TOWARDS A FAST NVMe LAYER FOR A DECOMPOSED KERNEL

by

Abhiram Balasubramanian

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

December 2017

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Abhiram Balasubramanian**
has been approved by the following supervisory committee members:

Anton Burtsev ,	Chair(s)	**18 Aug 2017** Date Approved
Ryan Stutsman ,	Member	**18 Aug 2017** Date Approved
Robert Ricci ,	Member	**18 Aug 2017** Date Approved
Kobus Van der Merwe ,	Member	**18 Aug 2017** Date Approved

by **Ross Whitaker** , Chair/Dean of
the Department/College/School of **Computing**
and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

Operating system (OS) kernel extensions, particularly device drivers, are one of the primary sources of vulnerabilities in commodity OS kernels. Vulnerabilities in driver code are often exploited by attackers, leading to attacks like privilege escalation, denial-of-service, and arbitrary code execution. Today, kernel extensions are fully trusted and operate within the core kernel without any form of isolation. But history suggests that this trust is often misplaced, emphasizing a need for some isolation in the kernel.

We develop a new framework for isolating device drivers in the Linux kernel. Our work builds on three fundamental principles: (1) strong isolation of the driver code; (2) reuse of existing driver while making no or minimal changes to the source; and (3) achieving same or better performance compared to the nonisolated driver. In comparison to existing driver isolation schemes like driver virtual machines and user-level device driver implementations, our work strives to avoid modifying existing code and implements an I/O path without incurring substantial performance overhead. We demonstrate our approach by isolating a unmodified driver for a null block device in the Linux kernel, achieving near-native throughput for block sizes ranging from 512B to 256KB and outperforming the nonisolated driver for block sizes of 1MB and higher.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

To my life-coach, my late grandfather Ramanathan: I owe everything to you. I would not be where I am today, if not for you; many thanks!

I would like to thank my advisor, Anton Burtsev, for guiding me through university life and motivating me in this work. He showed immense faith in me, despite my apprehensions, and steered me in the right direction.

Ryan Stutsman has been instrumental in guiding me through research and graduate school. I could always walk up to his door, whenever I had questions about my research or coursework. He taught me how to present research ideas concisely. I have always enjoyed the lengthy discussions at his office, and I would like to thank him for agreeing to be on my committee.

I would also like to thank Robert Ricci for being one of the prime reasons to attend the University of Utah for graduate school. I still vividly remember the phone call I made from India to discuss the research opportunities. He was kind enough to explain me every possible detail and pointed me to contact the right people matching my interests.

Charles Jacobsen and Vikram Narayanan have been a tremendous help during my master's program. I have had so much fun discussing new ideas and I am grateful for having such wonderful people around. I would also like to thank my fellow lab mates who contributed to this work: Scott Bauer, Sarah Spall, and Michael Quigley.

Special thanks to my mother Lalitha for her undying encouragement and love throughout my life. At last, I would also like to thank my family and friends for the continuous support throughout my years of study.

The work presented in this thesis was supported by the National Science Foundation under Grants No. 1319076 and No. 1527526.

CHAPTER 1

INTRODUCTION

A typical monolithic operating system (OS) used today [24, 26, 28, 29, 33] runs numerous kernel extensions including device drivers, filesystems, and storage stacks tightly intertwined with the core kernel. These extensions are fully trusted, co-exist in the same address space of the OS kernel, and have unfettered access to kernel data structures. But evidence reveals that this trust is often misplaced. Due to the lack of isolation across kernel subsystems, kernel vulnerabilities are routinely discovered and exploited particularly in kernel extensions, resulting in a poor overall security of an operating system.

In 2016, the Common Vulnerabilities and Exposures (CVE) database lists 217 Linux kernel vulnerabilities that lead to denial-of-service (DoS), memory corruption, and other exploits [31]. Despite the diverse nature of the vulnerabilities, exploitable vulnerabilities are particularly dangerous because they allow an attacker to write arbitrary values to sensitive kernel data structures, overwrite kernel memory to execute attacker-defined code, etc. For instance, CVE-2016-8633 [11] reports an arbitrary code execution vulnerability in the FireWire driver allowing remote attackers to execute arbitrary code via crafted fragmented packets. The driver lacked input validation while handling incoming fragmented datagrams, which led to a copy of data past the datagram buffer, enabling the attacker to execute code in kernel memory.

Several protection mechanisms have been developed to confine the effects of vulnerable code. For instance, static and dynamic mechanisms like stack guards and address space layout randomization (ASLR) have been designed to protect the kernel from arbitrary code execution [10, 35], but attackers always come up with new ways to defeat these protection techniques [34]. Lack of isolation implies that, once a monolithic OS like Linux is breached, the attacker has full control over a system, thereby bypassing any protection guarantees the system may have had. This strongly emphasizes a need for isolating kernel extensions

to improve the overall security of an operating system.

In this work, we focus on isolating device drivers in the Linux kernel because they introduce a significant fraction of vulnerabilities to the kernel. Our analysis of kernel vulnerabilities in 2016 reveals that out of 217 kernel vulnerabilities, 54 were from device drivers, 33 were from the network subsystem, and 22 were from the filesystem layer of the kernel. Device drivers account for the primary source of vulnerabilities. First, they are vulnerable to malformed input from userspace applications, allowing attackers to write arbitrary data into kernel memory, execute code with kernel privileges, etc. Second, by their very nature, they handle complex asynchronous semantics like interrupts, hotplug events, and parallel threads of execution, making them vulnerable to programming errors. Finally, drivers also copy data from devices into kernel memory, allowing a slew of possibilities for an attacker to gain control of the system.

Several projects in the past have attempted to isolate device drivers to confine the effects of faulty driver code [14, 23, 30, 39, 40] (refer to Chapter 6). However, we argue that these efforts either compromise performance for safety or require significant development effort. In this work, we explore the feasibility of isolating device drivers in Linux, in a way that improves security and performance, while also being transparent to the source.

Thesis Statement: **Unmodified drivers can be isolated with little effort while not compromising performance.**

1.1 Contributions

In this work, we develop general techniques for isolating high-performance device drivers in the Linux kernel. We demonstrate the feasibility of our approach by developing an isolated driver for a *null block device*. In this process, we present a detailed analysis of I/O submission and completion paths in the Linux kernel. On top of the available techniques for isolation developed in our previous work [20], we add the following support for isolating device drivers: (a) provide infrastructure for driver modules to communicate with user space processes; (b) modify the user application to implement a zero-copy data path from application to the isolated driver; and finally, (c) we introduce asynchronous behavior in kernel code to improve I/O performance. In order to understand the security guarantees of our system, we classify the Linux kernel vulnerabilities in driver modules

reported in 2016 and discuss the types of vulnerabilities addressed by our framework.

1.2 Architecture Overview

Figure 1.1 depicts the overall architecture of our system. The primary objective is to provide strong isolation for the unmodified driver code in a lightweight manner. For strong isolation guarantees, we rely on hardware-assisted virtualization (Intel VT-x), because with VT-x, we can isolate code in separate address spaces and also restrict access to privileged registers. Our isolation architecture comprises two parts, the isolated and nonisolated part. The isolated part runs the unmodified driver code inside an Intel VT-x container and we refer to these containers as *Lightweight Capability Domains* (LCDs). On the other hand, the nonisolated part runs a native Linux kernel that boots and executes as before and also embeds a type 2 hypervisor (Microkernel as shown in Figure 1.1) to manage the LCDs. This implies that we introduce a microkernel architecture within the Linux kernel without affecting the rest of the system.

Since the code inside an LCD is isolated, it can no longer invoke kernel functions directly or interact with the rest of the system using shared memory. We provide a small library, `liblcd`, within the LCD, to provide minimal infrastructure like memory allocation and synchronization primitives. We also provide another library, `kliblcd`, which provides a similar interface for the nonisolated code to interact with an LCD.

By design, we do not share anything with or among the LCDs. The *glue layer* provides access to core Linux functions that are not provided by `liblcd`. For instance, the *glue layer* intercepts the `register_blkdev` function invoked by the isolated driver and translates it into an IPC message. The `libfipc` library, implemented as a part of `liblcd`, provides a message passing interface to the isolated code. The *glue layer* seamlessly translates shared memory interaction patterns into message passing, providing an illusion to the isolated code that it still operates in a shared memory environment. We also install a layer of glue code in the nonisolated part to receive the IPC message for `register_blkdev`, invoke the real kernel function, and return the response to the LCD through another IPC message.

1.2.1 Benefits

We argue that there are several benefits to our approach: First, we eliminate the need for a full OS stack inside the LCD to resolve the dependencies of the unmodified driver. We keep the `liblcd` as small as possible and address all other dependencies using glue code. We construct the glue code systematically based on a set of decomposition patterns that provide straightforward steps for translating shared memory protocols into message passing. Second, we eliminate the manual effort of developing glue code by using an Interface Definition Language (IDL) and a compiler that translates the IDL into glue code. Finally, our fast asynchronous IPC mechanism, provided by a library `libfipc` [3], allows us to perform cache-line aligned message transfers to improve the communication between the isolated driver and nonisolated kernel. These benefits made it feasible to isolate the null block driver in Linux kernel. We also believe that it is possible to decompose and isolate other drivers without actually having to re-write it from scratch and still achieve near-native performance.

1.3 Outline

The rest of the thesis is laid out as follows. Chapter 2 provides background information such as basic concepts of the block layer, architecture of the Multi-Queue block layer, and describes the I/O path in detail. Chapter 3 introduces our isolation approach, describes different techniques used for isolation, and provides a description of how the null block driver is isolated. Chapter 4 summarizes our results and evaluation. Chapter 5 presents our vulnerability analysis. Chapter 6 discusses related work. Chapter 7 presents our conclusions and future work.

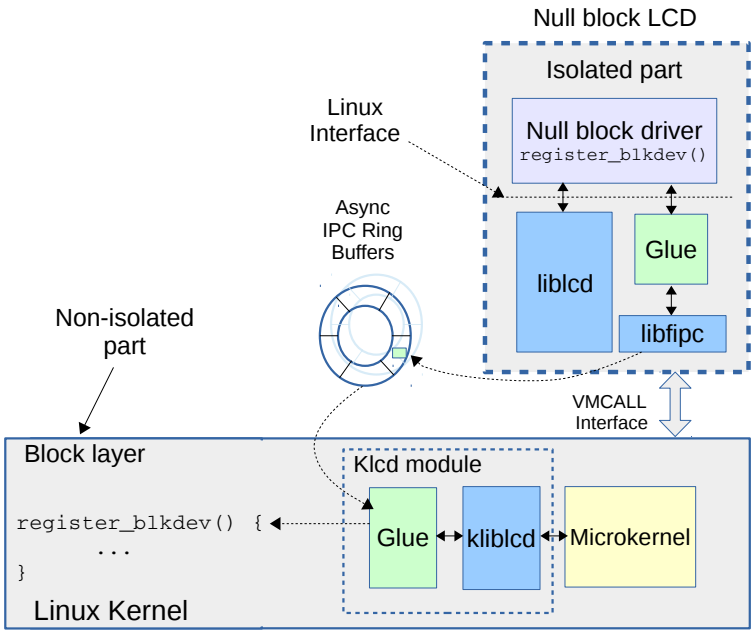

Figure 1.1. Overview of isolation architecture

CHAPTER 2

BACKGROUND

This chapter describes the basic concepts of the Linux block layer, key data structures used by the kernel to represent an I/O request, and finally, the I/O submission and completion paths in the kernel. It is helpful for the reader to understand the control flow and data structures involved in the I/O path since they benefit in comprehending the isolation of the null block driver described in the next chapter.

2.1 Overview of the Linux Block Layer

The block layer is a kernel subsystem that is responsible for handling block devices like hard disks, solid state disks (SSDs), and CD-ROMs in the system. Managing block devices naturally brings in a complexity owing to the nature of these devices. Unlike character devices (e.g., keyboards and printers), where data is always read/written sequentially, access to block devices are entirely random. Apart from user applications, other parts of the kernel also access the block I/O subsystem. For instance, the virtual memory subsystem moves inactive pages in the memory to a block device to make system resources available. If the block I/O subsystem is not well tuned, it can affect the performance of an entire system. For these reasons, the kernel maintains a separate subsystem to abstract the complexity of block devices and provide useful block-related services to other parts of the kernel.

2.1.1 Key Concepts and Data Structures

The block layer is the main interface between applications and the storage medium. Applications submit I/O requests to the kernel via a system call. Each I/O request contains information such as the *address* of the data buffer, *size* of the request, the *opcode* (read/write), and the *type* (asynchronous/synchronous). Depending on the *type*, the I/O request enters the kernel either through the asynchronous I/O (*AIO*) or the virtual filesys-

tem (*VFS*) layer of the kernel, as shown in Figure 2.1. Before we delve into the I/O path, we will briefly explain the data structures used by the kernel to represent an I/O at different stages. **NOTE: All the source code listings/file names shown in the upcoming sections refer to the Linux kernel v4.8.4.**

2.1.1.1 The iov_iter Interface

A user application submits a buffer representing user data to the kernel through a system call. The user buffer is either passed as a single buffer or as a vector of buffers represented by `struct iovec` (as defined by the POSIX standard). In both the cases, the kernel converts them into a kernel-equivalent `struct iovec`. The kernel employs an iterator, represented by `struct iov_iter`, to make buffer processing simpler and less error-prone [22]. Figure 2.2 shows the important members of these data structures. The `iov_iter` can also be defined on other kernel data structures like `struct bio_vec` or `struct kvec`. The `type` field determines the type of the iterator. Other members of `iov_iter` include: `iov_offset`, which points to the offset of the data in the first `iovec`; `count`, which refers to the total size of the data buffer; and `nr_segs`, which holds the total number of `struct iovec` data structures passed from the user application.

2.1.1.2 Block, Sector and Segment

The kernel uses the following abstractions to describe the location of data on block devices. First, a *sector* is the smallest addressable unit on a block device. In most cases, the size of a sector is 512 bytes, and it is never possible to issue a transfer less than a sector size. Second, a *block* represents the basic unit of addressing for the VFS and filesystem layers in the kernel. A block typically contains one or more sectors, and the size of a block is never more than the page size. When the kernel accesses file data on a block device, it issues accesses in *block* units that correspond to one or more adjacent *sectors* on the device. Finally, a *segment* is a region of memory that includes chunks of data that belong to nearby *sectors* on the disk.

Historically, a disk block was represented by a `struct buffer_head` in host memory and was also considered to be a unit of block I/O. Since `buffer_heads` could only describe a single page in memory, large block I/O operations had to be broken into several `buffer_head` units. This resulted in increased memory overhead as `buffer_heads` were

large and led to the introduction of `struct bio` data structure, the basic unit of block I/O operations in the Linux kernel today.

The `struct bio` treats I/O operations as a list of *segments* and also allows the kernel to represent a block buffer with multiple locations scattered in memory (scatter-gather I/O). Figure 2.3 shows the fields of `struct bio`. The important members are `bi_io_vec` and `bi_vcnt`. The `bi_io_vec` field represents an array of `struct bio_vec` structures, and `bi_vcnt` is the total number of such elements. Each `bio_vec` structure represents a single segment, described by the `<bv_page, bv_len, bv_offset>` tuple. An array of such `bio_vec` structures point to multiple pages in memory, thereby allowing a single buffer to be scattered anywhere, as shown in Figure 2.4.

Not all I/O requests from the application result in a disk access because of data caching performed by the kernel in the *page cache*. When the kernel issues an I/O to a block device, it first obtains the block number corresponding to the data by contacting the mapping layer. At this point, the kernel constructs a `struct bio`, populates it with the mapping information and user data from the `struct iovec`, and dispatches to the block layer through the `submit_bio` interface, as shown in Figure 2.1.

2.1.1.3 Request and Request queue

Within the block layer, the kernel maintains two data structures: `struct request` and `struct request_queue`. A `struct request` represents an I/O request in the block layer. It embeds a list of `bios` to support multiple block operations as shown in Figure 2.5. On the other hand, a `struct request_queue` represents the queue into which these `requests` are buffered.

Each `request` has a list of `bios` described by the members `bio` and `bio_tail`, respectively. The `queuelist` member points to the `request_queue` (implemented as a doubly linked list) to queue the `requests`. The `__sector` and `__data_len` specify the sector where the data transfer starts and the length of the transfer, respectively. When the `requests` reach the block device driver, `nr_phys_segments` is used to identify the total number of segments required to perform the scatter-gather DMA operation. In the next section, we will talk about the `request_queue` in detail.

2.2 Conventional Request Queue Interface

Every block device maintains a `request_queue` to queue pending I/O requests. The conventional block layer offered a single `request_queue` for each block device. Although it catered to the needs of rotational media, it proved to be a bottleneck for modern SSDs, which are capable of handling more than a million Input/output operations per second (IOPS). The block layer had to synchronize shared accesses to the `request_queue` using a lock, whenever I/O `requests` were inserted or removed, during optimizations like merging and scheduling, and also while remote memory accesses across CPU cores. The effect of lock contention was severe on NUMA-factor machines, thereby limiting the submission rate of requests to `request_queue`. These reasons led to a complete redesign of the block layer and the introduction of the Multi-Queue block layer [6].

2.3 Multi-Queue Block Layer

The Multi-Queue (MQ) block layer introduces two levels of queues, *software staging* and *hardware dispatch* queues, to address the scalability issues in the conventional block layer and also to exploit parallelism in software. The software staging queues are configured based on the number of CPU cores in the system, and aim to reduce the lock contention with a single `request_queue`. The hardware dispatch queues provide a second level of buffering to prevent device buffer overflow problems. The software and hardware staging queues together make the `request_queue` interface of the MQ block layer. In the next section, we will describe the software interface of the MQ block layer.

2.3.1 MQ Block Layer Interface

To use the services of the MQ block layer, a block device driver has to register itself with the MQ block layer. Figure 2.6 depicts the details of the interaction between a block driver and the MQ block layer. The following events happen during the registration process:

1. The driver invokes `blk_mq_alloc_tag_set()` to register itself with the MQ block layer. The function takes a pointer to a `blk_mq_tag_set` structure that contains a set of function pointers, number of hardware queues supported by the device, depth of each queue, etc.

2. The function pointers defines the interface between the driver and the MQ block

layer. The critical interfaces are as follows:

(a) `queue_rq()`: pass I/O `requests` to the device.

(b) `map_queue()`: map the hardware queues to the software queues.

(c) `complete()`: complete outstanding I/O requests.

(d) `poll()`: poll for completion events from the device (alternative for interrupts).

(e) Other interfaces like `init_hctx_fn()` and `exit_hctx_fn()` are used to initialize/tear down matching hardware structures in the driver.

3. To initialize a `request_queue` for a block device, `blk_mq_alloc_tag_set()` invokes the `blk_mq_init_queue()` function. This function creates software and hardware staging queues in the block layer and maps them to actual device queues allocated by the device driver.

2.4 Life of a Block I/O

Up to this point, we have explained the required background information necessary to understand the rest of the sections covered in this thesis. In this section, we will describe the I/O submission and completion paths through the direct I/O and MQ block layer of the Linux kernel. We will use Flexible I/O generator (*fio*) [21] in the example that follows, to generate I/O load, so that the reader can understand how to create I/O traffic from user-level. The job description file for the analysis is shown in Figure 2.7.

We perform direct device access (`direct = 1`) against a null block device represented by `filename=/dev/nullb0`. We use `libaio` as the `ioengine` to overlap I/O submissions. Other parameters include: `bs=512`, the size of the I/O we submit; `iodepth=8`, the maximum I/O units in flight; and `iodepth_batch=8`, the number of I/O units that can be batched in a single system call to the kernel. It is recommended to read the below sections while also referring to the Linux kernel source v4.8.4.

2.4.1 Submission Path

Figure 2.8 depicts the block I/O submission path through the direct I/O layer of the Linux kernel.

- *fio* issues I/O requests to the kernel using the `io_submit` system call. It queues an array of I/O requests represented by `struct iocb`, where each `iocb` holds the user data buffer, size, and the opcode (read/write) of the operation.
- The `io_submit` system call invokes `aio_run_iocb` to translate the `iocb` into `iov_iter`, the Linux kernel's internal abstraction (refer to Section 2.1.1.1). The function also associates a callback for every `iocb` to be invoked during the completion path. The callback function, `aio_complete`, is stored in the `ki_complete` member of `iocb` .
- Depending on the type of operation (read), `aio_run_iocb` invokes the corresponding file operations callback for the target of the I/O. Recall that the file on which our operations will be performed is a block device file, `/dev/nullb0`. Its file operations are defined in `fs/block-dev.c`, including `generic_file_read_iter` that handles read.
- `generic_file_read_iter` invokes `blkdev_direct_IO`, which is the callback for direct I/O operations.
- The main workhorse within `blkdev_direct_IO` is `do_blockdev_direct_IO`. It maintains two data structures, `struct dio` and `struct dio_submit`, to capture the state of direct I/O operations.
- The `do_blockdev_direct_IO` function first allocates a `dio` and stores the `iocb` to retrieve it during the completion path. It then populates `dio_submit` with block-related information and invokes `do_direct_IO`.
- This function obtains the pages (`struct page`) backing the user buffer by calling `get_user_pages_fast`, obtains the block number of the corresponding page, and invokes `submit_page_section`. Recall that the user buffer is passed through the `iocb`. Also note that the kernel avoids copying the user buffer into kernel memory.
- The final work of `do_blockdev_direct_IO` is to submit the I/O to the block layer. Before that, it constructs the `bio` descriptor (recall that `bio` is the basic unit of block I/O in the kernel) by invoking `dio_new_bio`, populates `bio` with information from `dio_submit`, stores the `dio` pointer in `bi_private` member for handling completions, and finally transfers it to the block layer via a call to `submit_bio`.

- For every `bio` allocated, a callback is set in the `bi_end_io` member of `bio` to point to either `dio_bio_end_aio` or `dio_bio_end_io` depending on the mode of submission. This callback will be invoked when the device posts a completion event (refer to Section 2.4.2). In our case, `bi_end_io` will be set to `dio_bio_end_aio` since we submitted requests asynchronously.

- The `submit_bio` function translates the `bios` to `requests` and queues them into the per-process plug (represented by `struct blk_plug`) before transferring them to the device driver.

- The concept of plugging allows the I/O submitting process to accumulate a set of requests to better utilize the bandwidth of the hardware and also to allow merging of sequential requests into a single large request.

- When a specific number of requests have been accumulated, the plug is flushed and the requests are sent to the device driver for setting up DMA operations. If not, `submit_bio` returns to where the I/O originated from, i.e., `do_io_submit`. This function calls `blk_finish_plug` to flush the queued requests to the device driver.

- The reason behind flushing the plug at this point is because we are about to return to user space after the system call and by the very definition of direct I/O, the I/O must have been submitted to the device.

- The `blk_finish_plug` invokes `blk_mq_flush_plug_list` to flush the I/O requests to the software staging queues of the MQ block layer.

- The I/O requests from the software staging queues are emptied into a specific hardware dispatch queue by `__blk_mq_run_hw_queue`.

- Finally, the `__blk_mq_run_hw_queue` flushes the hardware dispatch queue to the device driver through the `q->mq_ops->queue_rq` interface of the driver (refer to Section 2.3.1).

  Since we used asynchronous mode of submission (`ioengine=libaio`), the submission path returns to the user space without waiting for I/O completions. The user

application can either poll or invoke a system call to obtain the I/O completions. This will be the main focus of understanding in the next section.

2.4.2 Completion Path

For every I/O request submitted to the device, a completion is posted by the device to indicate that the I/O request has been processed. The block device driver forwards the I/O completions to the block layer to signal the status of an I/O request to the application process. For the sake of understanding, we analyze the completion path with device interrupts. Also note that the completion path will unwrap each of the data structures we encountered in the submission path in the reverse order (from `request` to `bio` to `dio` to `iocb`).

- When the device triggers an interrupt, the kernel invokes the corresponding interrupt handler registered for the interrupt. To forward the completion event to the upper layer, the driver identifies the `request` for which the completion has arrived.

- The driver uses a unique identifier called the tag in the completion message to obtain the matching `request`. The MQ block layer associates every `request` with a tag and provides a helper function, `blk_mq_tag_to_rq`, to obtain the corresponding `request` for a tag. Note that this is the first stage of unwrapping.

- The driver invokes `blk_mq_end_request` to forward the I/O `request` to the MQ block layer. The `struct request` has all the information for the MQ block layer to complete the I/O request.

- `blk_mq_end_request` invokes `blk_update_request` to perform accounting on the I/O request and calls `req_bio_endio`. Since a single `request` can have a list of `bios` embedded in it, `req_bio_endio` iterates over the `bios` by calling `bio_endio`. Note that this is the second stage of unwrap where we obtain the `bios` from the `request`.

- Recall that for every `bio` submitted to the block layer, the callback pointer that was set (`dio_bio_end_aio`) during the I/O submission path (refer to Section 2.4.1) will be invoked to forward the completion to the application process.

- `dio_bio_end_aio` invokes `dio_bio_complete` to retrieve the `struct dio` that was stored in the `bi_private` member of `bio` during the I/O submission. This marks another stage of unwrap where we obtain the `dio` from `struct bio`.

- Up to this point, we have explained the path from the device interrupt until the callback to application (*fio*, in our example) and also the how the different data structures are related to one another.

- The final stage of unwrap happens inside `dio_complete` as it obtains the `iocb` pointer stored in the `dio` and invokes the callback to the application process. Recall that the callback function `aio_complete` was stored in the `ki_complete` member of `iocb`.

- The `aio_complete` call notifies an event in the ring buffer maintained in the AIO layer to indicate the completion of an `iocb`. The application process can either invoke the `io_getevents` system call or poll the ring buffer directly from user space to read the completion events. Figures 2.9 and 2.10 depict the completion path in detail.

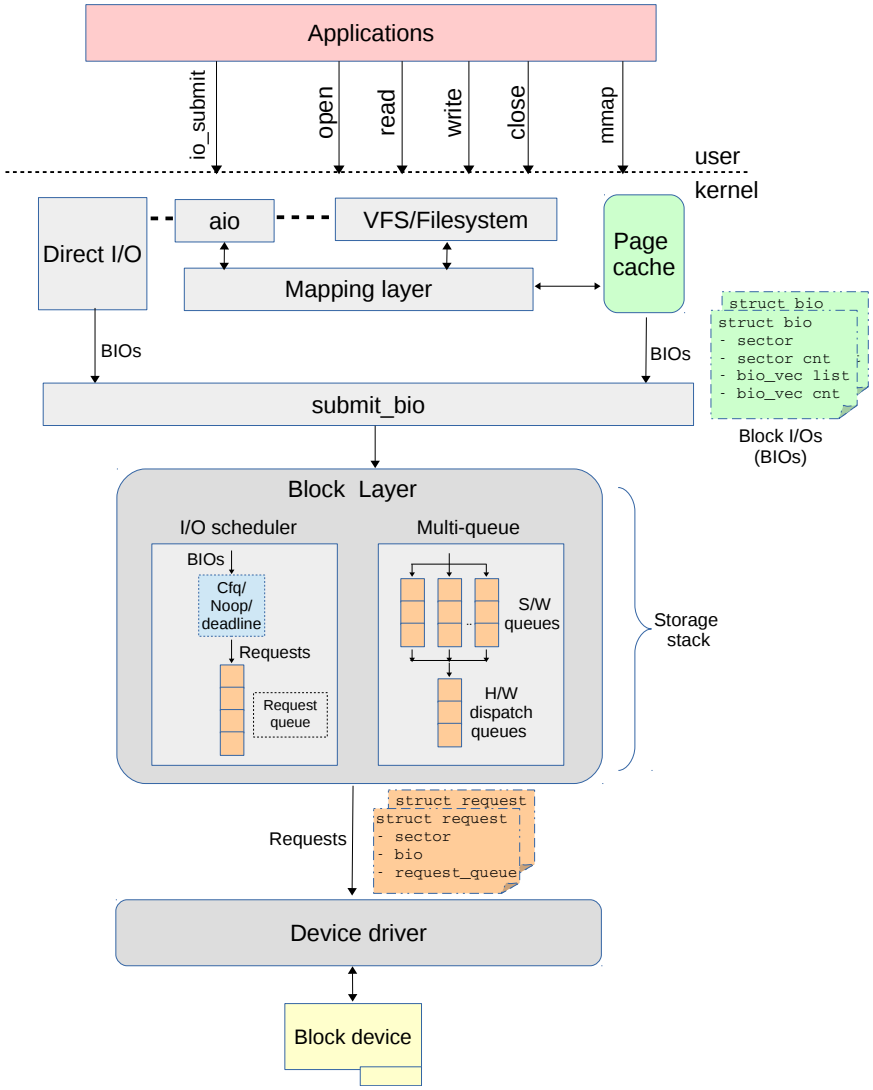

Figure 2.1. Overview of the Linux block layer

```
/* <uapi/linux/uio.h> */
struct iovec {
    void __user *iov_base;
    __kernel_size_t iov_len;
};

/* <linux/uio.h> */
struct iov_iter {
    int type;
    size_t iov_offset;
    size_t count;
        union {
        const struct iovec *iov;
        const struct kvec *kvec;
        const struct bio_vec *bvec;
    };
    unsigned long nr_segs;
};
```

Figure 2.2. `struct iov_iter` and `struct iovec` data structures

```
/* <include/linux/blk_types.h> */
struct bio {
    struct bio *bi_next; /* request queue link */
    struct block_device *bi_bdev;
...
    struct bvec_iter bi_iter;
    unsigned short bi_vcnt; /* how many bio_vec's */
    struct bio_vec *bi_io_vec; /* the actual vec list */
...
    /* Simplified structure, other members not shown */
};

/* <include/linux/bvec.h>*/
struct bio_vec {
    struct page *bv_page;
    unsigned int bv_len;
    unsigned int bv_offset;
};

struct bvec_iter {
    sector_t bi_sector; /* device address in 512 byte sectors */
    unsigned int bi_size; /* residual I/O count */
    unsigned int bi_idx; /* current index into bvl_vec */
    unsigned int bi_bvec_done; /* number of bytes completed in current bvec */
};
```

Figure 2.3. Simplified version of `struct bio` and `struct bio_vec` data structures

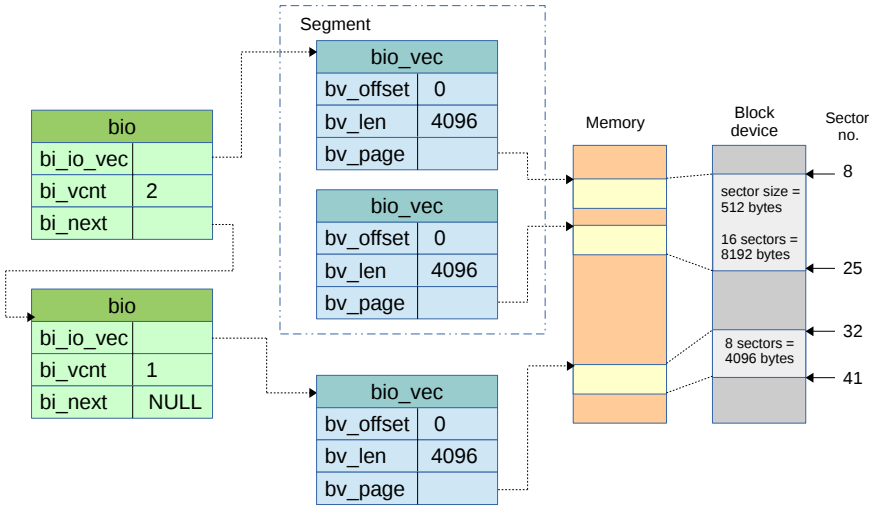

Figure 2.4. Representation of `struct bio` and its members

```
struct request {
    struct list_head queuelist;
    struct request_queue *q;
...
    /* the following two fields are internal, NEVER access directly */
    unsigned int __data_len; /* total data len */
    sector_t __sector;  /* sector cursor */
    struct bio *bio;
    struct bio *biotail;
...
    /* Number of scatter-gather DMA addr+len pairs after
     * physical address coalescing is performed.
     */
    unsigned short nr_phys_segments;
...
};
```

Figure 2.5. Simplified version of `struct request` data structure

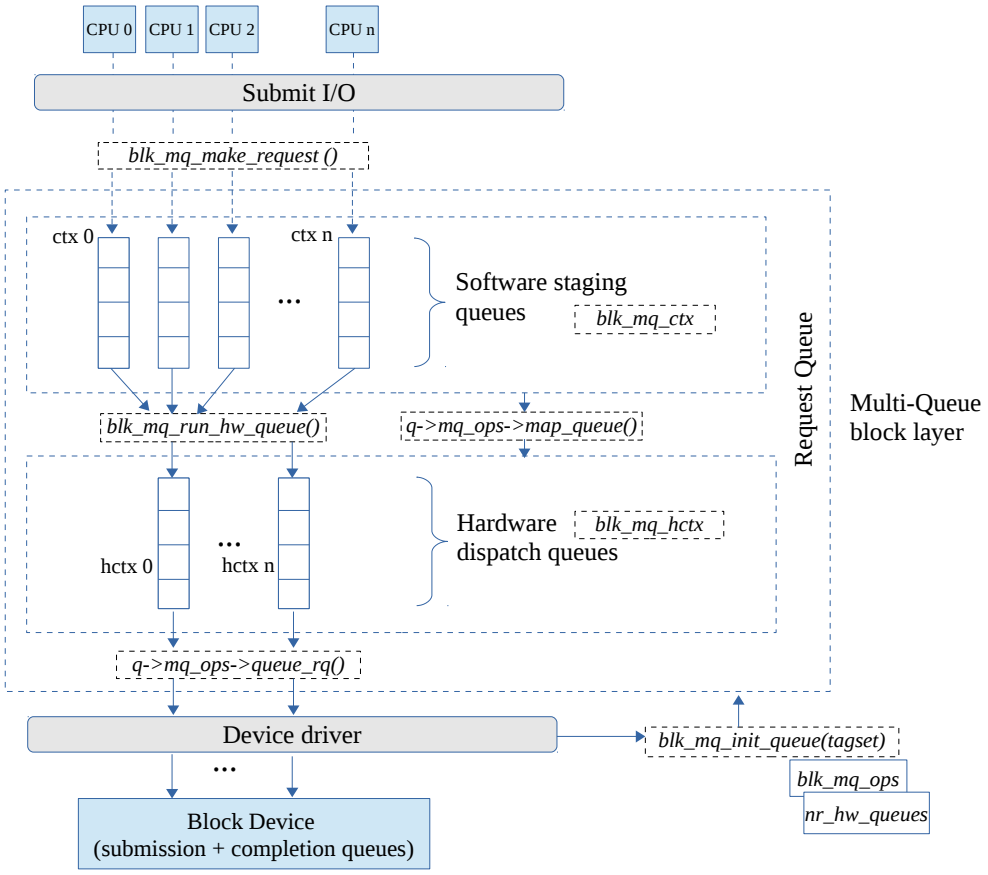

Figure 2.6. Architecture of the MQ block layer

```
[global]
direct=1
filename=/dev/nullb0
numjobs=1
iodepth=1
ioengine=libaio
rw=read
mem=mmapshared

; 512 bytes read
[512-bs-iod=8]
bs=512
iodepth=8
iodepth_batch=8
size=1024M
```

Figure 2.7. Fio job configuration

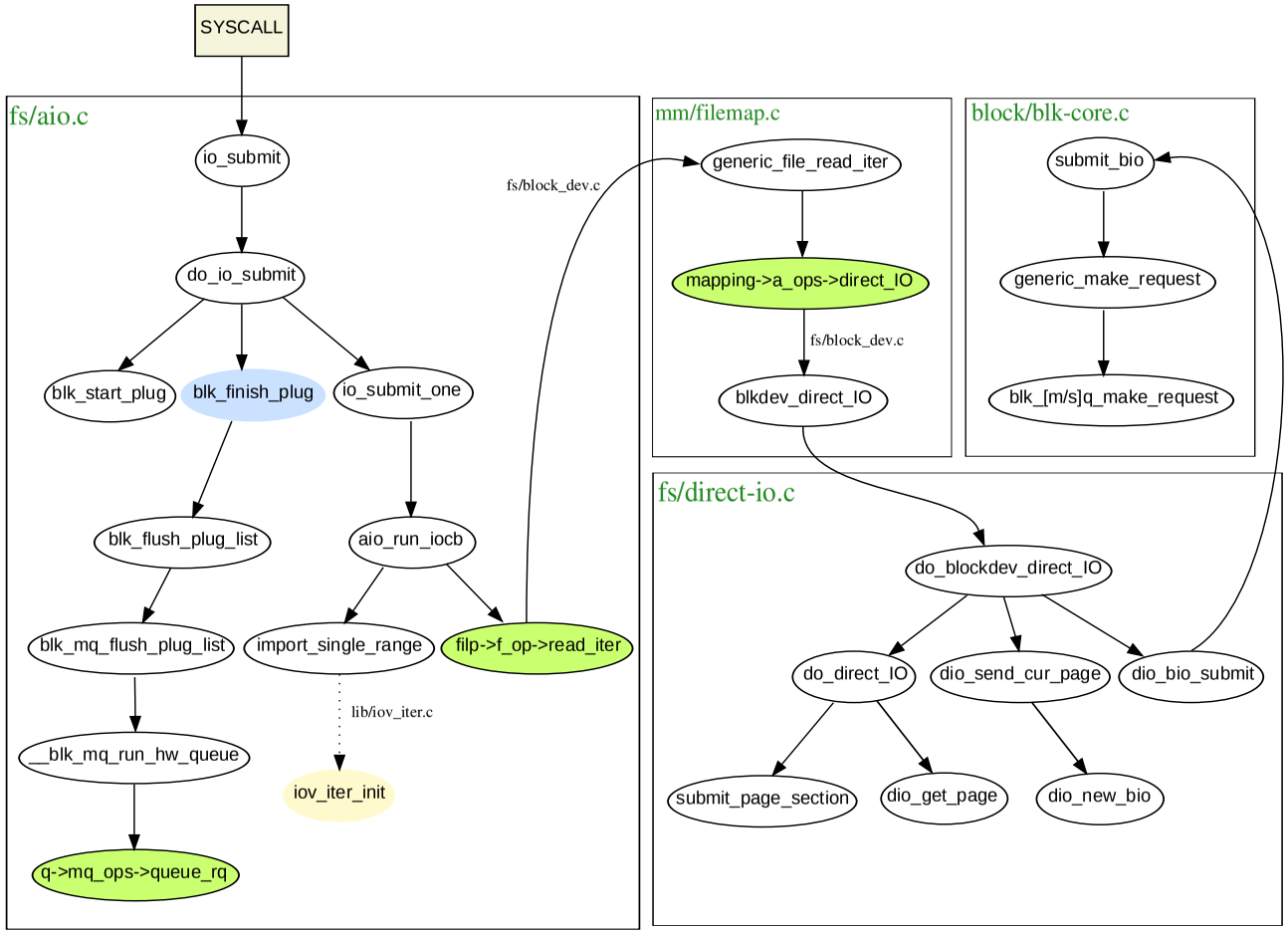

Figure 2.8. Block I/O submission path through the direct I/O layer

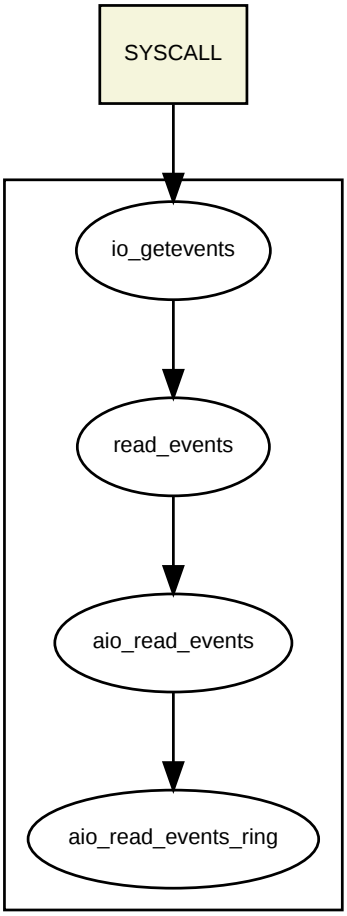

Figure 2.9. Reading completions from user space through the io_getevents system call

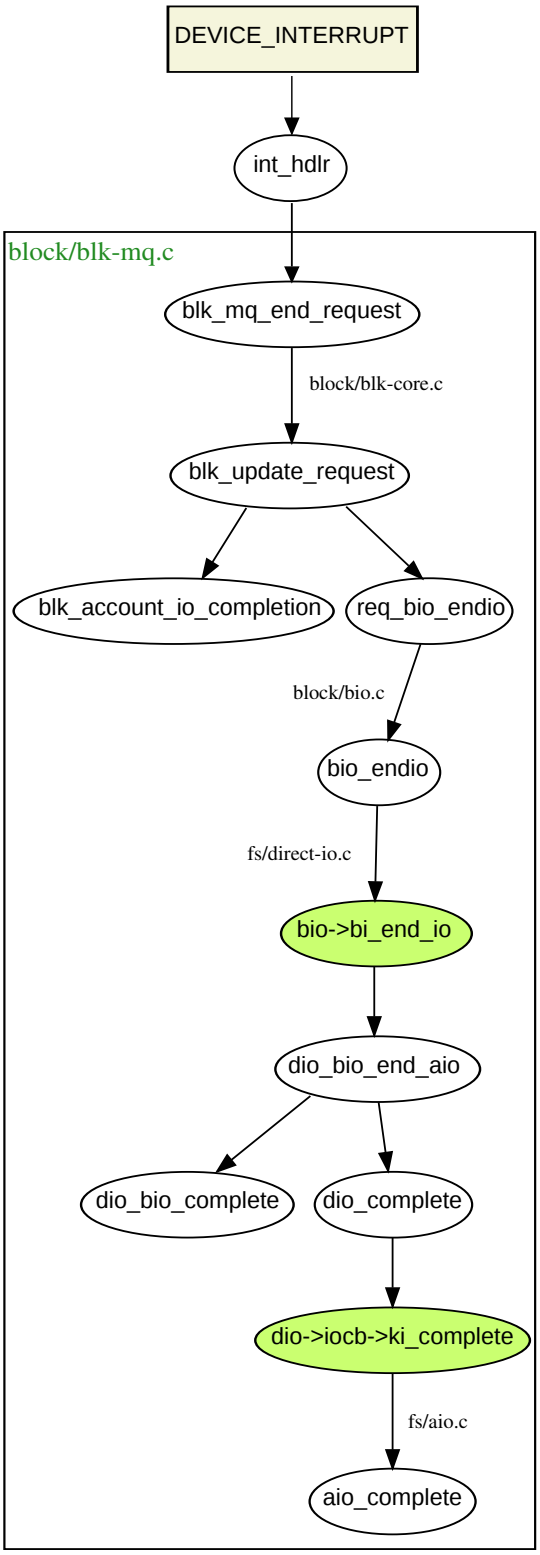

Figure 2.10. I/O completion path from device

CHAPTER 3

DRIVER ISOLATION

This chapter introduces our principles of isolation and describes the different techniques used to isolate the null block driver.

3.1 Introducing Isolation

We develop a general framework, Lightweight Capability Domains (LCDs), for decomposing a fully featured commodity operating system kernel. We extend a commodity kernel like Linux with a general capability-based microkernel interface. This interface enables to execute isolated subsystems along with the rest of the kernel, thereby providing a path for incremental decomposition and an environment for decomposing subsystems on the fly. We also develop a set of general techniques to decompose common code patterns like function pointers, shared data structures, etc.

Since one of our main objectives is to avoid running an entire OS inside an LCD, we develop lightweight interposition layers to handle the dependency of the isolated code. We use an Interface Definition Language (IDL) to capture common interaction patterns in code and a compiler to generate source-level compatible code with the non-decomposed subsystem. The generated code provides an illusion to the isolated code by replicating state and synchronizing through message passing. To make decomposed subsystems faster on modern hardware, we develop a fast cross-core asynchronous communication mechanism that exploits the behavior of cache-coherence protocol. Finally, to efficiently use the CPU core during blocking cross-domain calls, we introduce a lightweight execution model that supports composable asynchronous threads. These threads overlap blocking invocations with computation by performing a lightweight context switch to another thread. In the following sections, we will briefly go through the techniques discussed in [20] and show how a device driver like null block can be isolated using them. Figure 3.1 depicts the overall architecture of our system.

3.1.1 Lightweight Capability Domains

We make several careful design choices to simplify development. We rely on hardware-assisted virtualization (VT-x) to isolate code inside an LCD. Even though context switching into a VT-x domain is expensive and slower compared to traditional address spaces, we still rely on VT-x to isolate code. VT-x is relatively easier to program, offers a convenient interface to handle device passthrough, and, more importantly, provides separate address space to execute isolated code. We do not allow LCDs to perform context switches often.

To collectively manage the LCDs, we install a small microkernel inside the Linux kernel. Similar to seL4 [17], the LCD microkernel implements a capability-mediated interface (VMCALL interface) that explicitly controls all communication across isolated subsystems. The microkernel uses capabilities to explicitly track the resources like physical memory, IPC channels, etc., accessed by an LCD. It also maintains a *capability space* or *CSpace* for an LCD to store every object owned by it along with its access rights. The *(object, access rights)* pair is referred to as *capabilities*. Although the LCD microkernel is similar to the KVM virtual machine monitor in the Linux kernel, it provides a more general interface for development of semantically rich decomposed subsystems, other than merely providing low-level CPU virtualization.

Though LCDs run independently from the rest of the kernel, they still require a kernel environment to handle common primitives like memory allocation, synchronization, string manipulation, etc. We provide a small library, `liblcd`, within the LCD to provide a lightweight kernel environment for the isolated code. We also insist that the nonisolated code use a similar interface to interact with an LCD. For this purpose, we provide a symmetrical library, `kliblcd`, for the nonisolated code. This interface allows the nonisolated code to communicate with the microkernel through the capability-mediated interface rather than directly accessing the internal data structures. But this cannot be strictly enforced because the nonisolated kernel code has full control over the entire system. However, the threads in the nonisolated world should initialize a unique context by calling `lcd_enter` to interact with the LCDs. After that, the microkernel sets up a runtime, then initializes a per thread CSpace and resources for the nonisolated thread to communicate with the LCDs.

3.1.2 Interprocess Communication

We support two Interprocess Communication (IPC) mechanisms for LCDs to communicate amongst themselves and with the nonisolated part of the system.

- **Synchronous IPC**: The synchronous IPC is capability-mediated, provided by the LCD microkernel for the LCDs, to request resources, establish regions of shared memory, and to perform cross-domain calls. This mechanism requires both the send and receiver to synchronize on the endpoint so that the microkernel can transfer the contents from the sender to the receiver. But synchronous IPC is slow because it requires a context switch into the microkernel. So we use them only in control-plane operations and instead use the fast asynchronous IPC for fast data-plane activities.

- **Fast, Asynchronous IPC**: We pin isolated subsystems on different CPU cores and use a fast cross-core communication mechanism to trigger *call* and *reply* invocations between them. Motivated by Barrelfish [4], our design works by establishing a region of shared memory between the sender and the receiver. Each asynchronous IPC channel consists of two ring buffers: one for outgoing *call* messages and the other for incoming *reply* messages. Since the IPC mechanism leverages the cache-coherence protocol of the processor, the performance of cross-core communication depends on the latency of the protocol. To achieve the lowest possible communication overhead, we optimize the number of cache-coherence transactions. We ensure that every message is cache-line aligned (64 bytes) and avoid shared producer-consumer pointers, as they add two extra transitions per message. Instead, we depend on an explicit state flag that signals available message and free slots. To send a message, the sender first obtains a free slot from the ring buffer, prepares the message, and sets a flag to indicate that the message is ready for receiving. The receiver, on another CPU core, listens for a message by polling on the same slot until it sees a ready status of the flag.

3.1.3 Interface Definition Language

We rely on an IDL to automatically generate all the inter-domain glue code and dispatch loops. The IDL is parsed by a compiler [37] to generate *caller* and *callee* glue code, which allows transparent invocation of functions and synchronization of data structures

across domain boundaries. The *glue* layer translates the function invocations into Remote Procedure Calls (RPC) using the IPC mechanism as its communication medium. It marshals the parameters of function calls, translates them into IPC, and unmarshals return values if any. All these events happen behind the scenes and provide an illusion to the isolated code that it still invokes the real function.

We illustrate IDL with an example from the null block driver as shown in Figure 3.2. The kernel function `blk_mq_alloc_tag_set` registers the driver with the MQ block layer by passing a pointer to `blk_mq_tag_set` structure as its argument. In IDL, we use the *rpc* keyword to represent functions. This keyword instructs the compiler to generate two functions: a caller and a callee function. The caller has the same function signature as the original kernel function, whereas the callee has the required information to retrieve data from an IPC message. The IDL also defines the IPC mechanism to be used through the means of *channel* keyword (not shown in Figure 3.2). Since synchronous communication is relatively slow due to context switch overhead, we use the fast asynchronous IPC mechanism.

One of the fundamental properties of our design is that LCDs do not share any state with the nonisolated part or among other LCDs by default. Instead, we allow LCDs to maintain its own private hierarchy of data structures and synchronize them during function invocations. In our example from Figure 3.2, the function `blk_mq_alloc_tag_set` takes a pointer to the `blk_mq_tag_set` structure that describes the configuration parameters supported by the null block device. We support transparent synchronization of data structure copies across domains through *projections*. A projection defines how a data structure is projected into another domain by capturing the fields of a data structure that will be marshaled and unmarshaled during function calls. The projection `blk_mq_tag_set` lists the fields that will be used by the nonisolated kernel code to register the driver with the MQ block layer. The IDL supports explicit *[in]* and *[out]* directional attributes to specify the direction of flow, from the caller to callee or vice versa. The default direction is *[in]*, i.e., all the fields are copied from the caller to the callee side. The compiler infers it from the *[alloc(callee)]* qualifier. The *alloc* specifier instructs the IDL compiler to allocate a copy of the projected type `blk_mq_tag_set`. As the data structure already exists on the caller side, the *callee* keyword specifies to perform the allocation on the callee side. In most cases, the

LCDs refer the same data structure multiple times. In our example, the `blk_mq_tag_set` is also accessed during a call to `blk_mq_init_queue`. We provide a mechanism of *remote reference* to refer to a specific object during function invocation. The same remote reference also identifies the object in its CSpace.

The IDL also implements support for exporting function pointers. In most parts of the kernel, function pointers are used to support dynamic registration. In our example, `blk_mq_alloc_tag_set` also registers a set of function pointers through the `blk_mq_ops` data structure. For every function pointer, the IDL compiler generates caller and callee code like normal functions. To implement cross-domain function pointers, while still preserving the original semantics of function signatures, we implement a concept of *hidden arguments* [20]. For every function pointer, the IDL compiler generates a trampoline function in the caller's executable (nonisolated part) address space. The hidden arguments specify the missing arguments require to perform a cross-domain call. But the caller invokes the trampoline like any other function pointer.

Figure 3.1 shows the different components of isolation and Figure 3.3 highlights the caller and callee stub code in interaction.

3.1.4 Kernel Modules

We implement LCDs as kernel modules. By doing so, we reuse the linking and loading functionality provided by the Linux kernel. By mapping the kernel module to the same guest virtual address as that of the host, we avoid relocating the symbols in the module. In our design, we have two kernel modules, *lcd* and *klcd*, that fulfill the functionality of the isolated code.

We compile the *lcd* kernel module with the isolated driver code, the `liblcd` library, and the glue code. We also provide another module, *boot*, that loads the *lcd* module inside an VT-x container.

We also have another module, *klcd*, that runs along with the nonisolated part of the kernel. This module is mainly responsible for servicing the RPC requests from the *lcd* and also to translate kernel/user process's requests into an RPC to the *lcd*.

In both modules, the glue layer is responsible for intercepting the calls and translating them into RPCs. The glue code comprises of two parts: a set of *caller* functions statically

defined to translate the function calls into RPC, and a *callee dispatch loop* that listens for the incoming RPC requests. The dispatch loop is implemented using AC threads, which we will discuss in the next section.

Figure 3.3 shows the interaction between *lcd* and *klcd* module to service a request from the block layer.

3.1.5 AC Threads and ASYNC Runtime

In LCDs, we replace function calls with cross-domain calls. Even though we optimize the cross-domain invocations for low latency, in most cases, it is still wasteful for the caller to wait for a reply from the callee. We have seen that cross-domain invocations take roughly around 380 to 1500 cycles. While it is long enough to be wasteful, it is still short to yield the CPU through a context switch.

We implement a lightweight asynchronous thread language (AC) and a runtime, borrowed from Barrelfish research OS [16], with extensions for asynchronous IPC [32]. The AC provides a lightweight asynchronous functionality to C language using macros and GCC extensions like nested functions. At the core of the AC language, there are two macros: `DO_FINISH` and `ASYNC`, which enables us to create a runtime and spawn lightweight threads. All invocations of `ASYNC` must happen within a `DO_FINISH` block, and it guarantees that all asynchronous work will finish by the end of the block.

These threads are lightweight because a thread is only a *stack frame* and an instruction pointer. In a scenario where there are more than one asynchronous threads, when one of the blocks, the runtime checks to see if there are other any other pending threads. If so, the runtime switches control to the unfinished thread and services it. This constitutes to the asynchronous behavior.

The AC language also defines functions that can be used to yield an execution. They are `THCYield`, `THCYieldTo`, etc. The main purpose of yielding is to suspend the current execution path and pass control to another. We integrate these functions with the asynchronous IPC mechanism [32] to improve the responsiveness of IPC to handle multiple requests. We present an example of how the runtime is used to implement a dispatch loop in Section 3.3.1.

3.2 Null Block Driver Architecture

Up to this point, we have discussed the different pieces of LCD architecture. In this section, we put the various components of the LCD architecture into practice by isolating the null block driver.

We chose to isolate the null block driver to understand the pure software overheads of isolation. The null block driver does not interact with an actual storage medium, hence the *null* name. Instead, it emulates the behavior of the fastest possible block device. The emulated device allows us to study overheads of isolation without any artificial limits of existing physical storage mediums. Moreover, these findings can be directly used to decompose and implement an isolated NVMe subsystem.

In the next section, we will discuss about the various configurations of the driver and its interface with the kernel. Figure 3.4 depicts the architecture of the null block driver.

3.2.1 Configurations

We configure the interface and behavior of the driver through the module parameters exported by the driver. We restrict ourselves to a single hardware queue by setting `submit_queues = 1` because the current LCD architecture is single threaded. Although we have the support to handle multiple hardware queues using a single thread in a round-robin fashion, we leave that to future work.

We chose to configure the driver in such a way that it exposes the software overheads of the MQ block layer. For instance, we do not configure the driver to emulate device interrupts to signal completion. Instead, we signal completions immediately after processing the I/O request to eliminate any device latency.

3.2.2 Interfaces

We configure the driver to interact with the MQ block layer (`queue_mode = 2`) because of the several benefits discussed in the previous chapter. The line numbers mentioned in below section refers to the null block driver source in Linux kernel v4.8.4 [13].

3.2.2.1 Initialization and Registration

We will describe the functions the driver uses during initialization to interact with the kernel. We represent the line numbers of the functions within square braces.

- [805] `register_blkdev`: The driver registers a new block device (`/dev/nullb0`) with the kernel.

- [820] `null_add_dev`: To interact with the MQ block layer, the driver does the following:

    1. [674] `blk_mq_alloc_tag_set`: Registers the driver with the MQ block layer.

    2. [678] `blk_mq_init_queue`: Allocates and initializes a `request_queue` to exchange I/O requests between the MQ block layer and the driver.

- [714] `blk_queue_logical_block_size`: Indicate the lowest possible block size (512 bytes) the device can address.

- [715] `blk_queue_physical_block_size`: Indicate the lowest possible sector size the device can operate without reverting to read-modify-write operations.

- [727] `alloc_disk_node`: To allocate a disk node, `struct gendisk`, representing the emulated disk.

- [733] `set_capacity`: To set the capacity of the disk.

- [743] `add_disk`: The driver populates the `gendisk` with a set of function pointers and registers with the kernel.

3.2.2.2 I/O Path

The driver is ready to exchange I/O requests from the user application once it initializes itself with the kernel successfully. The I/O path is defined by a set of function pointers registered by `blk_mq_alloc_tag_set` function. The following function pointers are exported by the driver:

- `queue_rq`: Set to the function `null_queue_rq`, defined at line `[354]`. The MQ block layer transfers I/O requests from the application process to the driver through this interface.

- `map_queue`: Set to `blk_mq_map_queue`, defined at line `[396]`. This function initializes the mapping between software and hardware queues in the MQ block layer.

- `init_hctx`: Set to `null_init_hctx`, defined at line `[396]`. It initializes a specific hardware queue in the driver.

In the next section, we will describe our approach to isolate the interfaces and handle the dependencies.

3.3 Isolated Null Block Driver

Our central objective is to run the unmodified null block driver inside an LCD. To achieve this, we first analyze the interactions between the driver and the kernel, then identify the functions and data structure dependencies. We develop IDL specifications of the block driver interface, which consists of 68 lines of IDL code. Our isolated system comprises three kernel modules, `lcd`, `klcd`, and the `boot`. The `klcd` module implements the glue code and dispatch loop for the nonisolated part of the kernel. The module starts a kernel thread to process IPC messages from the `lcd`. The `lcd` module loads the unmodified driver into a VT-x container that creates a new kernel thread to execute the driver inside the isolated domain. The `boot` module is responsible for basic initialization, loading, and unloading of `lcd` and `klcd` modules. Figure 3.5 depicts the overall architecture of the isolated null block driver.

3.3.1 Initialization

When the `lcd` and `klcd` modules boot, they initialize a *CSpace* for remote references and set up their own address spaces. The boot module initializes a synchronous IPC channel between them and invokes the `module_init` function of the driver. The driver continues to execute its initialization routine and invokes the first cross-domain call, `register_blkdev`. The glue code intercepts the call and transparently triggers an IPC message to the `klcd` using the synchronous IPC channel set up by the boot module. Meanwhile, it also allocates memory for the asynchronous IPC ring buffers, and initializes an exclusive asynchronous IPC channel between the `lcd` and the `klcd` modules. Once the asynchronous IPC buffers are initialized, the `klcd` enters into a *dispatch loop*, while the `lcd` continues to invoke other initialization routines.

We present a simplified version of the dispatch loop in Figure 3.6. The dispatch loop is implemented using the asynchronous runtime macros discussed earlier in this chapter

(refer to Section 3.1.5). The thc_ipc_poll_recv function listens for a message over the asynchronous IPC channel represented by async_chnl. Once the channel receives a message, it spawns an ASYNC thread to handle the message. When ASYNC blocks, the control reaches to the immediate line of the ASYNC macro and the dispatch loop continues to listen for other messages.

Once the dispatch loop is set up, the lcd continues to invoke its initialization routines as mentioned in the Section 3.2.2.1. On a successful completion of register_blkdev function, the driver invokes blk_mq_alloc_tag_set to register with the MQ block layer and blk_mq_init_queue to initialize a request_queue for the device. Successful completion of these functions prepares the I/O interface between the MQ block layer and the driver. We will discuss the isolation of I/O path in the next section.

3.3.2 I/O Path

We saw in Section 2.4.1 that the I/O requests submitted by an application process enter the device driver through the queue_rq interface. The MQ block layer invokes __blk_mq_run_hw_queue to flush the I/O requests from the hardware dispatch queues to the driver. Figure 3.7 shows the function flow through the MQ block layer into the driver and Figure 3.8 depicts the I/O request processing loop in the MQ block layer.

The isolated null block driver requires *three* cross-domains calls in the I/O path. First, the MQ block layer invokes the queue_rq function pointer, which points to null_queue_rq function inside the driver. The driver itself invokes two functions, blk_mq_start_request and blk_mq_end_request, which represent the other two cross-domain calls in the I/O path. The blk_mq_start_request function passes a pointer to request data structure back to the MQ block layer to inform that the request processing has begun and I/O is ready to be issued to the device. The MQ block layer associates a timer for this particular request to ensure that if the completion for that request does not arrive in time, it can either abort the I/O operation or re-queue the request again. On the other hand, the blk_mq_end_request function informs the block layer that the I/O request is completed by the device, and is now ready to remove any pending timers on the request and notify the application process that submitted the I/O request.

3.3.3 Isolation Infrastructure

On top of the existing LCD infrastructure, we introduce essential features that allow device drivers to communicate with application processes, share data buffers, and speed up I/O processing loops by introducing asynchrony in the nonisolated kernel code.

3.3.3.1 Access to User Applications

Applications operate on devices by invoking system calls like `open`, `close`, `ioctl`, etc. Device drivers expose a set of function pointers, called *file operations* in Linux, to allow these system calls on devices. The isolated null block driver registers function pointers for `open` and `close` during the call to `add_disk` function. We set up trampolines for these function pointers in the glue code of the `klcd` module to translate function calls into IPC messages.

In LCDs, an application thread in the nonisolated part has no information to access an isolated subsystem. When an application thread invokes the `open` call on the null block device (`/dev/nullb0`), we set up a per-thread context for this thread by calling the `lcd_enter` library function. The context includes a per-thread CSpace, buffers for synchronous IPC, and a state for the ASYNC runtime. We also initialize a dedicated asynchronous IPC channel for each application thread that intends to communicate with the isolated null block driver. We cannot allow the application thread to access the already existing asynchronous IPC channel between the `lcd` and `klcd` module because the ring buffers are lock-free queues dedicated for every producer-consumer pair.

3.3.3.2 Sharing Data Buffers

In contrast to the non-decomposed kernel, the isolated null block driver has no rights over the memory of an application process that issues block I/O. To avoid expensive page remapping operations from the user process' memory to the `lcd` module, we require the process to explicitly share a region of memory with the isolated null block driver. The application can allocate data buffers for block I/O from the shared region. In the current work, we implement this infrastructure inside the *fio* benchmarking application. Alternatively, we also propose that a dedicated library in the user space can implement `malloc` and `free` calls, managed by a simplified version of slab allocator [7] to provide this infrastructure. An application process can use the `LD_PRELOAD` trick to route the standard

memory allocation calls to our library. We will describe the memory sharing protocol implemented within the *fio* benchmarking application below:

- We instantiate a character device represented by `/dev/nullb_user` and register a set of `file_operations` to facilitate `open`, `mmap`, and `close` on the device. Refer to Figure 3.9.

- We analyze the memory allocation scheme in `fio` and notice that the `mem` or `iomemem` option can specify the type of memory allocation in the job file. For instance, we can instruct `fio` to allocate memory using huge pages in the system by specifying `mem=mmaphuge`.

- Similarly, we introduce another option, `mem=mmapdevmem`, to instruct `fio` to memory map device memory and use it for allocating I/O buffers. The name of the device is specified as `mem=mmapdevmem:/dev/nullb_user` in the job file.

- When `fio` invokes the `mmap` call to allocate memory for I/O buffers, it passes a size that is a function of the maximum allowed block size multiplied by the I/O depth for the job.

- We intercept the `mmap` call in the `klcd` module's glue code, allocate memory pages for the requested size, and map it into the address space of `fio`. At the same time, we also *volunteer* the pages into `lcd` module and map them in a dedicated region of address space within the `lcd` module.

- The data buffers are accessible within the `lcd`'s address space by an offset from the base address of the memory region. To identify the corresponding I/O buffer during the submission path, we calculate the relative offset of the data buffer on the nonisolated part and marshal the value to `lcd` module. The same offset identifies the buffer in `lcd` module's address space.

- Since the null block driver does not access the pages of data buffer, we ignore these fields in the isolated driver. In the case of a NVMe driver, these fields will aid in programming the IOMMU for DMA operations.

- Finally, when `fio` performs cleanup of its resources, the allocated pages for the I/O buffers are also released.

3.3.3.3 Introducing Asynchrony

We saw in Section 3.3.2 that the MQ block layer invokes the `queue_rq` function pointer in a tight loop to flush I/O requests to the driver (refer to Figure 3.8). We also noticed that the null block driver invokes `blk_mq_start_request` and `blk_mq_end_request` to process a single I/O request.

In LCDs, we replace these function calls by cross-domain IPC requests. Even though we aggressively optimize the IPC in most cases, it is wasteful to wait for a reply for every `queue_rq` invocation. Instead, to effectively utilize the CPU cycles and to meet the performance aspect of our goal, we introduce asynchrony in the kernel code. We implement a parallel loop using the `ASYNC` primitives described in Section 3.1.5. Figure 3.10 depicts the asynchronous I/O request processing loop in the MQ block layer.

By introducing asynchrony, we still preserve the original semantics of the code and do not re-implement the function in a message passing style. The `ASYNC` macro spawns a new lightweight thread for handling an IPC call. When the call blocks, the control exits out of the `ASYNC` block, and yields to the runtime (`DO_FINISH`) to dispatch pending requests in the `rq_list`.

While profiling the I/O path in the isolated null block driver, we noticed that a single `queue_rq` IPC request was consuming 5130 cycles. We will examine the timing in more detail in the next chapter. The main reason for high isolation overhead is because the `lcd` module is wasting CPU cycles while waiting for the responses of `blk_mq_start_request` and `blk_mq_end_request` functions. To lower the cost of IPC, we introduce two optimizations:

- **Weak return semantics:** We asynchronously trigger both `blk_mq_start_request` and `blk_mq_end_request` from the `lcd` module. Since the kernel implements both of these as `void` functions, they do not return any value to the caller. We follow weak return semantics, i.e., we do not wait for the IPC response of these functions in the `lcd` module. We argue that it does not hurt the semantics of the driver code because in a real scenario, the completions from any storage device arrive in an asyn-

chronous manner either via interrupts or by polling. So asynchronously dispatching `blk_mq_end_request` is acceptable. We also ensure that the `blk_mq_start_request` for a particular I/O `request` executes before `blk_mq_end_request` because of the very nature of the asynchronous IPC message buffers.

- **Optimizing asynchronous IPC:** When the request processing loop (refer to Figure 3.10) has only a single I/O request, introducing a runtime around the loop adds extra overhead in the IPC path. In such cases, we optimize the IPC routine to busy poll for a reply instead of yielding back to the runtime to check for additional messages.

Figure 3.11 depicts the flow of an I/O request from a user application to the isolated driver.

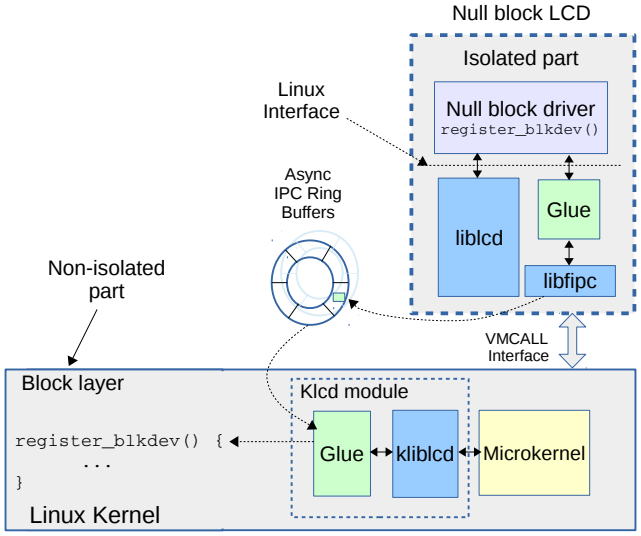

Figure 3.1. Key components of isolation architecture

```
/* Original kernel function */
int blk_mq_alloc_tag_set(struct blk_mq_tag_set *set);

/* Equivalent IDL representation */
rpc int blk_mq_alloc_tag_set(projection blk_mq_tag_set [alloc(callee)] *set);

/* Data structure representation */
projection <struct blk_mq_tag_set> blk_mq_tag_set {
    unsigned int nr_hw_queues;
    unsigned int queue_depth;
    ...
    projection blk_mq_ops [alloc(callee)] *ops;
}

/* Function pointer representation */
projection <struct blk_mq_ops> blk_mq_ops {
    rpc [alloc] int (*queue_rq_fn)(projection blk_mq_hw_ctx [alloc(caller)] *ctx,
        projection blk_mq_queue_data [alloc(caller)] *bd);
    ...
}
```

Figure 3.2. Snippet capturing IDL representation

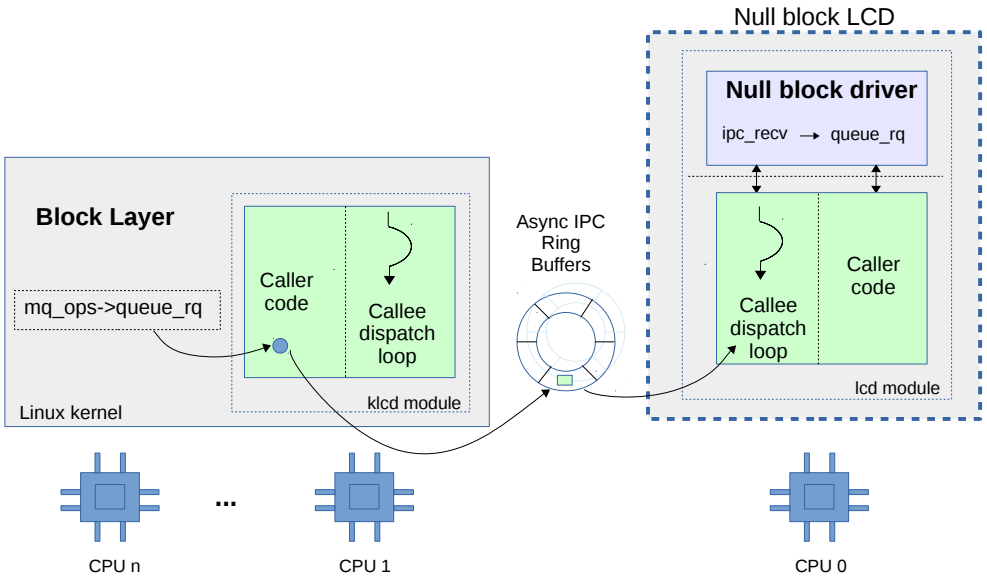

Figure 3.3. Caller and callee dispatch loop

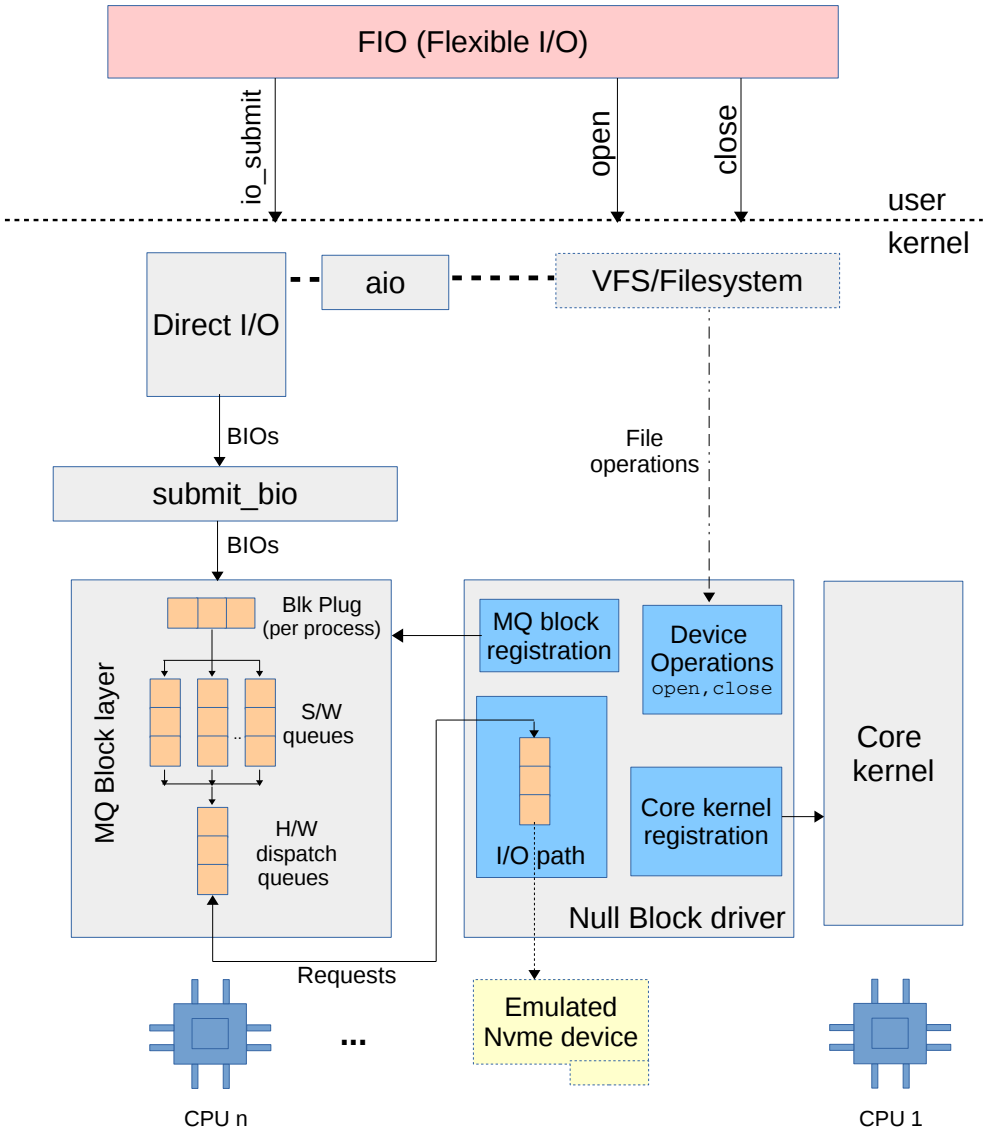

Figure 3.4. Different components of the nonisolated null block driver

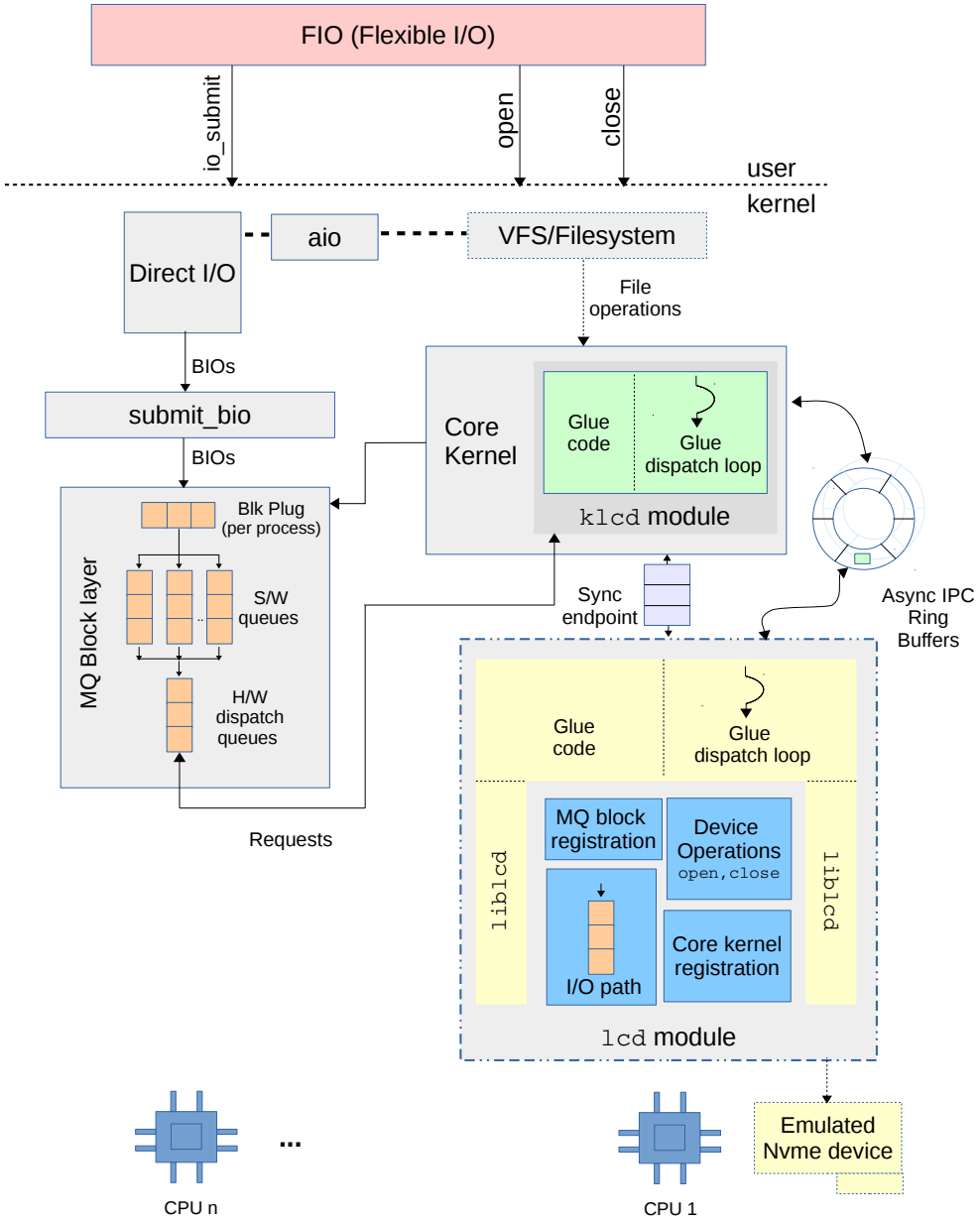

Figure 3.5. Architecture of the isolated null block driver

```
static void dispatch_loop(void) {

    int stop = 0;
    int ret = 0;
    struct fipc_message **msg_out;

    DO_FINISH(
        while (!stop) {
            ...
            ret = thc_ipc_poll_recv(async_chnl, msg_out);
            if (!ret) {
                ASYNC(
                    ret = handle_msg(msg_out);
                    if (ret) {
                        LIBLCD_ERR("drv dispatch err");
                        stop = 1;
                });
            } else if (ret != -EWOULDBLOCK) {
                LIBLCD_ERR("async loop failed");
                stop = 1;
                break;
            }
            ...
        }
    );
}
```

Figure 3.6. Simplified version of the dispatch loop using ASYNC runtime

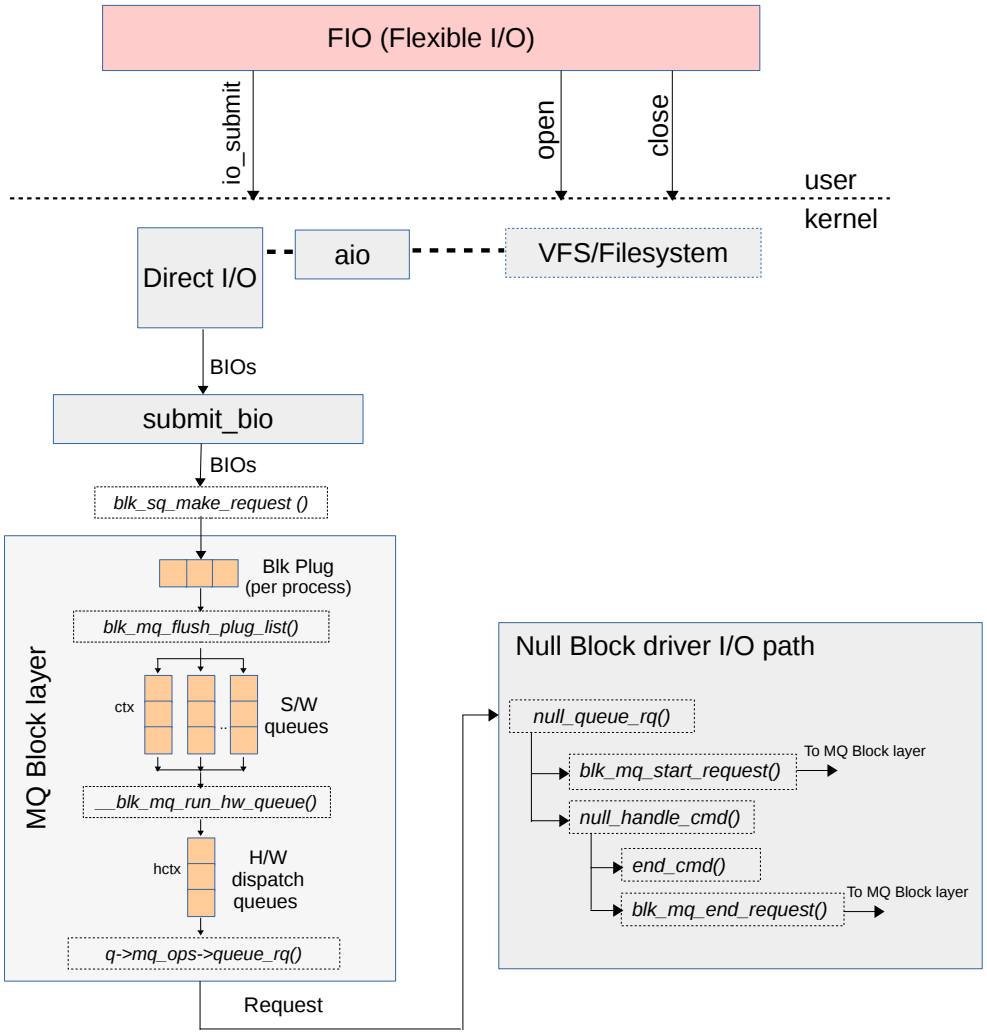

Figure 3.7. Function flow through the MQ block layer to the null block driver

```
static void __blk_mq_run_hw_queue(struct blk_mq_hw_ctx *hctx)
{
    struct request_queue *q = hctx->queue;
    struct request *rq;
    LIST_HEAD(rq_list);
    ...
    /* Touch any software queue that has pending entries. */
    flush_busy_ctxs(hctx, &rq_list);
    ...
    /* Now process all the entries, sending them to the driver. */
    queued = 0;
    while (!list_empty(&rq_list)) {
        struct blk_mq_queue_data bd;
        int ret;

        rq = list_first_entry(&rq_list, struct request, queuelist);
        list_del_init(&rq->queuelist);

        bd.rq = rq;

        ret = q->mq_ops->queue_rq(hctx, &bd);

        switch (ret) {
        case BLK_MQ_RQ_QUEUE_OK:
            queued++;
            break;
        ...
        }
        ...
    }
    ...
}
```

Figure 3.8. Simplified version of request processing loop in the MQ block layer

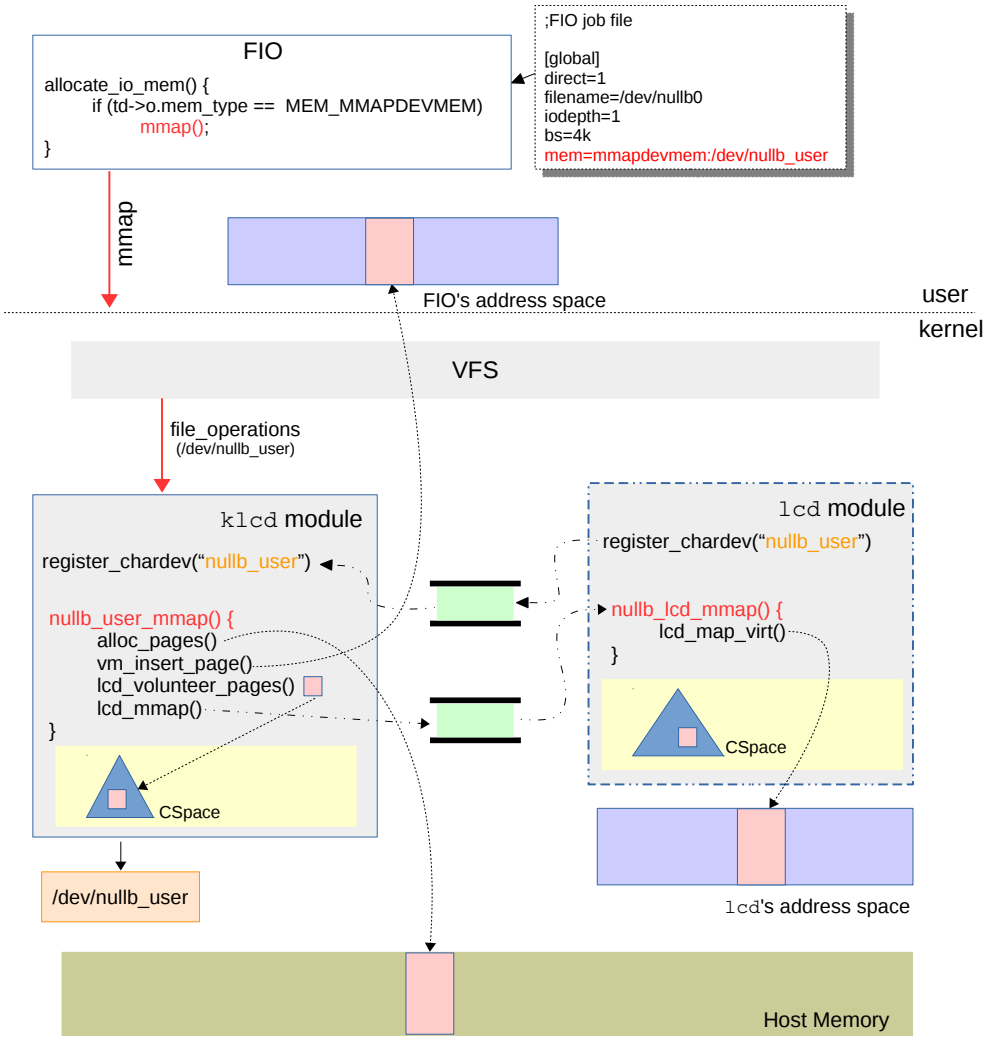

Figure 3.9. Memory sharing protocol between the application (*fio*) and the isolated null block driver

```
static void __blk_mq_run_hw_queue(struct blk_mq_hw_ctx *hctx)
{
    struct request_queue *q = hctx->queue;
    struct request *rq;
    LIST_HEAD(rq_list);
    ...
    /* Touch any software queue that has pending entries. */
    flush_busy_ctxs(hctx, &rq_list);
    ...
    /* Now process all the entries, sending them to the driver. */
    queued = 0;

    DO_FINISH(
        while (!list_empty(&rq_list)) {
            struct blk_mq_queue_data bd;
            int ret;

            rq = list_first_entry(&rq_list, struct request, queuelist);
            list_del_init(&rq->queuelist);

            bd.rq = rq;
                ASYNC({
                    ret = q->mq_ops->queue_rq(hctx, &bd);


                switch (ret) {
                case BLK_MQ_RQ_QUEUE_OK:
                    queued++;
                    break;
                ...
                    }
            });
            ...
        });
        ...
}
```

Figure 3.10. Simplified version of the request processing loop implemented using DO_FINISH and ASYNC macros

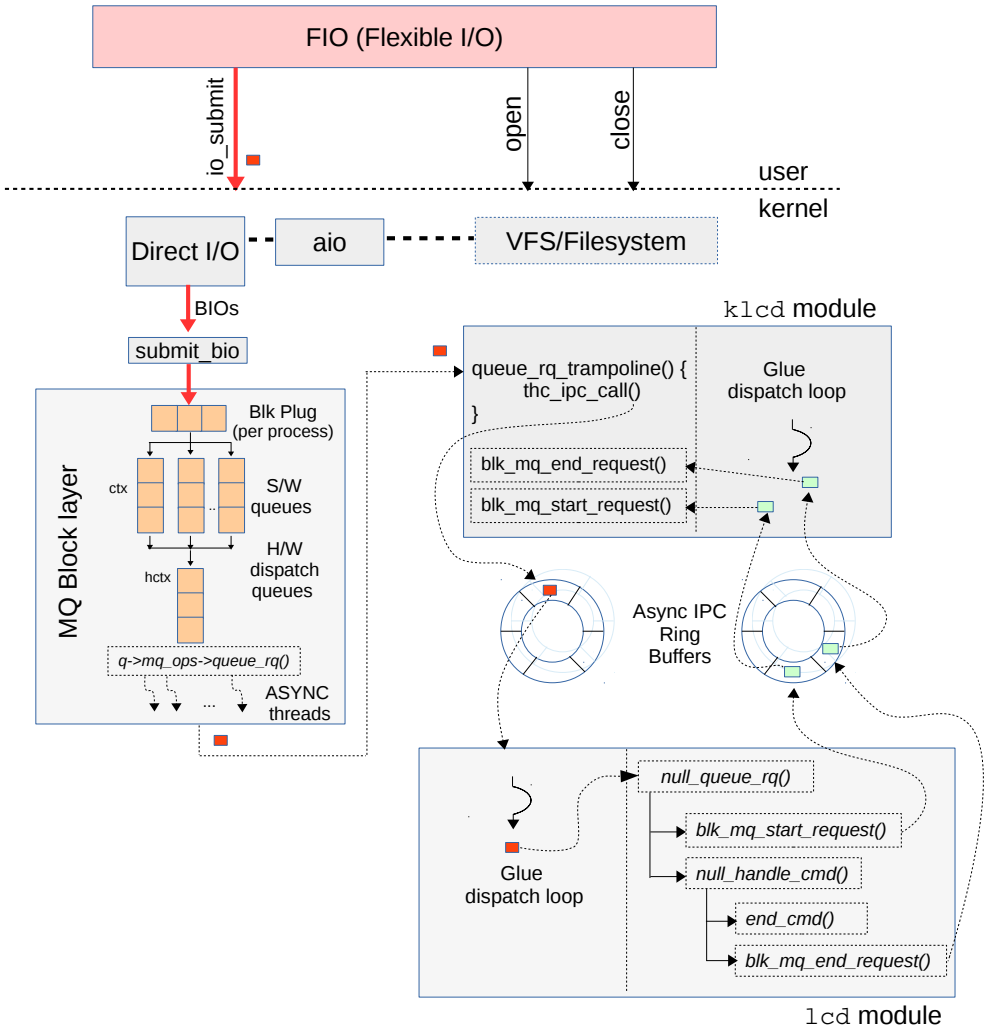

Figure 3.11. The flow of an I/O request from a user application to the driver through the MQ block layer

CHAPTER 4

RESULTS AND EVALUATION

We compare the performance of the isolated null block driver to the native driver, i.e., the nonisolated driver in the Linux kernel. We profile the I/O path and experiment with different block I/O size to understand the overheads of isolation.

4.1 Experiment Setup

We conduct our experiments on an Intel Xeon E5-4620 (2.20GHz) machine running Linux kernel v4.8.4. We disable hyper-threading, turbo boost, and frequency scaling to reduce the variance in benchmarking. To reduce the cache-coherency overheads on the IPC path, we pin the `lcd` and `klcd` threads on dedicated CPU cores within the same NUMA node.

4.2 I/O Load Generation

In our block device experiments, we rely on *fio* to generate I/O traffic. It is a widely used standard I/O benchmarking tool that allows us to carefully tune different parameters like I/O depth (`io_depth`) and block size (`bs`) for our tests. To set an optimal baseline for our evaluation, we choose the configuration parameters that can give us a lowest latency path to the driver. We use *fio*'s `libaio` engine (`ioengine=libaio`) to overlap I/O submissions and enable the `direct` I/O flag (`direct=1`) to ensure raw device performance. Even though our current LCD architecture can poll multiple IPC channels in a single dispatch thread, we restrict the number of I/O submission threads to one (`numjobs=1`) to understand the overheads of isolation induced by a single thread. In all our tests, we use the memory allocation scheme described in Section 3.3.3.2. We also implement the same data sharing mechanism in the native driver to keep the performance comparisons consistent.

4.3 Performance Evaluation

We profile the I/O submission and completion path from the user application to the driver and compare the timing of critical interfaces between the native and isolated driver. We also present the latency and throughput metrics of `fio` to assess the I/O performance of our isolated driver.

4.3.1 Timing Analysis

To measure the timing of critical functions in the I/O path, we configure *fio* to issue a single block I/O of lowest possible size (512 bytes) using the `libaio` engine. We use the `rdtsc` instruction provided by the architecture (x86) to profile these functions.

To submit an I/O request, *fio* issues the `io_submit` system call to the kernel. We saw earlier (see Section 2.4.1 and 3.3.2) that the MQ block layer executes a request processing loop within the `__blk_mq_run_hw_queue` function to flush the I/O requests to the driver. We also saw that the request processing loop calls the `queue_rq` interface of the driver to process a particular I/O request. We now present the timing comparison of these functions between the native and isolated null block driver.

- **Native driver:** We notice that the `io_submit` system call consumes 5677 cycles in user space. The `__blk_mq_run_hw_queue` function consumes 1439 cycles, which include the cost of the request processing loop (1140 cycles), `queue_rq` interface (1135 cycles), and the I/O processing functions `blk_mq_start_request` (155 cycles) and `blk_mq_end_request` (973 cycles), respectively. Figure 4.1 shows the timing split up in the native driver. It is important to note that, when `io_submit` returns to the user space, the `blk_mq_end_request` has already executed, and the completion of the I/O request is available. Later when *fio* looks for completions, it immediately finds the required number of completions without any delay.

- **Isolated driver:** Recall that in the isolated null block driver, we replace the function calls with cross-domain IPC requests. It is worthwhile to note that there are three threads in action: *fio*'s I/O submission, `lcd`'s, and the `klcd`'s dispatch threads, each pinned on separate CPU cores within the same NUMA node.

  Our measurements show that the `queue_rq` IPC request consumes 5130 cycles, which

includes two IPC call-reply invocations for `blk_mq_start_request` (344 cycles) and `blk_mq_end_request` (2300 cycles), respectively, as shown in Figure 4.2. The cost of isolation is 2782 cycles because we wait for the responses of `blk_mq_start_request` and `blk_mq_end_request` functions in the `lcd` module. With the optimizations discussed in Section 3.3.3.3, we reduce the cost of isolation significantly. The request processing loop inside `__blk_mq_run_hw_queue` consumes 895 cycles, which is less than the native driver's 1140 cycles. The `queue_rq` IPC request consumes 600 cycles, as shown in Figure 4.3. When we batch I/O requests, the request processing loop takes 1315 cycles because of the overhead introduced by the ASYNC runtime. It is worthy to note that the execution times of `blk_mq_start_request` and `blk_mq_end_request` functions are not factored inside the 600 cycles. It also implies that the `io_submit` system call returns to the user space without executing `blk_mq_start_request` and `blk_mq_end_request` functions as in the case of the native driver.

4.3.2 Fio Benchmarks

To assess the I/O performance of our isolated null block driver, we rely on the throughput and latency metrics reported by *fio*. We configure *fio* to batch I/O submissions and poll for completions from user space. By polling for I/O completions directly from user space, we eliminate the latency of a system call in the completion path. Apart from the configuration parameters described in Section 4.2, we tune the I/O depth (`io_depth`) from 1 to 16 and vary the block size (`bs`) from 512 bytes to 4 MB. To experiment with throughput, we batch I/O requests by setting `iodepth_batch` to the value of I/O depth. We experiment with latency by issuing a single I/O request at a time and also try to retrieve up to the whole submitted queue depth by polling directly from user space (`userspace_reap=1`). In all the tests, we ensure that at least a million I/O requests are submitted to the driver. The graphs shown in Figure 4.4, 4.5, and 4.6 compare the performance between the native and isolated null block driver. The x-axis of the graphs represents the test case, which is of the form *block size-I/O depth*. For instance, *512-16* indicates the test run with the block size of 512 bytes and I/O depth of 16. The values represent the average over five separate test runs.

4.3.2.1 Isolation Overhead

We can see from the IOPS graph (Figure 4.4) that the native driver achieves 308K IOPS for a single request of 512 bytes. In other words, a single I/O request takes 2.63μs to complete, whereas our isolated driver achieves 264K IOPS (3.77μs). We incur an additional overhead of 1.1μs (2500 cycles) due to isolation. We saw in the previous section (Section 4.3.1) that the native driver finds I/O completions immediately because the `blk_mq_end_request` function finishes during the `io_submit` call, whereas in the isolated null block driver, the `io_submit` call returns to the user space before the MQ block layer starts processing the I/O request. The isolated null block driver takes less time to submit an I/O request, but it loses time while polling for completions. The submission and completion latency graphs shown in Figure 4.5 and 4.6 captures this effect. More importantly, the I/O submission and completion happen on two different cores resulting in a remote memory accesses to a bitmap tag.

For higher I/O depths (`iodepth > 8`), the isolated driver matches the performance of the native driver. Moreover, for the block sizes of 1MB and higher, the isolated null block driver is 3.2% faster due to the request pipelining introduced in the MQ block layer.

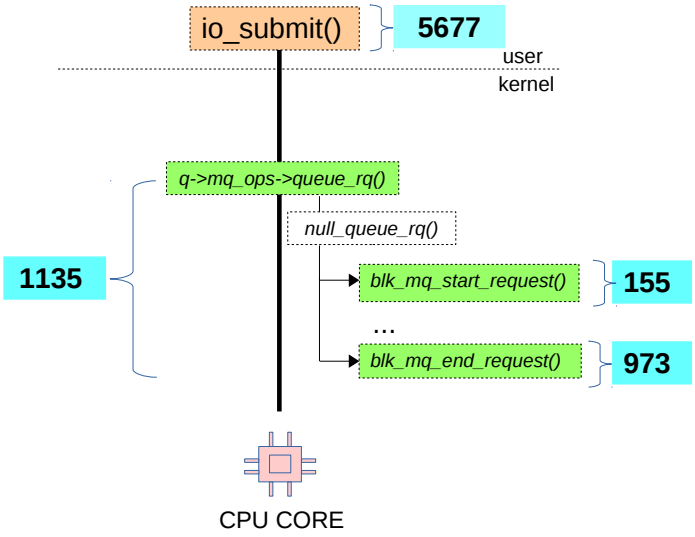

Figure 4.1. Timing analysis of native null block driver

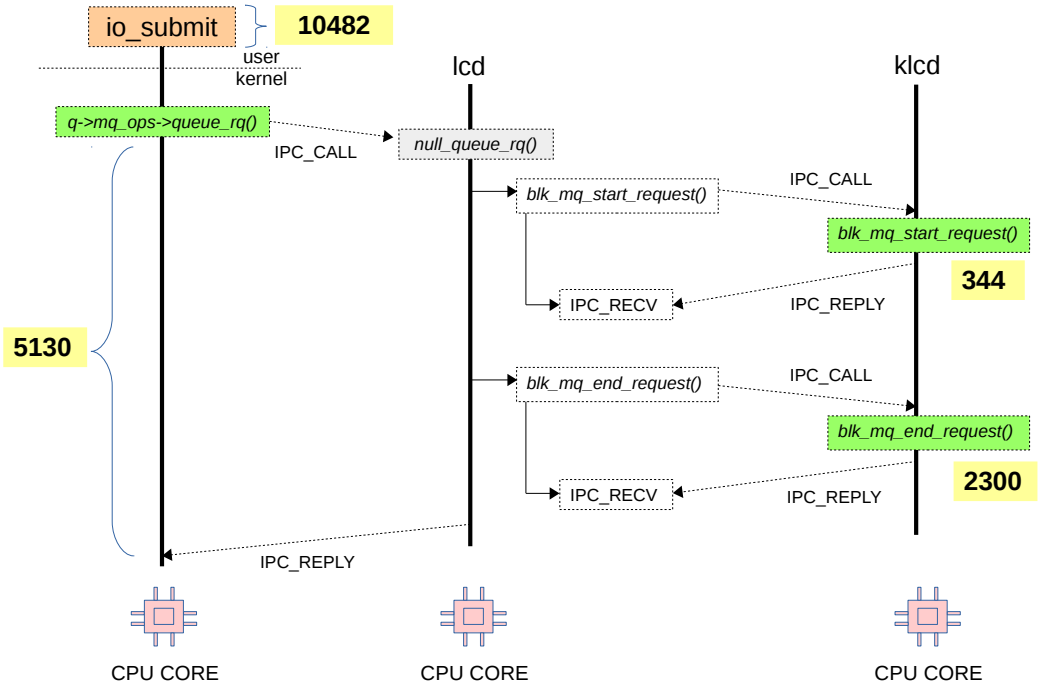

Figure 4.2. Timing analysis of unoptimized isolated null block driver

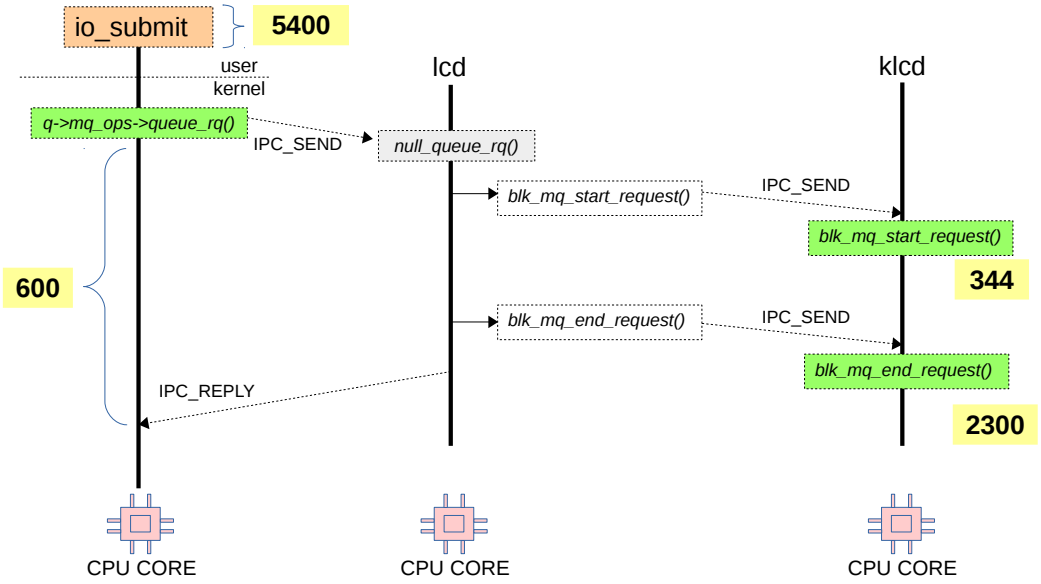

Figure 4.3. Timing analysis of optimized isolated null block driver. Note that the functions blk_mq_start_request and blk_mq_end_request execute in the background and its execution time is not factored in the IPC cost

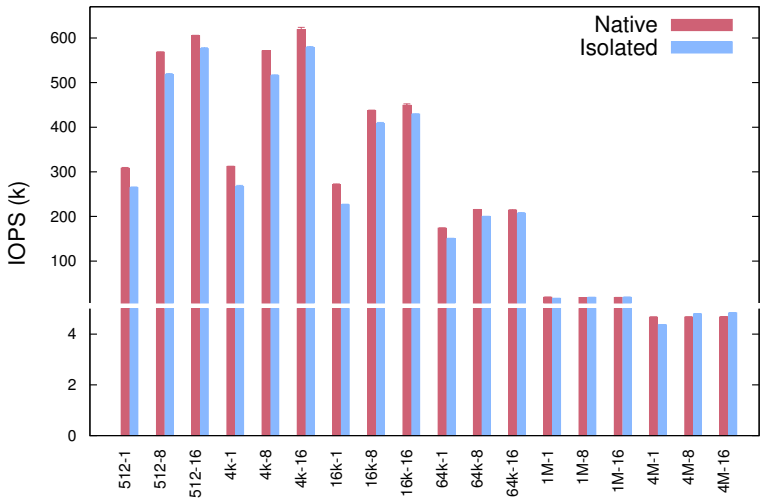

Figure 4.4. IOPS

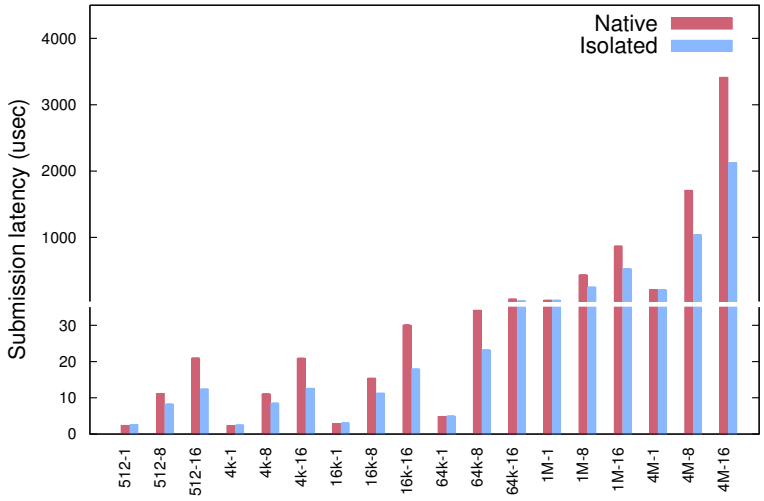

Figure 4.5. Submission latency

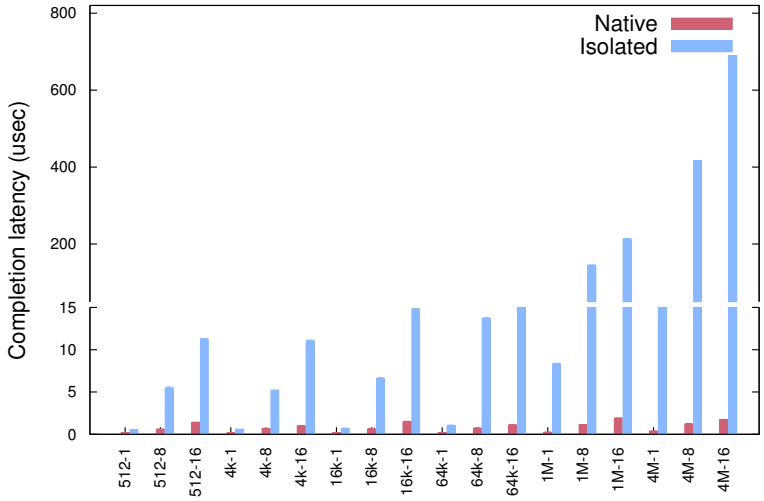

Figure 4.6. Completion latency

CHAPTER 5

VULNERABILITY ANALYSIS

In this chapter, we examine the security guarantees of our LCD architecture by evaluating the effects of kernel vulnerabilities. We classify the Linux kernel vulnerabilities, published in CVE database [31] in 2016, based on the type of attacks. We observe that out of the 217 vulnerabilities, 54 are from device drivers, 33 from the network subsystem, and 22 in the filesystems.

To test our hypothesis that LCDs can provide strong isolation of driver code, we carefully examine the vulnerabilities found in device drivers and categorize them based on the type of attacks. We choose a CVE under each type, analyze the source of the bug, and evaluate its effect in our framework. Based on the evaluation, we generalize the possibility of different attack scenarios. Table 5.1 summarizes our classification. Note that some vulnerabilities allow for more than one kind of attack.

- **Denial-of-service (DoS):** Out of the 42 vulnerabilities that lead to DoS attacks, 23 of them are only DoS and the rest also allow for other attacks. Out of the 23, 13 are because of NULL pointer deferences in the code. Although our framework does not confine DoS attacks, the effects of NULL pointer deferences do not lead to a complete system crash. For instance, CVE-2016-3951 reports a double-free vulnerability that leads to a system-wide crash, but in LCDs, the crash is only limited within its domain, and it does not propagate the fault to the nonisolated kernel. In summary, LCDs cannot prevent DoS attacks from happening, but they do not result in a complete system crash.

- **Code execution:** CVE-2016-8633 reports an arbitrary code execution vulnerability in the FireWire driver allowing remote attackers to execute arbitrary code via crafted fragmented packets. The driver lacked input validation while handling incoming fragmented datagrams, which led to a copy of data past the datagram buffer, en-

abling the attacker to execute code in kernel memory. In LCDs, the code execution is limited to resources within an isolated domain. In the worst case, Return-oriented programming (ROP) attacks can be constructed using the VMCALL interface, but the adversary will be not able to volunteer arbitrary kernel or write to arbitrary kernel memory. Therefore, LCDs completely weaken code execution to DoS.

- **Buffer overflow and memory corruption:** LCDs weaken both buffer overflow and memory corruption to DoS. For instance, CVE-2016-9083 reports a memory corruption vulnerability caused due to improper sanitation of user-supplied arguments. LCDs cannot prevent memory corruption, but it can confine the effect within its address space. LCDs can trigger an EPT fault restricting the fault within its domain. The same applies for CVE-2016-5829, which reports a heap overflow caused because of improper validation of user-supplied values.

- **Information leak:** CVE-2016-0723 reports information leak from kernel memory caused because of a race condition in accessing a data structure pointer. In LCDs, we replicate the data structures. Thus, the leak is restricted within LCD's address space. On a similar note, CVE-2016-4482 and CVE-2015-8964 also fall in the same category.

- **Gain privileges:** In CVE-2016-2067, the GPU driver erroneously interprets read permissions and marks user pages as writable by the GPU. An attacker can successfully map shared libraries with write permissions and modify the code pages to gain privileges. In LCDs, we would still mark the user requested pages with write permissions and we do not have a way to prevent this attack.

We show from the set of 54 driver vulnerabilities that LCDs can contain code execution, buffer overflow, and memory corruption attacks by weakening them to a DoS and still keep the rest of the kernel unaffected. We restrict information leak attacks within the LCD. In the worst case, if the LCD maps a set of kernel pages into its address space, an attacker can potentially leak that information. With privilege escalation attacks, we saw corner cases where the current LCD architecture is not capable of handling them. We also observed that improper handling of `ioctl` calls from userspace was one of the primary sources of bugs

in drivers. At this point, LCDs do not have the infrastructure to handle `ioctl` calls, but doing so in future requires a much more profound analysis of these vulnerabilities.

Table 5.1. Vulnerabilities in device drivers classified based on the type of attack

DoS	Code execution	Buffer overflow	Memory corruption	Information leak	Gain privileges
42	2	13	7	7	8

CHAPTER 6

RELATED WORK

During the past several years, multiple projects have focused on decomposing an OS for improved security and reliability [14, 23, 30, 40]. We classify the different approaches as follows:

- **Protection domains**: Several approaches decompose the kernel to isolate kernel components into protection domains. Nooks [40] isolates unmodified device drivers inside the Linux kernel by creating lightweight protection domains. It uses hardware page tables to restrict write access to kernel pages. Similar to LCDs, it also maintains and synchronizes private copies of kernel objects; however, the synchronization code is built manually. In LCDs, we use an IDL to automate the generation of the synchronization code. Moreover, Nooks requires switching page tables on each context switch between the protection domain and the core kernel, so the performance overhead is quite significant.

  Sawmill [15] decomposes the Linux kernel as user-level servers on top of the L4 microkernel. Similar to LCDs, it relies on an IDL compiler (Flick and IDL4) to generate stub-code. Although Sawmill promises near-native performance, it is not clear how much code had to be manually built to factor subsystems into user-level. Unfortunately, their implementation is not openly available for analysis.

  SIDE [39] runs unmodified drivers in kernel space but by lowering the privileges. A helper module facilitates driver's communication with the kernel and is implemented via system calls. Unlike LCDs, the helper module is built manually. SIDE achieves near native performance at an increased CPU overhead.

- **User-space device drivers**: Some researchers [23] propose the idea of running device drivers as user space applications. Microkernel-based systems like L4 [17] and Minix

3 [18] take this approach.

Microdrivers [14] partition a device driver into a performance critical *k-driver* and a noncritical *u-driver*. While the performance overhead is close to zero, they do not isolate the kernel component. A bug in the performance critical kernel part can still crash the entire system. This approach requires a lot of engineering effort to rewrite drivers as opposed to reusing device drivers of a more mature monolithic kernel such as Linux.

SUD [8] runs unmodified device drivers as user processes in a user mode Linux (UML) infrastructure. For every unmodified driver, a kernel proxy driver is used to handle the corresponding device and to channel user requests. To avoid the overhead of context-switches, SUD uses message queues. SUD provides strong isolation guarantees by achieving near-native performance, but it incurs more than two times CPU overhead. Moreover, a kernel mode proxy driver has to be manually developed for implementing a user driver.

- **Hardware virtualization**: Alternatively, Sumpf et al. [38] and Fraser et al. [12] achieve device driver isolation by running unmodified driver code in a virtualized container called a Driver Domain (DD). DD has a back end driver that multiplexes I/O from different front end drivers running in separate virtual domains over the real device driver. The problem with both of the approaches is that they run a full OS stack alongside the device driver to handle the dependencies of the driver.

  VirtuOS [30] employs hardware virtualization to partition kernel components into service domains. Each domain implements a subset of the kernel's functionality like storage and networking. The system is built on top of Xen [2] and relies on the shared memory capabilities provided by Xen to establish communication between different domains. Though the service domains provide full protection and survive failures, they run a near stock version of the kernel to provide an execution environment.

- **Software fault isolation**: Another method to achieve isolation is by using software fault isolation (SFI) techniques [9, 27, 36, 41]. But unfortunately, these techniques either compromise performance in favor of isolation or require modifications to existing code. For instance, LXFI[27] uses SFI to isolate kernel modules from the core

kernel. To use LXFI, device driver programmers must first specify the security policy for a kernel API using source-level annotations. LXFI then guarantees security with the help of two components: a compiler plugin that inserts calls and checks into the code and a runtime that validates whether a module has necessary privileges for any given operation. This technique is not transparent to existing code, and it involves nontrivial modifications adding a layer of complexity during debugging.

- **Language-based isolation**: Finally, researchers have also tried a more radical approach to isolation by implementing the kernel in a type-safe language. Projects like Singularity [19] and Spin [5] take this approach. In Singularity, researchers implement a microkernel using an extension of the C# language, whereas Spin leverages the features of the Modula-3 programming language. Both of these approaches suffer performance nondeterminism due to managed runtime and garbage collection (GC). Hence, these solutions have remained impractical.

  Recent developments in programming languages have led to a new systems programming language called Rust. Rust enforces type and memory safety through a restricted ownership model and has zero GC overhead. In our other work [1], we show that safe features of Rust can be used for fault isolation. Also, recent projects [25] show that Rust can be used to build a practical embedded system OS.

  Although Rust offers type safety at a zero performance overhead, rewriting a kernel from scratch takes years of development effort. Moreover, the single ownership model restricts the ability to express cyclic data like linked lists, so many of the low-level data structures have to remain unsafe.

CHAPTER 7

CONCLUSIONS

In this thesis, we augment the LCD architecture with useful features to isolate high-performance device drivers. Our work motivation was that the existing driver isolation techniques either compromise performance to safety or require significant development effort. We demonstrated by developing an isolated null block driver that unmodified drivers can be isolated with little effort while not compromising performance. Although additional infrastructure like multithreading and interrupts may be required to isolate other driver subsystems, we think that LCDs will remain lightweight. We also anticipate that future trends in OS design will move toward a distributed kernel design, where subsystems are pinned to specific resources in the system. We believe that it is entirely feasible to transition into a distributed kernel model while still reusing code of a mature monolithic kernel like Linux.

7.1 Limitations

In the current architecture, we restrict ourselves to a single submission thread, because the LCDs are single threaded. Although LCDs can listen to multiple IPC channels in a round-robin fashion, we think that by making LCDs multithreaded, we can arrive at new design possibilities.

The current implementation of isolated null block driver requires two CPU cores. We explored the idea of using the logical core within the same CPU to pin one of these tasks, but we did not see any performance gains with this approach. One other idea could have been to relinquish the CPU core to other tasks in the system, but we leave this to future work.

7.2 Future Work

Armed with the lessons learned from our current work, we plan to isolate the NVMe driver in the future. Our initial analysis brings out some of the missing features like timers, workqueues, support for Direct Memory Access (DMA), and interrupts. We require support for direct device assignment within LCDs, which includes the ability to program the IOMMU and handle interrupts without exits to the microkernel.

Finally, although the IDL compiler automatically generates the glue code, we still require a manual analysis of the kernel code to generate the IDL. Moreover, the current IDL compiler cannot handle complex patterns like circular dependency of data structures. We believe that by improving the IDL compiler and by eliminating the manual effort to generate IDL can help to decompose other complex subsystems in the kernel.

REFERENCES

[1] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, *System programming in rust: Beyond safety*, in Proceedings of the 16th Workshop on Hot Topics in Operating Systems, HotOS '17, New York, NY, USA, 2017, ACM, pp. 156–161.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, *Xen and the art of virtualization*, in Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, New York, NY, USA, 2003, ACM, pp. 164–177.

[3] S. Bauer, *Fip-see: A low latency, high throughput IPC mechanism*, tech. rep., 2016.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, *The multikernel: A new os architecture for scalable multicore systems*, in Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, New York, NY, USA, 2009, ACM, pp. 29–44.

[5] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, *Extensibility safety and performance in the spin operating system*, in Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, New York, NY, USA, 1995, ACM, pp. 267–283.

[6] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, *Linux block io: Introducing multi-queue ssd access on multi-core systems*, in Proceedings of the 6th International Systems and Storage Conference, SYSTOR '13, New York, NY, USA, 2013, ACM, pp. 22:1–22:10.

[7] J. Bonwick, *The slab allocator: An object-caching kernel memory allocator*, in Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1, USTC'94, Berkeley, CA, USA, 1994, USENIX Association, pp. 6–6.

[8] S. Boyd-Wickizer and N. Zeldovich, *Tolerating malicious device drivers in linux*, in Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'10, Berkeley, CA, USA, 2010, USENIX Association, pp. 9–9.

[9] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, *Fast byte-granularity software fault isolation*, in Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, New York, NY, USA, 2009, ACM, pp. 45–58.

[10] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, *Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks*, in Proceedings of the 7th Conference on USENIX

Security Symposium - Volume 7, SSYM'98, Berkeley, CA, USA, 1998, USENIX Association, pp. 5–5.

[11] Eyal Itkin, *Cve 2016-8633: Linux kernel firewire driver remote code execution*. `https://eyalitkin.wordpress.com/2016/11/06/cve-publication-cve-2016-8633/`, 2016.

[12] K. Fraser, *Safe hardware access with the xen virtual machine monitor*, in Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure, (OASIS), 2004.

[13] Free electrons, *Null block driver*. `http://elixir.free-electrons.com/linux/v4.8.4/source/drivers/block/null_blk.c`.

[14] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, *The design and implementation of microdrivers*, in ACM SIGARCH Computer Architecture News, vol. 36, ACM, 2008, pp. 168–178.

[15] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther, *The sawmill multiserver approach*, in Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, EW 9, New York, NY, USA, 2000, ACM, pp. 109–114.

[16] T. Harris, M. Abadi, R. Isaacs, and R. McIlroy, *Ac: composable asynchronous io for native languages*, ACM SIGPLAN Notices, 46 (2011), pp. 903–920.

[17] G. Heiser and K. Elphinstone, *L4 microkernels: The lessons from 20 years of research and deployment*, ACM Trans. Comput. Syst., 34 (2016), pp. 1:1–1:29.

[18] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, *Minix 3: A highly reliable, self-repairing operating system*, ACM SIGOPS Operating Systems Review, 40 (2006), pp. 80–89.

[19] G. C. Hunt and J. R. Larus, *Singularity: rethinking the software stack*, ACM SIGOPS Operating Systems Review, 41 (2007), pp. 37–49.

[20] C. Jacobsen, *Lightweight capability domains: Toward decomposing the linux kernel*, Master's thesis, University of Utah, 2016.

[21] Jens Axboe, *Flexible i/o generator*. `https://github.com/axboe/fio/blob/master/HOWTO`.

[22] Jonathan Corbet, *The iov_iter interface*. `https://lwn.net/Articles/625077/`, 2014.

[23] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.-T. Shen, K. Elphinstone, and G. Heiser, *User-level device drivers: Achieved performance*, Journal of Computer Science and Technology, 20 (2005), pp. 654–664.

[24] J. Levin, *Mac OS X and iOS Internals: To the Apple's Core*, Wrox, 2012.

[25] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, *The case for writing a kernel in rust*, in Proceedings of the 8th Asia-Pacific Workshop on Systems, ACM, 2017, p. 1.

[26] R. Love, *Linux System Programming: Talking Directly to the Kernel and C Library*, O'Reilly Media, Inc., 2007.

[27] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, *Software fault isolation with api integrity and multi-principal modules*, in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ACM, 2011, pp. 115–128.

[28] R. McDougall and J. Mauro, *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*, Prentice Hall, 2006.

[29] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The Design and Implementation of the FreeBSD Operating System*, Addison-Wesley Professional, 2014.

[30] R. Nikolaev and G. Back, *Virtuos: An operating system with kernel virtualization*, in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ACM, 2013, pp. 116–132.

[31] S. Özkan, *Linux kernel vulnerability statistics*. `http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33`.

[32] M. Quigley, *Extensions to barrelfish asynchronous c*, tech. rep., 2016.

[33] W. River, *Vxworks programmer's guide*, 2003.

[34] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, *Return-oriented programming: Systems, languages, and applications*, ACM Transactions on Information and System Security (TISSEC), 15 (2012), p. 2.

[35] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, *On the effectiveness of address-space randomization*, in Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04, New York, NY, USA, 2004, ACM, pp. 298–307.

[36] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, *Enforcing kernel security invariants with data flow integrity*, in Proceedings of the 23th Annual Network and Distributed System Security Symposium, 2016.

[37] S. Spall, *kIDL: interface definition language for the kernel*, tech. rep., 2016.

[38] S. Sumpf and J. Brakensiek, *Device driver isolation within virtualized embedded platforms*, in 2009 6th IEEE Consumer Communications and Networking Conference, IEEE, 2009, pp. 1–5.

[39] Y. Sun and T.-c. Chiueh, *Side: isolated and efficient execution of unmodified device drivers*, in Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on, IEEE, 2013, pp. 1–12.

[40] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, *Nooks: An architecture for reliable device drivers*, in Proceedings of the 10th Workshop on ACM SIGOPS European Workshop, EW 10, New York, NY, USA, 2002, ACM, pp. 102–107.

[41] T. Yoshimura, *A study on faults and error propagation in the linux operating system*, (2016).