgston Technology, *NAND Flash Technology and Solid-State Drives (SSDs)*, https://www.kingston.com/en/blog/pc-performance/nand-flash-technology-and-ssd, Accessed: 2023/08/31.

[Top] Top500.org, *List Statistics*, https://top500.org/statistics/list/, Accessed: 2023/08/27.

[Ven08] Sreekrishnan Venkateswaran, *Essential Linux Device Drivers*, Prentice Hall, 2008.

[Vera] VeraCrypt, *AES*, https://veracrypt.eu/en/AES.html, Accessed: 2023/9/9.

[Verb] Adarsh Verma, *Linus Torvalds's Famous Email — The First Linux Announcement*, https://fossbytes.com/linus-torvaldss-famous-email-first-linux-announcement/, Accessed: 2023/08/28.

[Wika] Wikipedia, *Advanced Encryption Standard*, https://en.wikipedia.org/wiki/Advanced_Encryption_Standard, Accessed: 2023/08/27.

[Wikb] Wikipedia, *Block cipher*, https://en.wikipedia.org/wiki/Block_cipher, Accessed: 2023/9/3.

[Wikc] Wikipedia, *Block cipher mode of operation*, https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation, Accessed: 2023/9/7.

[Wikd] Wikipedia, *Ciphertext stealing*, `https://en.wikipedia.org/wiki/Ciphertext_stealing`, Accessed: 2023/9/9.

[Wike] Wikipedia, *Flash memory*, `https://en.wikipedia.org/wiki/Flash_memory`, Accessed: 2023/08/31.

[Wikf] Wikipedia, *Hard link*, `ihttps://en.wikipedia.org/wiki/Hard_link`, Accessed: 2023/9/1.

[Wikg] Wikipedia, *History of Linux*, `https://en.wikipedia.org/wiki/History_of_Linux`, Accessed: 2023/08/28.

[Wikh] Wikipedia, *Solid-state drive*, `https://en.wikipedia.org/wiki/Solid-state_drive`, Accessed: 2023/08/29.