Alex Magalhães da Silva Junior
Gabriel Pereira Mendonça Passos

Projeto Final

Projeto final solicitado pelo professor Rodrigo Campiolo na disciplina de Sistemas Operacionais do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR

Departamento Acadêmico de Computação – DACOM

Bacharelado em Ciência da Computação – BCC

Campo Mourão

Dezembro / 2025

Sumário

1 Introdução

O desenvolvimento de módulos para o kernel Linux é uma ferramenta essencial para compreender, de forma prática, como sistemas operacionais gerenciam recursos, dispositivos e a comunicação entre o espaço de usuário e o espaço do kernel. Segundo The Linux Kernel Module Programming Guide — uma das obras introdutórias mais consolidadas sobre o tema — "módulos permitem estender o kernel dinamicamente, adicionando funcionalidades sem necessidade de recompilação ou reinicialização do sistema" (SALZMAN; BURIAN; POMERANTZ, 2016). Essa característica torna o ambiente ideal para experimentação, especialmente no contexto acadêmico.

Neste trabalho, foi desenvolvido um módulo denominado `cryptochannel`, cujo objetivo é integrar diferentes conceitos fundamentais de Sistemas Operacionais: drivers de dispositivo, sincronização, gerenciamento de buffers, operações de entrada e saída, e interação via sistema de arquivos virtual `/proc`. O módulo cria um dispositivo de caractere em `/dev/cryptochannel`, através do qual processos podem trocar mensagens. Para essa comunicação, implementa-se um mecanismo de criptografia simples (XOR), possibilitando que os dados escritos sejam cifrados antes de serem armazenados em um buffer FIFO interno, e decifrados quando lidos.

Além da troca de mensagens, o módulo também expõe uma interface de monitoramento e configuração acessível por meio de `/proc/cryptochannel/`. Nela, o usuário pode ler estatísticas de uso em tempo real — como quantidade de mensagens processadas e erros ocorridos — bem como alterar parâmetros operacionais, como chave e modo de criptografia, ilustrando o uso prático de sistemas de arquivos virtuais no kernel.

A realização deste projeto permitiu observar, de forma aplicada, como conceitos apresentados em obras clássicas como Linux Device Drivers (CORBET; RUBINI; KROAH-HARTMAN, 2005) e nas diretrizes dos laboratórios do Linux Kernel Labs (DEVICE. . . ,) se integram para formar um driver funcional, modular e seguro. Dessa forma, este trabalho não apenas reforça o estudo teórico, mas também proporciona uma visão concreta de como mecanismos internos do kernel operam em conjunto.

2 Descrição

A atividade proposta consiste no desenvolvimento de um módulo de kernel para Linux capaz de realizar comunicação segura por meio de um dispositivo de caractere. O módulo deve criar um dispositivo acessível em `/dev/cryptochannel`, através do qual processos podem enviar e receber mensagens. As mensagens escritas no dispositivo devem ser **cifradas**, enquanto as mensagens lidas devem ser **decifradas**, utilizando um algoritmo de criptografia configurável.

Além do dispositivo de caractere, o módulo deve disponibilizar uma interface de controle por meio do sistema de arquivos `/proc`, localizada em `/proc/cryptochannel/`. Nessa interface, o arquivo `config` permite ajustar parâmetros do módulo, como o modo de operação e a chave de criptografia, enquanto o arquivo `stats` fornece estatísticas de uso, incluindo a quantidade de mensagens processadas, bytes cifrados e decifrados, além da ocorrência de erros.

A implementação requer o uso de buffers internos baseados em FIFO, mecanismo de sincronização por meio de mutexes e filas de espera (`wait queues`), tratamento adequado de chamadas de sistema bloqueantes e não bloqueantes, bem como integração com ferramentas padrão do Linux para carregamento, inspeção e remoção de módulos. O propósito da atividade é permitir que o aluno aplique, de forma prática, conceitos de comunicação entre processos, drivers de dispositivos, criptografia e mecanismos internos do kernel Linux.

3 Métodos

3.1 Estrutura do código-fonte

O código-fonte do módulo `cryptochannel` foi organizado em componentes bem definidos, seguindo o padrão de desenvolvimento de módulos do kernel Linux. Essa organização facilita a leitura, manutenção e validação da comunicação entre o dispositivo de caractere e a interface `/proc`. Abaixo são descritas as principais partes do arquivo `cryptochannel_dev.c`.

3.2 Bibliotecas e definições

O início do código contém os *headers* necessários, incluindo APIs oficiais do Linux para dispositivos de caractere, uso de buffers FIFO (`kfifo`), mecanismos de sincronização, interface `/proc`, operações sobre arquivos e funções auxiliares.

```c
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/uaccess.h>
#include <linux/slab.h>
#include <linux/kfifo.h>
#include <linux/mutex.h>
#include <linux/wait.h>
#include <linux/poll.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/atomic.h>
#include <linux/string.h>

#define DEVICE_NAME "cryptochannel"
#define PROC_DIR "cryptochannel"
#define PROC_STATS "stats"
#define PROC_CONFIG "config"

#define BUFFER_SIZE 8192
#define KEY_MAX_LEN 64
#define MAX_WRITE_SIZE 512
#define READ_TIMEOUT   (30 * HZ)
```

Figura 1 – Bibliotecas incluídas e definições de constantes

Além disso, foram definidas constantes globais responsáveis por parametrizar o funcionamento do módulo, tais como:

- Nome do dispositivo: `cryptochannel`;

- Nomes das entradas no sistema de arquivos `/proc`;
- Tamanho do buffer do FIFO utilizado para armazenamento das mensagens;
- Limite da chave de criptografia.
- Limite de tempo (*timeout*) para operações de leitura bloqueante.

3.3 Variáveis globais

O módulo utiliza um conjunto de variáveis globais responsáveis por manter o estado interno, gerenciar a concorrência e permitir a comunicação entre as diferentes rotinas do driver, conforme ilustrado na Figura 2.

```c
/**
 * struct cc_stats_t - Estrutura para estatísticas atômicas
 * Agrupa todos os contadores de desempenho e erro do módulo.
 */
static struct {
    atomic64_t messages;
    atomic64_t bytes_encrypted;
    atomic64_t bytes_decrypted;
    atomic64_t errors;
    atomic64_t reads_blocked;
    atomic64_t reads_timeout;
    atomic64_t reads_nonblock;
    atomic64_t writes_rejected;
    atomic64_t key_changes;
    atomic64_t invalid_ops;
} cc_stats;

static dev_t cc_dev;
static struct cdev cc_cdev;
static struct class *cc_class;

/* FIFO para armazenamento dos dados criptografados */
static DECLARE_KFIFO(cc_fifo, unsigned char, BUFFER_SIZE);

/* Mutex para proteção da chave e operações críticas */
static DEFINE_MUTEX(cc_lock);

/* Fila de espera para leitores (bloqueio na leitura) */
static DECLARE_WAIT_QUEUE_HEAD(cc_wq_read);

/* Configuração de criptografia */
static char cc_key[KEY_MAX_LEN] = {0};
static size_t cc_key_len = 0;
/* Inicia em -1 para obrigar a configuração explícita (Secure by Default) */
static int cc_mode = -1;
```

Figura 2 – Declaração das variáveis globais e estruturas de controle

As principais estruturas utilizadas são:

- `dev_t`, `cdev`, `class`, `device`: estruturas nativas do kernel Linux utilizadas para o registro dinâmico e criação do dispositivo de caractere `/dev/cryptochannel`;

- `DECLARE_KFIFO(cc_fifo, ...)`: define o buffer circular interno responsável por armazenar temporariamente as mensagens cifradas;
- `cc_lock` (mutex) e `cc_wq_read` (wait_queue): primitivas de sincronização empregadas para garantir a exclusão mútua na escrita e o bloqueio (dormência) dos processos leitores quando o buffer está vazio;
- Variáveis de configuração (`cc_key`, `cc_key_len`), com destaque para `cc_mode`, que é inicializada com o valor **-1** (sentinela) para impor a política de *Secure by Default*, obrigando a configuração explícita antes do uso;
- `cc_stats`: uma estrutura estática que agrupa todos os contadores atômicos (`atomic64_t`), como `messages`, `bytes_encrypted` e `errors`, permitindo a atualização segura de métricas em ambiente concorrente e facilitando a organização do código.

Essas variáveis representam o estado global do módulo e são fundamentais para o correto funcionamento do dispositivo e da interface de controle via `/proc`.

3.4 Função de criptografia

A função `xor_cipher()` é responsável por implementar o mecanismo de cifragem e decifragem adotado no módulo, baseado na operação lógica XOR. A função recebe um buffer de dados e aplica a operação XOR utilizando a chave previamente configurada pelo usuário.

Por se tratar de uma operação simétrica, o mesmo procedimento é utilizado tanto para cifrar quanto para decifrar as mensagens, simplificando a implementação. Essa camada de criptografia foi projetada de forma independente da lógica de leitura e escrita do dispositivo, o que facilita possíveis extensões futuras, como a substituição do algoritmo por mecanismos criptográficos mais robustos.

```c
/* ===== Cifra XOR ===== */
static void xor_cipher(unsigned char *buf, size_t len, const char *key, size_t key_len)
{
    size_t i;
    if (key_len == 0) return;

    for (i = 0; i < len; i++)
        buf[i] ^= key[i % key_len];
}
```

Figura 3 – Implementação da função de criptografia XOR

3.5 Operações do dispositivo de caractere

O módulo define a tabela `struct file_operations cc_fops`, responsável por associar as operações do dispositivo de caractere às funções implementadas no módulo. Dentre essas operações, destacam-se as funções `open()` e `release()`.

- `open()`: chamada quando um processo abre o dispositivo `/dev/cryptochannel`;
- `release()`: chamada quando o processo encerra o uso do dispositivo.

```
/* ===== Operações de arquivo do dispositivo ===== */
static int cc_open(struct inode *inode, struct file *filp) { return 0; }
static int cc_release(struct inode *inode, struct file *filp) { return 0; }
```

Figura 4 – Implementação das operações de abertura e fechamento

Essas funções possuem implementação simples e não realizam processamento adicional neste projeto, servindo principalmente para completar a interface do dispositivo de caractere conforme o modelo de drivers do kernel Linux.

- `read()` (bloqueante e não bloqueante)

```c
static ssize_t cc_read(struct file *filp, char __user *buf, size_t count, loff_t *ppos)
{
    unsigned int available;
    unsigned int n;
    unsigned char *kbuf;
    int ret;

    if (count == 0) return 0;

    /* Verifica modo não-bloqueante */
    if (filp->f_flags & O_NONBLOCK) {
        if (kfifo_is_empty(&cc_fifo)) {
            atomic64_inc(&cc_stats.reads_nonblock);
            return -EAGAIN;
        }
    }

    atomic64_inc(&cc_stats.reads_blocked);

    /* Espera bloqueante com timeout e suporte a sinais */
    ret = wait_event_interruptible_timeout(
        cc_wq_read,
        !kfifo_is_empty(&cc_fifo),
        READ_TIMEOUT
    );

    if (ret == 0) { // Timeout expirou
        atomic64_inc(&cc_stats.reads_timeout);
        return -ETIMEDOUT;
    }

    if (ret < 0) // Interrompido por sinal
        return -EINTR;

    /* Processamento dos dados */
    available = kfifo_len(&cc_fifo);
    n = min(available, (unsigned int)count);
```

Figura 5 – Função read: Tratamento de flags e wait_queue

```c
    kbuf = kmalloc(n, GFP_KERNEL);
    if (!kbuf) return -ENOMEM;

    /* Remove do FIFO (dados ainda cifrados) */
    if (kfifo_out(&cc_fifo, kbuf, n) != n) {
        atomic64_inc(&cc_stats.errors);
        kfree(kbuf);
        return -EIO;
    }

    mutex_lock(&cc_lock);
    /* Aplica descriptografia se houver chave e modo correto */
    if (cc_key_len > 0 && cc_mode == 0) {
        xor_cipher(kbuf, n, cc_key, cc_key_len);
    }
    atomic64_add(n, &cc_stats.bytes_decrypted);
    mutex_unlock(&cc_lock);

    if (copy_to_user(buf, kbuf, n)) {
        atomic64_inc(&cc_stats.errors);
        kfree(kbuf);
        return -EFAULT;
    }

    kfree(kbuf);
    return n;
}
```

Figura 6 – Função read: Descriptografia e cópia para o usuário

A operação `read()` foi implementada para suportar tanto o comportamento bloqueante quanto o não-bloqueante. Inicialmente, a função verifica a flag `O_NONBLOCK` no

arquivo. Se ativada e o buffer interno estiver vazio, a função retorna imediatamente o erro `-EAGAIN`, incrementando o contador de estatísticas `reads_nonblock`.

No modo padrão (bloqueante), o processo leitor é colocado em espera na `wait_queue` através da função `wait_event_interruptible_timeout`. O processo permanece suspenso até que: (1) dados estejam disponíveis no FIFO; (2) o processo receba um sinal (interrupção); ou (3) o tempo limite (`READ_TIMEOUT`) expire, retornando `-ETIMEDOUT`.

Quando dados são obtidos do `kfifo` (ainda criptografados), eles são movidos para um buffer temporário de kernel. A descriptografia ocorre dentro de uma região crítica protegida pelo mutex `cc_lock`. Neste momento, o driver verifica se há uma chave válida configurada e se o modo de operação está correto antes de aplicar o algoritmo XOR. Por fim, as estatísticas de bytes decifrados são atualizadas atomicamente e os dados em texto plano são copiados para o espaço de usuário via `copy_to_user`.]

- `write()` (não bloqueante)

```c
static ssize_t cc_write(struct file *filp, const char __user *buf, size_t count, loff_t *ppos)
{
    unsigned int avail;
    unsigned int n;
    unsigned char *kbuf;

    /* --- VALIDAÇÃO DE SEGURANÇA --- */
    mutex_lock(&cc_lock);
    if (cc_mode < 0 || cc_key_len == 0) {
        mutex_unlock(&cc_lock);
        return -ENOKEY; /* Requer configuração prévia */
    }
    mutex_unlock(&cc_lock);

    if (count > MAX_WRITE_SIZE) {
        atomic64_inc(&cc_stats.writes_rejected);
        return -EINVAL;
    }

    if (count == 0) return 0;

    avail = kfifo_avail(&cc_fifo);
    if (avail == 0) return -ENOSPC; /* Buffer cheio */

    n = min(avail, (unsigned int)count);

    kbuf = kmalloc(n, GFP_KERNEL);
    if (!kbuf) return -ENOMEM;

    if (copy_from_user(kbuf, buf, n)) {
        atomic64_inc(&cc_stats.errors);
        kfree(kbuf);
        return -EFAULT;
    }
```

Figura 7 – Função write: Validação e cópia do usuário

```
    mutex_lock(&cc_lock);
    /* Criptografa antes de armazenar */
    if (cc_mode == 0)
        xor_cipher(kbuf, n, cc_key, cc_key_len);

    atomic64_add(n, &cc_stats.bytes_encrypted);
    atomic64_inc(&cc_stats.messages);
    mutex_unlock(&cc_lock);

    /* Insere no FIFO */
    if (kfifo_in(&cc_fifo, kbuf, n) != n) {
        atomic64_inc(&cc_stats.errors);
        kfree(kbuf);
        return -EIO;
    }

    kfree(kbuf);

    /* Notifica leitores */
    wake_up_interruptible(&cc_wq_read);
    return n;
}
```

Figura 8 – Função write: Criptografia e inserção no FIFO

A operação `write()` segue o modelo não bloqueante, mas incorpora verificações de segurança robustas. Antes de qualquer processamento, a função valida o estado do dispositivo: caso o modo de operação ou a chave criptográfica não tenham sido configurados previamente via `/proc`, a chamada retorna imediatamente o erro `-ENOKEY`. Essa política de *Secure by Default* impede o tráfego acidental de dados não cifrados.

Após validar o tamanho da mensagem (limitado a `MAX_WRITE_SIZE`) e a disponibilidade de espaço no FIFO (retornando `-ENOSPC` se cheio), os dados são copiados para um buffer temporário de kernel. A etapa de cifragem ocorre dentro de uma região crítica protegida pelo mutex `cc_lock`. Isso garante a atomicidade da operação XOR e a consistência das estatísticas globais (`cc_stats`) mesmo com múltiplos produtores simultâneos. Por fim, os dados cifrados são inseridos no FIFO e os leitores são notificados.

- `poll()`

```
static unsigned int cc_poll(struct file *filp, poll_table *wait)
{
    poll_wait(filp, &cc_wq_read, wait);
    if (!kfifo_is_empty(&cc_fifo))
        return POLLIN | POLLRDNORM;
    return 0;
}
```

Figura 9 – Implementação da função poll para monitoramento de eventos

A função `poll()` permite que aplicações no espaço de usuário utilizem as chamadas `poll()` e `select()` para monitorar o dispositivo. A operação indica quando existem

dados disponíveis no FIFO interno, possibilitando a leitura sem bloqueio e a integração com aplicações orientadas a eventos.

- Tabela de operações (`file_operations`)

```
static const struct file_operations cc_fops = {
    .owner = THIS_MODULE,
    .open  = cc_open,
    .release = cc_release,
    .read  = cc_read,
    .write = cc_write,
    .poll  = cc_poll,
};
```

Figura 10 – Mapeamento das operações do sistema para funções do driver

A estrutura `struct file_operations` define as operações suportadas pelo dispositivo de caractere `/dev/cryptochannel`. Nela são associadas as funções implementadas pelo módulo às chamadas de sistema correspondentes, como `open()`, `read()`, `write()` e `poll()`, permitindo que o kernel redirecione corretamente as requisições feitas pelos processos no espaço de usuário.

3.6 Implementação da interface `/proc`

A interface disponibilizada em `/proc/cryptochannel/` constitui a camada administrativa do módulo, permitindo ao usuário visualizar estatísticas internas e configurar parâmetros de funcionamento do dispositivo. Essa interface é composta por dois arquivos principais.

3.6.1 /proc/cryptochannel/stats

```c
/* ===== /proc/cryptochannel/stats ===== */
static int proc_stats_show(struct seq_file *m, void *v)
{
    seq_printf(m, "Mensagens_trocadas: %lld\n", atomic64_read(&stat_messages));
    seq_printf(m, "Bytes_criptografados: %lld\n", atomic64_read(&stat_bytes_encrypted));
    seq_printf(m, "Bytes_descriptografados: %lld\n", atomic64_read(&stat_bytes_decrypted));
    seq_printf(m, "Modo: %d\n", cc_mode);
    seq_printf(m, "Tamanho_chave: %zu\n", cc_key_len);
    seq_printf(m, "Leituras_bloqueadas: %lld\n", atomic64_read(&stat_reads_blocked));
    seq_printf(m, "Leituras_timeout: %lld\n", atomic64_read(&stat_reads_timeout));
    seq_printf(m, "Leituras_sem_dados: %lld\n", atomic64_read(&stat_reads_nonblock));
    seq_printf(m, "Escritas_rejeitadas: %lld\n", atomic64_read(&stat_writes_rejected));
    seq_printf(m, "Mudancas_de_chave: %lld\n", atomic64_read(&stat_key_changes));
    seq_printf(m, "Tentativas_invalidas: %lld\n", atomic64_read(&stat_invalid_ops));
    seq_printf(m, "Erros: %lld\n", atomic64_read(&stat_errors));
    return 0;
}

static int proc_stats_open(struct inode *inode, struct file *file)
{
    return single_open(file, proc_stats_show, NULL);
}

/* proc_ops para stats (compatível kernel moderno) */
static const struct proc_ops proc_stats_ops = {
    .proc_open    = proc_stats_open,
    .proc_read    = seq_read,
    .proc_lseek   = seq_lseek,
    .proc_release = single_release,
};
```

Figura 11 – Função show para exibição das estatísticas via seq_file

O arquivo `stats` tem como finalidade fornecer informações estatísticas sobre o funcionamento do canal criptográfico, incluindo:

- número de mensagens processadas;
- quantidade total de bytes cifrados;
- quantidade total de bytes decifrados;
- erros ocorridos durante a execução;
- modo de criptografia atualmente ativo;
- tamanho da chave configurada.

A implementação utiliza a infraestrutura `seq_file`, adequada para a geração de saídas sequenciais no sistema de arquivos `/proc`. Ao abrir o arquivo `/proc/cryptochannel/stats`, o kernel invoca a função `proc_stats_open()`, que registra o callback responsável pela exibição dos dados por meio da chamada `single_open(file, proc_stats_show, NULL)`.

A lógica principal encontra-se na função `proc_stats_show()`, onde os contadores atômicos do módulo são coletados e exibidos utilizando `seq_printf`. O uso de variáveis

do tipo `atomic64_t` garante a consistência das estatísticas mesmo na presença de acessos concorrentes ao dispositivo.

3.6.2 `/proc/cryptochannel/config`

```c
/* ===== /proc/cryptochannel/config ===== */
static ssize_t proc_config_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    char temp[256];
    size_t len;

    mutex_lock(&cc_lock);
    len = scnprintf(temp, sizeof(temp), "modo=%d\nchave=%s\n", cc_mode, cc_key);
    mutex_unlock(&cc_lock);

    return simple_read_from_buffer(buf, count, ppos, temp, len);
}
```

Figura 12 – Leitura da configuração atual (Modo e Chave)

```c
static ssize_t proc_config_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
{
    char kbuf[256];
    size_t n;

    n = min(count, sizeof(kbuf) - 1);
    if (copy_from_user(kbuf, buf, n)) return -EFAULT;
    kbuf[n] = '\0';

    /* Remove quebra de linha final se existir */
    if (n > 0 && kbuf[n-1] == '\n')
        kbuf[n-1] = '\0';

    mutex_lock(&cc_lock);

    /* --- Configuração de MODO --- */
    if (strncmp(kbuf, "modo=", 5) == 0) {
        int new_mode;
        if (kstrtoint(kbuf + 5, 10, &new_mode) || new_mode != 0) {
            atomic64_inc(&cc_stats.invalid_ops);
            mutex_unlock(&cc_lock);
            return -EINVAL;
        }
        cc_mode = new_mode;
```

Figura 13 – Escrita de configuração: Parsing dos comandos

```
    /* --- Configuração de CHAVE --- */
    } else if (strncmp(kbuf, "chave=", 6) == 0) {
        /* Declaração local para evitar warning de unused variable em outros blocos */
        char *new_val_ptr = kbuf + 6;
        size_t new_len = strlen(new_val_ptr);

        if (!kfifo_is_empty(&cc_fifo)) {
            atomic64_inc(&cc_stats.invalid_ops);
            mutex_unlock(&cc_lock);
            return -EBUSY;
        }

        /* CORREÇÃO: Valida tamanho ANTES de alterar a chave global */
        if (new_len == 0) {
            atomic64_inc(&cc_stats.invalid_ops);
            mutex_unlock(&cc_lock);
            return -EINVAL; /* Chave vazia rejeitada */
        }

        strncpy(cc_key, new_val_ptr, KEY_MAX_LEN - 1);
        cc_key[KEY_MAX_LEN - 1] = '\0';
        cc_key_len = strnlen(cc_key, KEY_MAX_LEN - 1);

        atomic64_inc(&cc_stats.key_changes);

    } else {
        atomic64_inc(&cc_stats.invalid_ops);
        mutex_unlock(&cc_lock);
        return -EINVAL;
    }

    mutex_unlock(&cc_lock);
    return count;
}
```

Figura 14 – Escrita de configuração: Validação de segurança e atualização

O arquivo `/proc/cryptochannel/config` atua como o painel de controle do dispositivo, permitindo ao administrador:

- consultar o estado atual (modo e chave ativa);
- injetar a chave criptográfica inicial;
- alterar a chave em tempo de execução (rotação de chaves).

Ao executar a leitura (`cat`), a função `proc_config_read()` obtém o bloqueio do mutex, formata o estado das variáveis globais em um buffer temporário e o envia ao usuário, garantindo uma visualização consistente.

A escrita (`proc_config_write()`) recebeu implementações críticas de segurança para evitar a corrupção de dados. O fluxo de processamento é:

1. Os dados são copiados do usuário e o `mutex_lock(&cc_lock)` é adquirido imediatamente;
2. O comando é analisado (`"modo="` ou `"chave="`);

3. Caso seja uma alteração de `"chave="`, duas validações ocorrem antes da alteração:
    - **Verificação de Integridade:** O driver checa se o FIFO está vazio. Se houver mensagens pendentes no buffer, a troca é rejeitada com erro `-EBUSY`. Isso impede que mensagens cifradas com a chave antiga sejam decifradas incorretamente com a nova chave (gerando lixo);
    - **Validação de Argumento:** Chaves vazias são rejeitadas (`-EINVAL`);
4. Se as validações passarem, a variável global `cc_key` é atualizada e o contador estatístico `key_changes` é incrementado.

3.6.3 Importância da interface `/proc`

A interface `/proc` desempenha um papel arquitetural crítico no projeto, indo além do simples monitoramento:

- **Inicialização Segura (Secure by Default):**
    - Como o driver é carregado em modo restrito (Modo -1), a interface `/proc/cryptochannel/config` é o único mecanismo capaz de autenticar a operação inicial, obrigando o administrador a definir uma chave antes que qualquer dado possa trafegar.
- **Observabilidade e Diagnóstico:**
    - O arquivo `stats` fornece uma visão granular do comportamento do kernel, permitindo identificar gargalos (através dos contadores de `reads_blocked` e `reads_timeout`) e tentativas de uso indevido (`invalid_ops` e `writes_rejected`).
- **Consistência em Tempo Real:**
    - A implementação garante que alterações de configuração (como a troca de chaves) sejam atômicas e respeitem o estado do buffer. O uso de *mutexes* na interface de configuração impede condições de corrida que poderiam corromper o fluxo de dados criptografados.

3.7 Inicialização e finalização do módulo

A fase de inicialização do módulo é responsável por preparar todos os recursos necessários para que o dispositivo `cryptochannel` funcione corretamente dentro do kernel. Essa lógica está concentrada dentro da função `cc_init()`, que é automaticamente executada no momento em que o módulo é carregado via `insmod`. Da mesma forma, a limpeza

desses recursos ocorre na função `cc_exit()`, acionada quando o módulo é removido por `rmmod`.

3.7.1 Etapas de inicialização (`cc_init`)

A função `cc_init()` segue uma sequência estruturada de passos, cada um relacionado à criação e configuração dos elementos fundamentais do módulo:

```
static int __init cc_init(void)
{
    int ret;
    struct device *dev;

    pr_info("cryptochannel: iniciando carregamento do modulo\n");

    ret = alloc_chrdev_region(&cc_dev, 0, 1, DEVICE_NAME);
    if (ret) return ret;

    cdev_init(&cc_cdev, &cc_fops);
    cc_cdev.owner = THIS_MODULE;
    ret = cdev_add(&cc_cdev, cc_dev, 1);
    if (ret) goto err_unregister;

    cc_class = class_create(DEVICE_NAME);
    if (IS_ERR(cc_class)) {
        ret = PTR_ERR(cc_class);
        goto err_cdev;
    }
    dev = device_create(cc_class, NULL, cc_dev, NULL, DEVICE_NAME);
    if (IS_ERR(dev)) {
        ret = PTR_ERR(dev);
        goto err_class;
    }
```

Figura 15 – Inicialização: Alocação de major/minor e registro do cdev

```
    INIT_KFIFO(cc_fifo);
    /* Zera todas as estatísticas na inicialização */
    memset(&cc_stats, 0, sizeof(cc_stats));

    proc_dir = proc_mkdir(PROC_DIR, NULL);
    if (!proc_dir) {
        ret = -ENOMEM;
        goto err_device;
    }

    if (!proc_create(PROC_STATS, 0444, proc_dir, &proc_stats_ops)) {
        ret = -ENOMEM;
        goto err_proc;
    }
    if (!proc_create(PROC_CONFIG, 0666, proc_dir, &proc_config_ops)) {
        ret = -ENOMEM;
        goto err_proc;
    }

    pr_info("cryptochannel: carregado com sucesso\n");
    return 0;
```

Figura 16 – Inicialização: Criação da classe, device e arquivos proc

1. **Registro do número major/minor**

- O kernel reserva dinamicamente um número *major* para o módulo e associa um *minor* (0). Esse identificador permite que o dispositivo seja reconhecido pelo sistema operacional como um dispositivo de caractere. Caso essa etapa falhe, o módulo não pode ser registrado.

2. **Configuração e registro do `cdev`**

   - A estrutura `cdev` associa o dispositivo às operações definidas em `file_operations`. A partir desse ponto, o kernel passa a redirecionar chamadas como `open()`, `read()` e `write()` para as funções implementadas pelo módulo.

3. **Criação da classe do dispositivo e do nó em `/dev`**

   - A criação da classe permite a integração com o `udev`. A chamada `device_create()` gera efetivamente o arquivo `/dev/cryptochannel`, que representa a interface de entrada e saída utilizada pelos processos no espaço de usuário.

4. **Inicialização do FIFO e Estado Seguro**

   - O buffer circular (FIFO) é inicializado para o gerenciamento de dados. Crucialmente, nesta etapa garante-se que a estrutura de estatísticas (`cc_stats`) esteja zerada e que a variável de controle `cc_mode` esteja definida como **`-1`**. Isso estabelece a política de *Secure by Default*, garantindo que o dispositivo inicie em estado bloqueado até ser configurado explicitamente.

5. **Criação da interface `/proc/cryptochannel`**

   - É criado o diretório `/proc/cryptochannel`, contendo os arquivos `stats`, responsável pela exposição das métricas de desempenho e erro, e `config`, utilizado para a autenticação e configuração dinâmica. A criação bem-sucedida destes arquivos é vital, pois sem eles o dispositivo não pode ser destravado.

6. **Tratamento de erros com desalocação progressiva**

   - Caso qualquer etapa falhe, o módulo executa um processo estruturado de rollback, liberando progressivamente os recursos previamente alocados. Essa abordagem evita vazamentos de recursos no kernel e contribui para a estabilidade do sistema.

```
/* tratamento de erros e limpeza */
err_proc:
    remove_proc_entry(PROC_STATS, proc_dir);
    remove_proc_entry(PROC_CONFIG, proc_dir);
    remove_proc_entry(PROC_DIR, NULL);
err_device:
    device_destroy(cc_class, cc_dev);
err_class:
    class_destroy(cc_class);
err_cdev:
    cdev_del(&cc_cdev);
err_unregister:
    unregister_chrdev_region(cc_dev, 1);
    return ret;
}
```

Figura 17 – Padrão de tratamento de erros com labels (goto)

3.7.2 Finalização do módulo (`cc_exit`)

```
static void __exit cc_exit(void)
{
    /* Remove arquivos e diretórios em /proc */
    remove_proc_entry(PROC_STATS, proc_dir);
    remove_proc_entry(PROC_CONFIG, proc_dir);
    remove_proc_entry(PROC_DIR, NULL);

    /* Remove o dispositivo de caractere e estruturas associadas */
    device_destroy(cc_class, cc_dev);
    class_destroy(cc_class);
    cdev_del(&cc_cdev);
    unregister_chrdev_region(cc_dev, 1);

    pr_info("cryptochannel: descarregado\n");
}
```

Figura 18 – Rotina de saída: Liberação de recursos em ordem inversa

Quando o módulo é removido do kernel por meio do comando

`sudo rmmod cryptochannel_dev`, a função `cc_exit()` é invocada. Sua responsabilidade é desfazer as alocações realizadas na inicialização, seguindo a ordem inversa de criação para evitar erros de dependência.

As seguintes ações são realizadas:

- remoção recursiva do diretório e arquivos em `/proc/cryptochannel`;
- destruição do dispositivo lógico (`device_destroy`);
- destruição da classe do dispositivo (`class_destroy`);

- remoção da estrutura de caractere (`cdev_del`);
- liberação da região de números *major/minor* (`unregister_chrdev_region`).

Esse procedimento assegura que o sistema operacional retorne ao seu estado original, garantindo que nenhum recurso (como memória ou números de identificação) permaneça alocado desnecessariamente.

3.8 Ferramentas de espaço do usuário

3.8.1 Ferramenta de Envio (`produtor.c`)

O programa `produtor` foi desenvolvido para enviar mensagens de texto para o dispositivo `/dev/cryptochannel`. Diferente de comandos genéricos como `echo`, esta ferramenta implementa um tratamento robusto de erros, capaz de interpretar os códigos de retorno específicos definidos no driver.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define DEVICE_PATH "/dev/cryptochannel"
```

Figura 19 – Produtor: Bibliotecas e validação de entrada

```c
int main(int argc, char *argv[]) {
    int fd;
    ssize_t ret;
    char *mensagem;

    if (argc < 2) {
        printf("Uso: %s <mensagem>\n", argv[0]);
        return 1;
    }

    mensagem = argv[1];

    printf("[Produtor] Tentando abrir o dispositivo %s...\n", DEVICE_PATH);
    fd = open(DEVICE_PATH, O_WRONLY); // Abre apenas para escrita
    if (fd < 0) {
        perror("[Produtor] Falha ao abrir o dispositivo");
        return errno;
    }

    printf("[Produtor] Escrevendo mensagem: '%s' (%lu bytes)\n", mensagem, strlen(mensagem));
```

Figura 20 – Produtor: Abertura do dispositivo em modo escrita

```c
    // Tenta escrever no dispositivo
    ret = write(fd, mensagem, strlen(mensagem));

    if (ret < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            fprintf(stderr, "[Produtor] O buffer está cheio.\n");
        }
        else if (errno == ENOKEY) {
            fprintf(stderr, "[Produtor] ERRO: Dispositivo não configurado!\n");
            fprintf(stderr, "   É necessário definir o MODO e a CHAVE antes de usar:\n");
            fprintf(stderr, "   1. echo 'modo=0' > /proc/cryptochannel/config\n");
            fprintf(stderr, "   2. echo 'chave=SUA_SENHA' > /proc/cryptochannel/config\n");
        }
        else {
            perror("[Produtor] Falha na escrita");
        }
    } else {
        printf("[Produtor] Sucesso! %zd bytes escritos.\n", ret);
    }

    close(fd);
    return 0;
}
```

Figura 21 – Produtor: Lógica de escrita e tratamento de erros (ENOKEY/EAGAIN)

O fluxo de execução do programa, conforme ilustrado no código-fonte, segue os seguintes passos:

1. **Validação de Argumentos:** O programa verifica se o usuário forneceu uma mensagem via linha de comando (`argv[1]`).

2. **Abertura do Dispositivo:** Utiliza-se a chamada de sistema `open()` com a flag `O_WRONLY`, indicando que este processo atuará apenas como escritor. O caminho do dispositivo está definido na constante `DEVICE_PATH`.

3. **Escrita e Tratamento de Erros:** O programa tenta enviar a mensagem utilizando a syscall `write()`. O ponto crítico desta implementação é a verificação da variável global `errno` em caso de falha (retorno < 0), permitindo feedback preciso ao usuário:

   - **Erro `ENOKEY`:** Se o driver retornar este erro, o programa identifica que o módulo está no estado "Bloqueado"(Secure by Default) e imprime instruções amigáveis para o usuário configurar o modo e a chave em `/proc`, conforme visto no código.
   - **Erro `EAGAIN`:** Indica que o buffer circular (FIFO) do kernel está cheio e não pode aceitar novos dados no momento.
   - **Outros Erros:** Falhas genéricas de I/O são reportadas via `perror()`.

4. **Finalização:** Em caso de sucesso, o programa exibe a quantidade de bytes escritos e encerra o descritor de arquivo com `close()`.

Essa implementação garante que o usuário não fique "no escuro"caso tente escrever no dispositivo antes de configurá-lo, reforçando a usabilidade do sistema.

3.8.2 Ferramenta de Leitura (`consumidor.c`)

O programa `consumidor` atua como o receptor no sistema de comunicação, encarregado de ler continuamente os dados decifrados provenientes do dispositivo `/dev/cryptochannel`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <signal.h>

#define DEVICE_PATH "/dev/cryptochannel"
#define READ_BUF_SIZE 1024

static volatile int keep_running = 1;

/* Tratamento para fechar o arquivo ao receber Ctrl+C */
void handle_sigint(int sig) {
    keep_running = 0;
}
```

Figura 22 – Consumidor: Tratamento de sinais (Graceful Shutdown) e variáveis globais

```c
int main() {
    int fd;
    ssize_t ret;
    char buffer[READ_BUF_SIZE];

    signal(SIGINT, handle_sigint);

    printf("[Consumidor] Abrindo dispositivo para leitura contínua...\n");
    fd = open(DEVICE_PATH, O_RDONLY);
    if (fd < 0) {
        perror("[Consumidor] Falha ao abrir o dispositivo");
        return errno;
    }

    printf("[Consumidor] Monitorando mensagens... (Pressione Ctrl+C para sair)\n");
```

Figura 23 – Consumidor: Loop principal e leitura bloqueante

```c
    while (keep_running) {
        // Limpa buffer
        memset(buffer, 0, READ_BUF_SIZE);

        // Bloqueia aqui até chegar dados
        ret = read(fd, buffer, READ_BUF_SIZE - 1);

        if (ret < 0) {
            if (errno == EINTR) {
                break; // Interrompido por sinal (Ctrl+C), sai do loop
            } else if (errno == EAGAIN) {
                continue; // Tenta de novo (se fosse non-block)
            } else {
                perror("[Consumidor] Erro na leitura");
                break;
            }
        } else if (ret == 0) {
            // EOF ou nada lido, continua tentando
            continue;
        } else {
            buffer[ret] = '\0';
            printf("[Consumidor Recebeu]: %s\n", buffer);
        }
    }

    printf("\n[Consumidor] Encerrando...\n");
    close(fd);
    return 0;
}
```

Figura 24 – Consumidor: Sanitização do buffer e exibição da mensagem

A implementação deste software prioriza a estabilidade e o controle de fluxo, destacando-se pelos seguintes aspectos técnicos evidenciados no código:

1. **Tratamento de Sinais (Graceful Shutdown):** Diferente de leituras simples que são abortadas abruptamente, o consumidor implementa um manipulador de sinais para `SIGINT` (acionado via `CTRL+C`). Uma variável global `volatile int keep_running` controla o laço principal, garantindo que, ao receber o sinal de interrupção, o programa saia do loop ordenadamente e feche o descritor de arquivo antes de encerrar.

2. **Abertura em Modo Leitura:** O dispositivo é aberto utilizando a flag `O_RDONLY`. Isso garante que o processo tenha permissão apenas para extrair dados do FIFO do kernel, respeitando o princípio do privilégio mínimo.

3. **Leitura Bloqueante em Loop:** O núcleo do programa consiste em um laço `while` que executa a chamada de sistema `read()`.

    - Por padrão, esta operação é **bloqueante**: se não houver dados no buffer do kernel, o processo é colocado em estado de espera (*sleep*) até que o produtor escreva uma nova mensagem.

- O código trata especificamente o erro `EINTR` (Interrupted System Call), essencial para permitir que a chamada bloqueante seja interrompida pelo sinal `SIGINT` sem causar falhas.

4. **Processamento de Dados:** A cada leitura bem-sucedida, o buffer é sanitizado (limpo com `memset`) e a string recebida é terminada com o caractere nulo (`'\0'`) antes de ser exibida no terminal. Isso previne a exibição de lixo de memória de leituras anteriores.

Essa abordagem assegura que o consumidor possa ficar rodando indefinidamente em um terminal, aguardando mensagens de forma eficiente (sem consumir CPU enquanto espera), comportando-se como um verdadeiro daemon de monitoramento.

3.8.3 Script de Automação de Estresse (`teste_stress.sh`)

Para validar a robustez do driver em cenários de alta concorrência — simulando um ambiente real onde múltiplos processos tentam acessar o dispositivo simultaneamente — foi desenvolvido um script em Shell (*Bash*).

```
# Limpa a tela para começar "do zero"
clear

print_header "TESTE DE ESTRESSE: PRODUTOR x CONSUMIDOR"

# 1. Configuração Inicial
echo -e "${YELLOW}[SETUP] Configurando o Driver...${NC}"

if [ ! -w /proc/cryptochannel/config ]; then
    echo -e "${RED}[ERRO] Sem permissão de escrita em /proc/cryptochannel/config${NC}"
    echo "Rode: sudo chmod 666 /proc/cryptochannel/config"
    exit 1
fi

echo "modo=0" > /proc/cryptochannel/config
echo "chave=Segredo123" > /proc/cryptochannel/config

echo -e "${GREEN}[OK]${NC} Modo definido para 0"
echo -e "${GREEN}[OK]${NC} Chave de criptografia configurada"
sleep 1

# 2. Inicia o Consumidor
print_header "ETAPA 1: INICIANDO CONSUMIDOR"
echo -e "${CYAN}[SISTEMA] Iniciando processo Consumidor em Background...${NC}"
./consumidor &
PID_CONSUMIDOR=$!

# Pausa para garantir que o consumidor abriu o arquivo e bloqueou
sleep 1
echo -e "${CYAN}[SISTEMA] Consumidor aguardando dados (Bloqueado no Read)${NC}"
sleep 1
```

Figura 25 – Script de Estresse: Configuração e lançamento de processos em background

```
# 3. Dispara múltiplos Produtores
print_header "ETAPA 2: DISPARANDO 5 PRODUTORES SIMULTÂNEOS"
echo -e "${YELLOW}[AÇÃO] Lançando 5 processos ao mesmo tempo para testar o Mutex...${NC}\n"

# O sleep 0.05 entre eles é opcional, mas ajuda a não "explodir" o texto de uma vez só na tela,
# mantendo ainda a concorrência alta o suficiente para testar o driver.
./produtor "MSG_A: O rato roeu a roupa" &
sleep 0.05
./produtor "MSG_B: do rei de Roma" &
sleep 0.05
./produtor "MSG_C: Sistemas Operacionais" &
sleep 0.05
./produtor "MSG_D: UTFPR Campo Mourao" &
sleep 0.05
./produtor "MSG_E: Teste de Concorrencia" &

# Aguarda os produtores terminarem
wait $!

echo -e "\n${GREEN}[SUCESSO] Todos os produtores finalizaram a escrita.${NC}"

# Pequena pausa para garantir que o consumidor leu as últimas linhas do buffer
sleep 2

# 4. Finalização
print_header "RESULTADO FINAL"

# Mata o consumidor silenciosamente
kill $PID_CONSUMIDOR 2>/dev/null

echo -e "${GREEN}Teste finalizado com sucesso.${NC}"
echo -e "Observe acima que as mensagens não se misturaram."
echo -e "${BLUE}============================================================${NC}\n"
```

Figura 26 – Script de Estresse: Sincronização (wait) e validação dos resultados

Este script não apenas automatiza a execução dos testes, mas atua como um orquestrador de processos, desempenhando as seguintes funções críticas:

1. **Configuração Automática:** Realiza a configuração inicial do ambiente, definindo o modo de operação e a chave criptográfica via `/proc`, garantindo que o driver esteja pronto para receber dados.

2. **Execução Concorrente (Background Jobs):** O script utiliza o operador `&` para lançar múltiplas instâncias do programa `produtor` em segundo plano. Isso força o escalonador do Sistema Operacional a alternar rapidamente entre os processos, criando uma "disputa"(contention) pelo acesso ao arquivo `/dev/cryptochannel`.

3. **Sincronização:** Utiliza o comando `wait` para aguardar o término de todos os processos filhos antes de prosseguir. Isso é essencial para garantir que as estatísticas finais sejam coletadas apenas após o encerramento de todas as operações de escrita.

4. **Auditoria de Resultados:** Ao final, o script compara automaticamente o número de mensagens enviadas com os contadores do kernel, fornecendo um feedback visual imediato (Verde/Vermelho) sobre a integridade do teste.

Essa abordagem permite verificar, de forma reprodutível, se os mecanismos de exclusão mútua (*mutexes*) implementados no kernel estão prevenindo efetivamente condições de corrida (*race conditions*).

3.9 Resultados

Nesta seção são apresentados os resultados obtidos após a compilação, carregamento e utilização do módulo `cryptochannel`. Os testes foram realizados em um ambiente Linux (Ubuntu), utilizando dois mecanismos principais: o dispositivo de caractere criado em `/dev/cryptochannel` e a interface de monitoramento e configuração disponível em `/proc/cryptochannel/`.

3.9.1 Compilação e carga do módulo

A compilação do projeto foi realizada por meio do comando `make`. Diferente de um módulo isolado, o *build system* foi configurado para compilar tanto o módulo de kernel (`cryptochannel_dev.ko`) quanto as ferramentas de espaço de usuário (`produtor` e `consumidor`), garantindo que as versões dos softwares estejam sincronizadas. A saída do processo é ilustrada na Figura 27.

```
alexjr in Documentos/SO/projeto
● → make
--- Compilando Modulo Kernel ---
make -C /lib/modules/6.8.0-88-generic/build M=/home/alexjr/Documentos/SO/projeto modules
make[1]: Entrando no diretório '/usr/src/linux-headers-6.8.0-88-generic'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
  You are using:           gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
  CC [M]  /home/alexjr/Documentos/SO/projeto/cryptochannel_dev.o
  MODPOST /home/alexjr/Documentos/SO/projeto/Module.symvers
  CC [M]  /home/alexjr/Documentos/SO/projeto/cryptochannel_dev.mod.o
  LD [M]  /home/alexjr/Documentos/SO/projeto/cryptochannel_dev.ko
  BTF [M] /home/alexjr/Documentos/SO/projeto/cryptochannel_dev.ko
Skipping BTF generation for /home/alexjr/Documentos/SO/projeto/cryptochannel_dev.ko due to unavailability of vmlinux
make[1]: Saindo do diretório '/usr/src/linux-headers-6.8.0-88-generic'
--- Compilando Produtor ---
gcc -Wall -g produtor.c -o produtor
--- Compilando Consumidor ---
gcc -Wall -g consumidor.c -o consumidor
```

Figura 27 – Processo de compilação do módulo e ferramentas

Após a compilação, o módulo foi carregado no kernel utilizando o comando `sudo insmod cryptochannel_dev.ko`, conforme apresentado na Figura 28.

```
alexjr in Documentos/SO/projeto
● → sudo insmod cryptochannel_dev.ko

[sudo] senha para alexjr:
alexjr in Documentos/SO/projeto took 2,5s
● → sudo dmesg | tail
[13508.343414] usbhid: USB HID core driver
[13508.373115] r8169 0000:01:00.0 enp1s0: Link is Up - 100Mbps/Full - flow control off
[13510.463127] r8169 0000:01:00.0 enp1s0: Link is Down
[13512.137010] r8169 0000:01:00.0 enp1s0: Link is Up - 100Mbps/Full - flow control off
[13514.032133] r8169 0000:01:00.0 enp1s0: Link is Down
[13520.673491] r8169 0000:01:00.0 enp1s0: Link is Up - 100Mbps/Full - flow control off
[13520.932594] r8169 0000:01:00.0 enp1s0: Link is Down
[13522.575151] r8169 0000:01:00.0 enp1s0: Link is Up - 100Mbps/Full - flow control off
[13539.867930] cryptochannel: iniciando
[13539.868265] cryptochannel: carregado com sucesso
```

Figura 28 – Carregamento do módulo no kernel

Os resultados observados após a compilação e carga do módulo indicam que:

- o dispositivo de caractere foi corretamente registrado no kernel;
- o nó `/dev/cryptochannel` foi criado com sucesso;
- O diretório `/proc/cryptochannel` foi inicializado, juntamente com seus arquivos de controle e monitoramento.

3.9.2 Estrutura gerada pelo módulo

Após o carregamento do módulo, o sistema passou a apresentar a seguinte estrutura, conforme ilustrado na Figura 29.

Figura 29 – Estrutura de arquivos criada em /dev e /proc

A presença desses elementos confirma que o módulo foi corretamente inicializado, resultando na criação de:

- um arquivo de dispositivo do tipo caractere em `/dev/cryptochannel`;
- um diretório administrativo em `/proc/cryptochannel`, destinado à configuração e à exposição de estatísticas do módulo.

3.9.3 Testes de configuração

A configuração do modo de operação e da chave criptográfica foi realizada por meio do arquivo `/proc/cryptochannel/config`, conforme ilustrado na Figura 30.

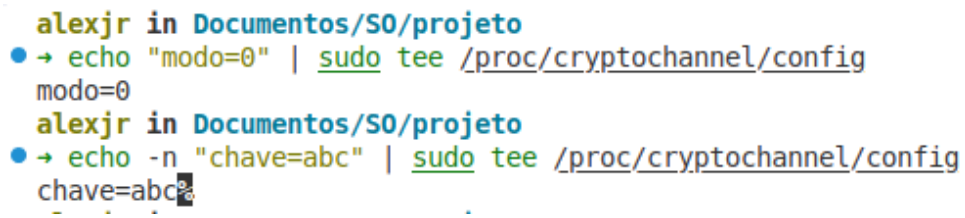

Figura 30 – Configuração do modo e chave via /proc

No modo de operação `modo=0`, o módulo passa a utilizar o algoritmo de criptografia simples baseado em XOR, aplicando a chave definida pelo usuário. Neste exemplo, foi utilizada a chave `abc`, embora qualquer string válida possa ser configurada.

Como o projeto implementa apenas o algoritmo XOR, o modo `0` é atualmente o único modo suportado, sendo obrigatório para a ativação da cifragem. Tentativas de configurar valores diferentes são corretamente rejeitadas pelo módulo, garantindo consistência do estado interno.

3.9.4 Teste Funcional (Manual)

Para validar a lógica básica de criptografia e o protocolo de mensagens de tamanho fixo, foi realizado um teste manual utilizando os programas desenvolvidos:

1. Terminal 1 (Consumidor): Iniciou-se o programa leitor, que aguarda pacotes de 64 bytes.
2. Terminal 2 (Produtor): Foram enviadas 3 mensagens distintas.

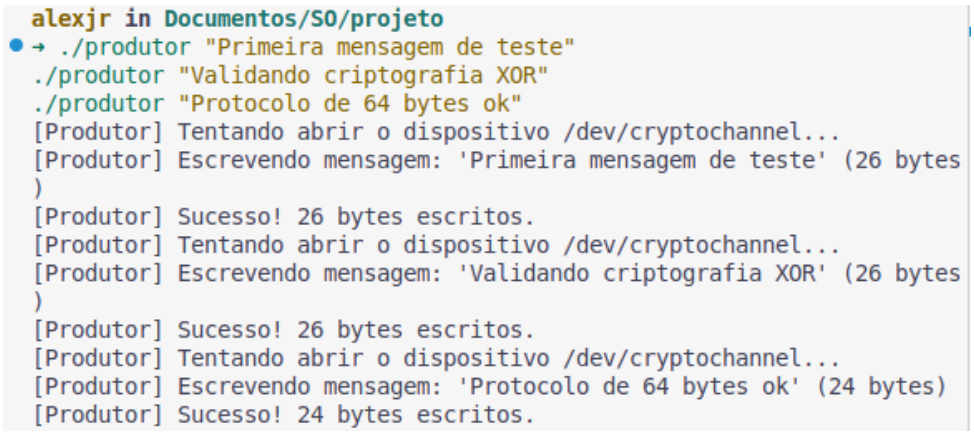

Figura 31 – Envio de mensagens pelo produtor

O terminal do consumidor apresentou a decifragem correta em tempo real, confirmando que a chave configurada no kernel coincidia com a operação de XOR realizada na leitura.

```
alexjr in Documentos/SO/projeto
→ ./consumidor
[Consumidor] Abrindo dispositivo para leitura contínua...
[Consumidor] Monitorando mensagens... (Pressione Ctrl+C para sair)
[Consumidor Recebeu]: Primeira mensagem de teste
[Consumidor Recebeu]: Validando criptografia XOR
[Consumidor Recebeu]: Protocolo de 64 bytes ok
^C
[Consumidor] Encerrando...
```

Figura 32 – Recebimento e decifragem pelo consumidor

3.9.5 Validação das Estatísticas

Após o envio das 3 mensagens de teste, consultou-se o arquivo `/proc/cryptochannel/stats` para validar a precisão dos contadores atômicos.

```
alexjr in Documentos/SO/projeto
→ cat /proc/cryptochannel/stats
Mensagens_trocadas: 3
Bytes_criptografados: 76
Bytes_descriptografados: 76
Modo: 0
Tamanho_chave: 3
Tentativas_de_Leitura: 4
Leituras_timeout: 0
Leituras_sem_dados: 0
Escritas_rejeitadas: 0
Mudancas_de_chave: 1
Tentativas_invalidas: 0
Erros: 0
```

Figura 33 – Estatísticas parciais após o teste manual

Os dados obtidos confirmam a integridade das operações:

- **Mensagens trocadas:** 3 (Corresponde exatamente às três execuções do produtor);
- **Bytes cifrados/decifrados:** 76 bytes.
  - Este valor representa a soma exata dos caracteres úteis enviados:
  - *"Primeira mensagem de teste"* (26 bytes) + *"Validando criptografia XOR"* (26 bytes) + *"Protocolo de 64 bytes ok"* (24 bytes) = 76 bytes.
  - Isso demonstra que o driver gerencia corretamente a memória, contabilizando apenas a carga útil (*payload*) processada.
- **Modo e Chave:** O sistema indicou corretamente o Modo 0 e o tamanho da chave 3 ("abc").

3.9.6 Comparação entre leitura bloqueante e não bloqueante

O módulo `cryptochannel` implementa dois comportamentos distintos para a operação de leitura, determinados pelas flags de abertura do arquivo: leitura bloqueante (padrão) e leitura não bloqueante (`O_NONBLOCK`). A alternância entre esses modos é monitorada pelos contadores atômicos em `cc_stats`.

1. Leitura Bloqueante (Padrão)

Neste modo, utilizado pelo programa `./consumidor` e pelo comando `cat`, quando o FIFO interno está vazio, o processo é colocado em estado de dormência (sleep) através da função:

```
wait_event_interruptible_timeout()
```

Essa função suspende a execução até que dados sejam escritos ou que o temporizador (`READ_TIMEOUT`) expire. O contador `reads_blocked` (exposto como *Tentativas_de_Leitura*) é incrementado a cada bloqueio. Caso o tempo expire sem novos dados, o evento é registrado em `reads_timeout` (*Leituras_timeout*).

Para validar, executou-se:

```
sudo cat /dev/cryptochannel
```

Observou-se que o terminal permaneceu estático (bloqueado) aguardando dados, confirmando a suspensão do processo.

```
alexjr in Documentos/SO/projeto
→ sudo cat /dev/cryptochannel
cat: /dev/cryptochannel: Tempo esgotado para conexão
alexjr in Documentos/SO/projeto took 30,3s
```

Figura 34 – Teste de leitura bloqueante com timeout

2. Leitura Não Bloqueante (`O_NONBLOCK`)

Quando o dispositivo é aberto com a flag `O_NONBLOCK`, a função `cc_read()` verifica o buffer e, caso esteja vazio, retorna imediatamente o erro `-EAGAIN`, sem bloquear o processo chamador.

Esse comportamento foi validado utilizando a ferramenta `dd`, que permite configurar flags de I/O explicitamente:

```
dd if=/dev/cryptochannel of=/dev/null iflag=nonblock count=1
```

```
alexjr in Documentos/SO/projeto took 32,7s
→ sudo dd if=/dev/cryptochannel of=/dev/null iflag=nonblock count=1
dd: erro ao ler '/dev/cryptochannel': Recurso temporariamente indisponível
0+0 records in
0+0 records out
0 bytes copied, 0,00230517 s, 0,0 kB/s
```

Figura 35 – Teste de leitura não-bloqueante com dd

Ao consultar as estatísticas após a execução, o contador `reads_nonblock` (*Leituras_sem_dados*) foi incrementado, comprovando que o kernel desviou o fluxo de execução para o caminho não bloqueante.

3.9.7 Teste de Estresse e Robustez em Concorrência

Para validar a estabilidade do driver em um cenário crítico, foi desenvolvido o script `teste_stress.sh`, que simula um ambiente de alta demanda. O teste foi dividido em três etapas automáticas para verificar a eficácia dos mecanismos de exclusão mútua (*mutexes*) e a integridade dos dados no buffer circular.

Etapa 1: Configuração e Inicialização do Consumidor

Inicialmente, o script realiza a configuração do driver via interface `/proc`, definindo o modo de operação e a chave criptográfica. Em seguida, lança o processo `consumidor` em segundo plano (*background*).

Conforme observado na Figura 36, o consumidor abre o dispositivo e entra imediatamente em estado de bloqueio (Sleep), aguardando a chegada de dados, o que valida o funcionamento correto da `wait_queue` quando o FIFO está vazio.

```
============================================================
   TESTE DE ESTRESSE: PRODUTOR x CONSUMIDOR
============================================================

[SETUP] Configurando o Driver...
[OK] Modo definido para 0
[OK] Chave de criptografia configurada

============================================================
   ETAPA 1: INICIANDO CONSUMIDOR
============================================================

[SISTEMA] Iniciando processo Consumidor em Background...
[Consumidor] Abrindo dispositivo para leitura contínua...
[Consumidor] Monitorando mensagens... (Pressione Ctrl+C par
[SISTEMA] Consumidor aguardando dados (Bloqueado no Read)
```

Figura 36 – Consumidor aguardando início do teste de estresse

Etapa 2: Execução Concorrente (Race Condition Test)

Na fase crítica do teste, o script dispara simultaneamente 5 instâncias do processo `produtor`, cada uma enviando uma mensagem distinta ("MSG_A"a "MSG_E"). O objetivo é forçar uma condição de disputa (*race condition*) pelo acesso ao dispositivo `/dev/cryptochannel`.

A Figura 37 evidencia que, apesar das tentativas de escrita ocorrerem quase no mesmo instante, o driver serializou corretamente o acesso. As mensagens foram gravadas e lidas pelo consumidor de forma atômica, mantendo sua integridade (ex: "O rato roeu a roupa"chegou completo, sem pedaços da mensagem "do rei de Roma"misturados no meio).

```
============================================================
   ETAPA 2: DISPARANDO 5 PRODUTORES SIMULTÂNEOS
============================================================

[AÇÃO] Lançando 5 processos ao mesmo tempo para testar o Mutex...

[Produtor] Tentando abrir o dispositivo /dev/cryptochannel...
[Produtor] Escrevendo mensagem: 'MSG_A: O rato roeu a roupa' (26 bytes)
[Produtor] Sucesso! 26 bytes escritos.
[Consumidor Recebeu]: MSG_A: O rato roeu a roupa
[Produtor] Tentando abrir o dispositivo /dev/cryptochannel...
[Produtor] Escrevendo mensagem: 'MSG_B: do rei de Roma' (21 bytes)
[Produtor] Sucesso! 21 bytes escritos.
[Consumidor Recebeu]: MSG_B: do rei de Roma
[Produtor] Tentando abrir o dispositivo /dev/cryptochannel...
[Produtor] Escrevendo mensagem: 'MSG_C: Sistemas Operacionais' (28 bytes)
[Produtor] Sucesso! 28 bytes escritos.
[Consumidor Recebeu]: MSG_C: Sistemas Operacionais
[Produtor] Tentando abrir o dispositivo /dev/cryptochannel...
[Produtor] Escrevendo mensagem: 'MSG_D: UTFPR Campo Mourao' (25 bytes)
[Produtor] Sucesso! 25 bytes escritos.
[Consumidor Recebeu]: MSG_D: UTFPR Campo Mourao
[Produtor] Tentando abrir o dispositivo /dev/cryptochannel...
[Produtor] Escrevendo mensagem: 'MSG_E: Teste de Concorrencia' (28 bytes)
[Produtor] Sucesso! 28 bytes escritos.
[Consumidor Recebeu]: MSG_E: Teste de Concorrencia

[SUCESSO] Todos os produtores finalizaram a escrita.
```

Figura 37 – Execução concorrente de múltiplos produtores

Isso confirma que o *mutex* implementado na função `cc_write` bloqueou efetivamente o acesso ao buffer enquanto uma escrita estava em andamento, forçando os outros processos a aguardarem sua vez.

Etapa 3: Resultado Final

Ao término da execução, o script verifica se todas as operações foram concluídas sem erros de I/O e se o fluxo de mensagens foi mantido. A mensagem de "SUCESSO"(Figura 38) confirma que o sistema suportou a carga de trabalho sem perda de

dados, travamentos ou inconsistências de memória.

```
============================================================
   RESULTADO FINAL
============================================================

Teste finalizado com sucesso.
Observe acima que as mensagens não se misturaram.
============================================================
```

Figura 38 – Conclusão bem-sucedida do teste de estresse

3.9.8 Análise das estatísticas finais

```
alexjr in Documentos/SO/projeto took 5,4s
→ sudo cat /proc/cryptochannel/stats
Mensagens_trocadas: 8
Bytes_criptografados: 204
Bytes_descriptografados: 204
Modo: 0
Tamanho_chave: 10
Tentativas_de_Leitura: 11
Leituras_timeout: 1
Leituras_sem_dados: 1
Escritas_rejeitadas: 0
Mudancas_de_chave: 2
Tentativas_invalidas: 0
Erros: 0
```

Figura 39 – Estatísticas acumuladas após todos os cenários de teste

Ao final da bateria completa de testes — incluindo a validação manual, testes de flags e o teste de estresse concorrente — o arquivo `/proc/cryptochannel/stats` fornece um registro auditável do comportamento do driver. A análise dos valores apresentados na Figura 39 permite as seguintes conclusões:

1. Consistência sob Carga (Payload) O contador `Mensagens_trocadas` registra um total de **8 operações de sucesso**. Este número representa a soma exata dos dois principais cenários de teste:

- **3 mensagens** do teste funcional manual;
- **5 mensagens** provenientes do teste de estresse (MSG_A a MSG_E).

Os campos `Bytes_criptografados` e `Bytes_descriptografados` totalizam **204 bytes** e são idênticos. A simetria entre esses valores comprova que, mesmo sob a concorrência de

5 produtores simultâneos, o driver não perdeu nenhum byte no buffer circular e garantiu a integridade dos dados durante todo o processo de cifragem e decifragem.

2. Auditoria das Operações de Leitura O contador `Tentativas_de_Leitura` acumula **11 chamadas** à função `read` do driver. A decomposição desse valor valida os mecanismos de controle de fluxo:

- **8 leituras efetivas:** Correspondem à extração das mensagens úteis pelo consumidor;
- **1 leitura com timeout (`Leituras_timeout`):** Resultado do teste de bloqueio, onde o processo consumidor foi colocado para dormir na *wait queue* e acordado pelo temporizador;
- **1 leitura vazia (`Leituras_sem_dados`):** Ocorrida no teste com a flag `O_NONBLOCK`, retornando erro `-EAGAIN` corretamente;
- **1 leitura final:** Tentativa natural de leitura do laço do consumidor ao final do buffer, antes de voltar a dormir.

3. Ciclo de Vida da Configuração O campo `Mudanças_de_chave` apresenta o valor **2**. Isso reflete precisamente as duas etapas de configuração realizadas: a inicialização manual pelo usuário e a reconfiguração automática executada pelo script `teste_stress.sh` antes de disparar os processos, demonstrando que a interface `/proc` funcionou consistentemente em ambas as situações.

4. Estabilidade Os contadores críticos de falha (`Tentativas_invalidas`, `Escritas_rejeitadas` e `Erros`) permaneceram zerados. Isso é o indicador mais forte da robustez do código: mesmo quando múltiplos processos tentaram escrever simultaneamente, o uso correto de *mutexes* preveniu erros de concorrência, rejeições indevidas ou corrupção de estado.

Em suma, as estatísticas comprovam matematicamente que o módulo `cryptochannel` atendeu aos requisitos de atomicidade, segurança e auditoria propostos no projeto.

4 Conclusão

O desenvolvimento do módulo `cryptochannel` permitiu a aplicação prática de conceitos fundamentais da arquitetura de Sistemas Operacionais. A transição da teoria para a implementação de um driver de dispositivo de caractere (*cdev*) evidenciou a complexidade e a responsabilidade envolvidas na execução de código em modo kernel, onde a estabilidade e a eficiência são críticas.

Um dos principais diferenciais técnicos alcançados neste projeto foi a garantia de robustez em cenários de concorrência. A utilização estratégica de *mutexes* para proteger regiões críticas — especialmente durante a rotação de chaves criptográficas — e o uso de variáveis atômicas (`atomic64_t`) para as estatísticas asseguraram que o módulo operasse livre de condições de corrida (*race conditions*), mesmo sob estresse de múltiplos produtores. Além disso, a implementação dual da leitura (bloqueante via *wait queues* e não bloqueante via flag `O_NONBLOCK`) demonstrou conformidade com os padrões POSIX de I/O.

Destaca-se também a arquitetura de segurança adotada. A implementação da política *Secure by Default*, que inicia o dispositivo em estado bloqueado (Modo -1) até que uma configuração explícita seja realizada via `/proc`, simula cenários reais de proteção de dados. Essa separação clara entre o canal de dados (`/dev`) e o canal de controle (`/proc`) provou ser um modelo eficiente de design de software de sistema.

Em suma, o projeto atendeu plenamente aos requisitos, entregando não apenas um canal de comunicação cifrada funcional, mas um sistema resiliente e observável. A atividade consolidou competências essenciais no ciclo de vida de módulos Linux — da compilação e carregamento à depuração — contribuindo significativamente para o amadurecimento técnico no desenvolvimento de software de baixo nível.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. *Linux Device Drivers*. 3. ed. [S.l.]: O'Reilly Media, 2005. https://lwn.net/Kernel/LDD3/. Citado na página 3.

DEVICE Drivers — Linux Kernel Labs. https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html. Citado na página 3.

SALZMAN, P. J.; BURIAN, M.; POMERANTZ, O. *The Linux Kernel Module Programming Guide*. 2016. https://sysprog21.github.io/lkmpg/. Citado na página 3.