МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МЕТЕМАТИКИ И ИНФОРМАТИКИ

Кафедра компьютерных технологий и систем

КУЛЬГАВЫЙ
Иван Сергеевич

РАЗРАБОТКА МОДУЛЯ ЯДРА С КРИПТОГРАФИЧЕСКОЙ ФУНКЦИЕЙ И РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА ВЗАИМОДЕЙСТВИЯ С МОДУЛЕМ В ОПЕРАЦИОННОЙ СИСТЕМЕ LINUX

Дипломная работа

Научный руководитель:
кандидат физ.-мат. наук,
доцент К. В. Василевский

Допущена к защите
«__» _____2025 г.
Зав. кафедрой компьютерных технологий и систем
Доктор педагогических наук, профессор В.В. Казаченок

Минск, 2025

ОГЛАВЛЕНИЕ

РЕФЕРАТ

Дипломная работа, 54 страницы, 1 приложение, 15 источников.

Ключевые слова: ОПЕРАЦИОННАЯ СИСТЕМА, LINUX, МОДУЛЬ ЯДРА, КРИПТОГРАФИЧЕСКАЯ ФУНКЦИЯ, ВСТРАИВАНИЕ В ЯДРО, ШИФРОВАНИЕ.

Объект исследования: процесс статического встраивания модулей в ядро LINUX.

Цель работы: Исследование и реализация встраивания алгоритмов в ядро Linux для повышения эффективности и надежности работы.

Методы исследования: Анализ исходного кода ядра Linux, Разработка и компиляция статических модулей, Тестирование на различных конфигурациях ядра.

Результаты: успешно реализован модуль ядра Linux, содержащий криптографическую функцию шифрования, проведён сравнительный анализ скорости работы одинаковых реализация алгоритма шифрования в пользовательском пространстве и в ядре, доказана повышенная эффективность алгоритма, работающего в пространстве ядра.

РЭФЕРАТ

Дыпломная праца, 54 старонкі, 1 дадатак, 15 крыніц.

Ключавыя словы: АПЕРАЦЫЙНАЯ СІСТЭМА, LINUX, МОДУЛЬ ЯДРА, КРЫПТАГРАФІЧНАЯ ФУНКЦЫЯ, УСТАЎКА Ў ЯДРО, ШЫФРАВАННЕ.

Аб'ект даследавання: працэс статычнай устаўкі модуляў у ядро LINUX.

Мэта працы: паследаванне і рэалізацыя ўбудовы алгарытмаў у ядро Linux для павышэння эфектыўнасці і надзейнасці сістэмы.

Метады даследавання: аналіз зыходнага кода ядра Linux, распрацоўкаі кампіляцыя статычных модуляў, тэставанне на розных канфігурацыях ядра.

Вынікі: паспяхова рэалізаваны модуль ядра Linux з крыптаграфічнай функцыяй шыфравання, праведзены параўнальны аналіз хуткасці работы алгарытма ў карыстальніцкай прасторы і ў ядры, даказана павышаная эфектыўнасць алгарытма, які працуе ў прасторы ядра.

ABSTRACT

Diploma thesis, 54 pages, 1 appendix, 15 sources.

Keywords: OPERATING SYSTEM, LINUX, KERNEL MODULE, CRYPTOGRAPHIC FUNCTION, KERNEL INTEGRATION, ENCRYPTION.

Object of study: the process of statically embedding modules into the Linux kernel.

Purpose of the work: to investigate and implement the integration of algorithms into the Linux kernel to enhance system efficiency and reliability.

Research methods: analysis of the Linux kernel source code, development and compilation of static kernel modules, testing on various kernel configurations.

Results: successful implementation of a Linux kernel module containing an encryption function, comparative analysis of encryption algorithm performance in user space vs. kernel space, demonstrated higher efficiency of the kernel-space implementation.

ВВЕДЕНИЕ

Современные операционные системы и программные комплексы предъявляют всё более высокие требования к информационной безопасности. В условиях постоянного роста числа угроз, связанных с несанкционированным доступом, подменой данных и кибератаками, особую роль играет защита информации с помощью криптографических методов. Одним из критически важных направлений в этом контексте является интеграция криптографических алгоритмов непосредственно в ядро операционной системы.

Ядро Linux, как основа многих дистрибутивов Linux, включая серверные, встраиваемые и мобильные системы, обеспечивает возможность модульного расширения функциональности, включая реализацию механизмов безопасности. Встраивание криптографических алгоритмов непосредственно в ядро позволяет значительно повысить производительность и снизить задержки при обработке данных за счёт исключения лишних переходов между пользовательским и привилегированным пространством. Это особенно важно для систем с жёсткими требованиями к скорости обработки информации — например, в сетевых шлюзах, VPN-серверах, системах шифрования дисков и других инфраструктурных решениях.

Кроме того, реализация криптографических функций в пространстве ядра упрощает организацию централизованного контроля над безопасностью и позволяет унифицировать подход к защите данных во всей системе.

В ходе дипломной работы проводилось теоретическое и практическое исследование темы: **встраивание криптографического алгоритма в ядро Linux статически**. В рамках работы был выполнен анализ архитектуры ядра, изучены встроенные криптографические возможности, исследованы механизмы безопасной интеграции в ядро, а также произведена реализация и тестирование собственного модуля, реализующего криптографическую функцию.

Результаты работы позволяют оценить эффективность внедрения криптографических алгоритмов в ядро, сравнить производительность с пользовательскими реализациями, а также изучить потенциальные риски и пути их минимизации.

ГЛАВА 1
АНАЛИЗ СТРУКТУРЫ ЯДРА LINUX, ЕГО МОДУЛЬНОЙ АРХИТЕКТУРЫ, МЕХАНИЗМОВ ЗАГРУЗКИ МОДУЛЕЙ, МЕХАНИЗМОВ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ В ЯДРЕ

1.1 Общая структура ядра Linux

Ядро Linux представляет собой монолитное ядро, что означает, что все основные функции ОС, включая управление памятью, файловыми системами, сетевыми стеками и драйверами устройств, выполняются в пространстве ядра. Однако, несмотря на монолитность, ядро обладает высокой модульностью за счёт поддержки динамически загружаемых компонентов — **модулей ядра**.
Основные подсистемы ядра Linux включают:

- **Управление процессами** — реализация многозадачности, планирования процессов и обработки сигналов.
- **Управление памятью** — реализация виртуальной памяти, управление страницами и доступ к физической памяти.
- **Файловая система** — абстракция доступа к данным, поддержка множества форматов файловых систем.
- **Сетевая подсистема** — реализация сетевых протоколов и интерфейсов.
- **Подсистема устройств и драйверов** — взаимодействие с аппаратными устройствами через драйверы.

Ядро предоставляет чётко организованные интерфейсы между этими подсистемами и механизмами расширения через модули. Такая структура позволяет разработчикам подключать дополнительные функции, не изменяя основной код ядра, что важно для поддержки широкого спектра устройств и задач.

Модульная архитектура ядра

Модули ядра представляют собой отдельные двоичные объекты, которые могут быть динамически загружены или выгружены в работающую систему без необходимости перезагрузки. Это позволяет:

- Уменьшить размер базового ядра;
- Упростить обновление и разработку отдельных компонентов;
- Позволить гибко конфигурировать систему под специфические задачи.

Каждый модуль представляет собой расширение ядра — например, драйвер устройства, сетевая функциональность или криптографическая реализация. При этом модули могут взаимодействовать с другими частями ядра через экспортируемые символы (функции и переменные), благодаря чему обеспечивается изоляция и независимость.

Формально модуль представляет собой объектный файл (.ko — kernel object), скомпилированный с использованием заголовков ядра и инструментов компиляции (обычно make и kbuild). При загрузке модуль становится частью ядра и работает в его адресном пространстве с тем же уровнем привилегий.

Загрузка и выгрузка модулей

Ядро Linux предоставляет интерфейсы для управления модулями:

- insmod — ручная загрузка модуля;
- rmmod — удаление модуля из ядра;
- modprobe — высокоуровневая утилита, которая автоматически обрабатывает зависимости между модулями;
- lsmod — просмотр загруженных модулей;
- /sys/module — представление модулей в виде файловой системы sysfs.

Когда модуль загружается в систему, выполняются его функции инициализации и регистрации. Модуль может регистрировать обработчики прерываний, интерфейсы устройств, взаимодействие с подсистемами ядра.

Загрузка модуля включает в себя:

1. Проверку совместимости модуля с текущей версией ядра;
2. Разрешение всех символов, на которые ссылается модуль;
3. Выполнение функции инициализации (обычно init_module() или module_init()).

Выгрузка модуля (если он не является критически необходимым и не занят) сопровождается вызовом функции очистки (cleanup_module() или module_exit()), которая освобождает ресурсы [14].

Статическая и динамическая интеграция модулей

Модули ядра могут быть интегрированы в систему двумя способами:

- **Динамически**, во время работы ОС, через механизмы, описанные выше.
- **Статически**, во время компиляции ядра. В этом случае модуль становится частью ядра и компилируется вместе с ним. Это используется в высоконагруженных и встроенных системах, где важна производительность и минимизация зависимости от внешних модулей.

В рамках дипломной работы рассматривается именно **статическая интеграция** криптографического алгоритма, что предполагает его включение на этапе сборки ядра. Это позволяет:

- Повысить надёжность (модуль всегда загружен);
- Снизить накладные расходы на загрузку;
- Более тесно интегрировать алгоритм с другими частями ядра.

1.2 Анализ встроенных криптографических алгоритмов ядра Linux версии 6.8

В ядре Linux реализована специальная подсистема для работы с криптографией — **Crypto API**. Она предоставляет абстрактный интерфейс для использования различных криптографических алгоритмов и их реализаций как из пользовательского пространства, так и внутри самого ядра. Это позволяет разработчикам модулей и подсистем использовать криптографию, не реализуя её заново.

Crypto API поддерживает следующие типы криптографических операций [13]:

- **Блочные шифры** (AES, DES, Blowfish и др.);
- **Потоковые шифры** (ChaCha20, Salsa20 и др.);
- **Хэш-функции** (SHA-1, SHA-256, MD5 и др.);
- **MAC-алгоритмы** (HMAC);
- **Алгоритмы асимметричного шифрования** (RSA, DH, ECDSA и др.);
- **Алгоритмы сжатия данных**;
- **Криптографические генераторы случайных чисел**.

Алгоритмы регистрируются в системе Crypto API и могут вызываться другими частями ядра или модулями через единый интерфейс. При этом система позволяет выбирать между различными реализациями одного алгоритма — например, аппаратной и программной, SIMD-оптимизированной и базовой.

Архитектура работы с Crypto API

Основные структуры, используемые при работе с Crypto API:

- struct crypto_alg — описание криптографического алгоритма;
- struct crypto_tfm — трансформация (конкретный экземпляр алгоритма, связанный с реализацией);
- struct scatterlist — структура для работы с буферами данных.

Процесс использования алгоритма включает следующие этапы:

1. Получение трансформации (создание криптографического контекста) с помощью crypto_alloc_*() — например, crypto_alloc_cipher() или crypto_alloc_shash();
2. Настройка параметров (например, установка ключа);
3. Шифрование/дешифрование данных или хеширование;
4. Освобождение ресурса через crypto_free_*().

Поддерживаемые алгоритмы в версии 6.8

В версии ядра Linux 6.8 присутствует множество встроенных криптографических алгоритмов. Ниже перечислены некоторые из них [5]:

- **Симметричные блочные шифры**:
  - AES (в различных режимах: ECB, CBC, CTR, XTS, GCM);
  - DES и Triple DES;
  - Camellia;
  - Serpent;
  - Twofish.
- **Потоковые шифры**:
  - ChaCha20 (в том числе ChaCha20-Poly1305);
  - Salsa20.
- **Хэш-функции**:
  - MD5;
  - SHA-1;
  - SHA-224, SHA-256, SHA-384, SHA-512;
  - BLAKE2b, BLAKE2s;

    - SM3 (китайский криптографический стандарт).
- **MAC-алгоритмы**:
    - HMAC (на базе всех поддерживаемых хэшей);
    - CMAC.
- **Асимметричные алгоритмы**:
    - RSA;
    - Diffie-Hellman;
    - Elliptic Curve Cryptography (ECC, включая Curve25519 и NIST curves);
    - Ed25519.
- **Прочее**:
    - CRC32, CRC64;
    - Zlib/LZ4/LZ77 компрессия;
    - DRBG и другие генераторы случайных чисел.

В большинстве случаев эти алгоритмы реализованы как модули, которые можно загрузить вручную, либо включить в состав ядра статически при сборке.

Аппаратные оптимизации и реализация через SIMD

Многие алгоритмы имеют несколько реализаций — например, AES может быть реализован через:

- Чисто программную реализацию на C;
- Оптимизированную через SIMD (SSE, AVX, NEON);
- Аппаратную реализацию с использованием инструкций AES-NI (на x86) или аналогов на ARM.

Crypto API автоматически выбирает наиболее подходящую реализацию на основе архитектуры процессора и доступных инструкций. Это даёт значительный прирост производительности при использовании встроенных криптографических механизмов в ядре.

1.3 Анализ моделей интерфейса для разработки модулей ядра

Разработка модулей ядра требует особого подхода, отличающегося от написания пользовательских программ. Модули работают в пространстве ядра, где нет защиты памяти и привычных отладочных инструментов, а ошибки могут привести к панике ядра и перезагрузке системы. Поэтому разработка и тестирование требуют аккуратности, знания системного программирования и архитектуры ядра.

Для разработки модулей используются:

- Язык **C** (с расширениями под GCC);
- Заголовочные файлы ядра (linux/module.h, linux/kernel.h, linux/init.h и др.);
- Система сборки ядра (инструмент kbuild, Makefile).

Базовая структура модуля ядра

Каждый модуль содержит минимум две функции:

- Функцию инициализации (init_module() или определяемую через module_init()), которая вызывается при загрузке модуля;
- Функцию очистки (cleanup_module() или через module_exit()), которая вызывается при его выгрузке.

Пример простого модуля:

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init my_module_init(void) {
    printk(KERN_INFO "Модуль загружен\n");
    return 0;
}

static void __exit my_module_exit(void) {
    printk(KERN_INFO "Модуль выгружен\n");
}

module_init(my_module_init);
module_exit(my_module_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Имя Автора");
MODULE_DESCRIPTION("Пример простого модуля");
```

__init и __exit — позволяют оптимизировать код, удаляя инициализационные функции после загрузки;

MODULE_LICENSE — необходим для корректной работы модуля (ядро может отказать в загрузке, если лицензия не указана);

MODULE_* — метаинформация о модуле (автор, версия и т.д.).

Система сборки kbuild

Система сборки **kbuild** является стандартным механизмом, используемым для компиляции как самого ядра Linux, так и его модулей. Она играет важную роль в процессе разработки модулей ядра, обеспечивая автоматизированную компиляцию, линковку и генерацию необходимых метаданных для корректного взаимодействия модуля с ядром [3].

Общие сведения о kbuild

kbuild — это часть инфраструктуры сборки ядра Linux. Она основана на использовании Makefile-ов и предоставляет разработчику гибкий, но в то же время строгий формат описания, как именно должен собираться модуль или подсистема ядра. Система поддерживает модульную сборку, перекомпиляцию только изменённых компонентов, а также кросс-компиляцию.

Файловая структура и основные файлы

Для сборки модуля ядра с использованием kbuild, разработчик, как правило, создает следующие файлы:

- **Исходный код модуля**, например: my_module.c
- **Файл описания сборки**: Makefile
- В случае более сложных проектов: Kconfig — для интеграции модуля в меню конфигурации ядра (make menuconfig)

Пример простейшего Makefile для внешнего модуля:

```
obj-m += my_module.o

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Здесь:

- obj-m += my_module.o указывает kbuild, что нужно собрать модуль my_module.ko.
- -C /lib/modules/.../build указывает путь к исходникам текущего ядра.
- M=$(PWD) сообщает kbuild, что исходники модуля находятся в текущем каталоге.

Принцип работы kbuild

kbuild работает на основе вложенной структуры Makefile-ов:

- Корневой Makefile в дереве исходников ядра инициализирует процесс сборки.
- Для каждого подкаталога может быть свой Makefile, описывающий, какие файлы нужно компилировать.
- kbuild поддерживает переменные obj-y, obj-m, EXTRA_CFLAGS и другие.

Ключевые переменные:

- obj-y — объекты, собираемые в ядро статически.

- obj-m — модули, собираемые отдельно (с расширением .ko).
- EXTRA_CFLAGS — дополнительные флаги компилятора.

Процесс сборки модуля

1. Разработчик пишет исходный код и Makefile.
2. Вызывается make, который:
    - Загружает переменные окружения, включая путь к дереву исходников ядра.
    - Передаёт управление корневому Makefile ядра, который вызывает нужные подцели.
    - Компилирует .c-файл в .o, затем собирает .ko — загружаемый модуль ядра.

Итогом сборки является файл my_module.ko, который можно загрузить в ядро с помощью insmod или modprobe.

Примеры использования

```
#include <linux/module.h>
#include <linux/kernel.h>

static int __init my_init(void) {
   printk(KERN_INFO "Модуль загружен\n");
   return 0;
}

static void __exit my_exit(void) {
   printk(KERN_INFO "Модуль выгружен\n");
}

module_init(my_init);
module_exit(my_exit);

MODULE_LICENSE("GPL");
```

MODULE_AUTHOR("Автор");

MODULE_DESCRIPTION("Пример модуля");

Сборка:

make

Загрузка модуля:

sudo insmod my_module.ko

Проверка:

dmesg | tail

Удаление:

sudo rmmod my_module

Поддержка версии ядра и зависимости

Важно убедиться, что модуль собирается **для той же версии ядра**, которая установлена в системе. Также kbuild автоматически включает зависимости, такие как:

- Версия компилятора (CC_VERSION)
- Архитектура (ARCH)
- Модули, от которых зависит текущий модуль

Интеграция в дерево ядра

Если модуль разрабатывается как часть ядра, его Makefile и Kconfig добавляются в соответствующие подкаталоги, а затем подключаются в корневые конфигурационные файлы. Это позволяет включать или отключать модуль через make menuconfig [2].

Вывод

Система сборки kbuild — это мощный и гибкий инструмент, необходимый для разработки, тестирования и интеграции модулей ядра Linux. Она обеспечивает стандартизацию процесса сборки, автоматическое управление зависимостями и упрощает кроссплатформенную компиляцию, делая её незаменимой частью инфраструктуры ядра.

Инструменты отладки

Отладка модулей ядра ограничена, так как стандартные отладчики вроде GDB не работают напрямую в пространстве ядра. Однако доступны другие средства:

- **printk()** — аналог printf, выводит сообщения в лог ядра (dmesg);
- **dmesg** — просмотр лога ядра;
- **/proc и /sys** — интерфейсы для взаимодействия с модулем;
- **debugfs** — специальная файловая система для отладки;
- **kgdb, ftrace, perf** — инструменты для глубокого анализа ядра.

Взаимодействие модулей с ядром и другими модулями

Модуль может использовать функции и переменные, экспортированные ядром или другими модулями, ч ерез механизм **экспорта символов** (EXPORT_SYMBOL, EXPORT_SYMBOL_GPL). Это позволяет строить расширяемые подсистемы.

Модули могут регистрироваться в подсистемах ядра:

- В криптографической подсистеме (Crypto API);
- В файловых системах (register_filesystem);
- В сетевой подсистеме (через Netfilter, TCP/IP stack и др.).

Кроме того, модули могут создавать устройства (через misc_register, cdev, net_device и др.), экспортировать интерфейсы в /proc или /sys, и взаимодействовать с пользовательским пространством через IOCTL, netlink или sysfs.

1.4 Механизмы обеспечения безопасности в ядре

Модули ядра выполняются с полными правами ядра, т.е. **в привилегированном режиме** процессора (Ring 0). Это означает, что:

- Они могут обращаться к любой области памяти;
- Имеют доступ ко всем аппаратным ресурсам;

- Могут изменять поведение других компонентов ядра.

Любая ошибка — например, обращение к неверному указателю или гонка данных — может привести к **краху всей системы** (kernel panic) или созданию **уязвимости**, которую может использовать злоумышленник для эскалации привилегий или обхода защиты.

Поэтому безопасность — один из ключевых аспектов при разработке и интеграции модулей.

Механизмы контроля и защиты в ядре

Современные версии ядра Linux, включая 6.8, поддерживают механизм проверки **цифровой подписи модулей** при их загрузке. Это реализуется через конфигурацию ядра:

- Опция CONFIG_MODULE_SIG включает поддержку подписей;
- Модули подписываются приватным ключом при сборке;
- При загрузке ядро проверяет подпись с использованием встроенного публичного ключа.

Если подпись недействительна, загрузка отклоняется. Это важно для защиты от внедрения вредоносных модулей.

При **статической интеграции**, подпись не используется напрямую, но всё равно важна **верификация кода и соблюдение безопасных практик**. **Защита памяти ядра**

Механизмы защиты памяти ядра [6]:

- **KASLR (Kernel Address Space Layout Randomization)** — рандомизация расположения компонентов ядра в памяти, усложняет эксплуатацию уязвимостей;
- **RO-data (Read-Only Data Sections)** — сегменты памяти, которые запрещено модифицировать после инициализации;
- **stack canaries** — защитные значения на стеке, предотвращающие переполнение;
- **SMEP/SMAP (на x86)** — защита от исполнения кода из пользовательской памяти.

Разрабатываемый модуль должен быть совместим с этими механизмами и не нарушать ограничения, наложенные политиками ядра и CPU.

Разделение прав доступа: capabilities

Хотя модули работают в режиме ядра, доступ к ним из пользовательского пространства может быть ограничен через **механизмы capabilities**. Например, вызовы ioctl или чтение/запись в sysfs могут быть доступны только процессам с определёнными правами (CAP_SYS_ADMIN и др.).

Рекомендации по безопасной разработке ядра

При разработке и интеграции криптографических модулей важно соблюдать следующие практики:

- **Минимизация кода** — избегать избыточных операций, не относящихся к криптографии;
- **Проверка входных данных** — валидировать все буферы и параметры, приходящие из пользовательского пространства;
- **Обработка ошибок** — не игнорировать коды ошибок, корректно освобождать ресурсы;
- **Соблюдение границ памяти** — избегать переполнений и выходов за пределы буферов;
- **Использование проверенных API ядра** — не реализовывать вручную функциональность, уже представленную в Crypto API;
- **Аудит и анализ кода** — использовать статический анализ (например, sparse, smatch) и внутренние проверки (checkpatch.pl, Coccinelle);
- **Совместимость с LSM** — не нарушать политики безопасности системы.

Особенности встраивания криптографических алгоритмов

Криптографические модули требуют дополнительного внимания к:

- **Хранению и обработке ключей** — они не должны попадать в лог, дампы памяти или кэш;
- **Потокобезопасности** — криптографический код должен быть безопасным при многопоточном использовании;
- **Энергонезависимому хранению секретов** — избегать оставления ключей в незатираемой памяти;
- **Быстродействию без компромисса безопасности** — оптимизация допустима только при сохранении стойкости.

Особенно важно избегать **побочных каналов** (side-channel attacks), таких как:

- Временные атаки;
- Атаки через энергопотребление или кэш-поведение.

ГЛАВА 2
РЕАЛИЗАЦИЯ ВЫБРАННОГО КРИПТОГРАФИЧЕСКОГО АЛГОРИТМА И МОДУЛЯ ЯДРА

2.1 Выбор алгоритма

В качестве криптографического примитива для реализации был выбран **алгоритм симметричного блочного шифрования belt-block** из национального белорусского криптографического стандарта **СТБ 34.101.31**. Алгоритм входит в семейство **belt**, наряду с другими примитивами стандарта: belt-hash, belt-mac, belt-keywrap.

Причины выбора:

- Оригинальный и малоизвестный алгоритм, что делает его интересным с точки зрения исследовательской задачи;
- Полноценная замена AES в среде, где требуется соответствие локальным нормативам;
- Стандартизован и подробно описан, в том числе с официальными тестами и примерами;
- Возможность использовать его как расширение Crypto API.

Описание алгоритма belt-block

belt-block — это блочный симметричный шифр с:

- Размером блока: 128 бит (16 байт);
- Длиной ключа: 128, 192 или 256 бит;
- Числом раундов: 8, 10 или 12 в зависимости от длины ключа;
- Основой — **SP-сеть** (substitution-permutation), напоминающую структуру AES, но с иным S-блоком и операциями.

Алгоритм включает в себя:

- Раундовую функцию γ, использующую нелинейные подстановки;
- Линейное преобразование θ;

- Механизм чередования (ψ, σ, ρ) между раундами;
- Специфическую генерацию раундовых ключей из главного ключа (key schedule)

Разработка программной реализации belt-block

Для внедрения алгоритма в ядро была разработана его **чистая программная реализация на языке C**.

Основные задачи:

- Реализация всех базовых функций: генерация тактовых ключей, шифрование одного блока, дешифрование одного блока;
- Реализация режимов работы блочного шифра (на этапе тестирования — ECB и CTR);
- Совместимость с Crypto API (структуры, регистрация алгоритма и т. д.).

При разработке использовались следующие инструменты:

- **GCC** — основной компилятор;
- **Kbuild** — система сборки модулей ядра;
- **printk() + dmesg** — отладка внутри ядра;
- **VM с ядром Linux 6.8** — тестовая среда;
- **Проверка по официальным тест-векторам СТБ** — проверка корректности реализации.

Ключевые структуры кода:

- encrypt() — шифрование 16 байт данных.

Промежуточные итоги

На этом этапе был реализован и протестирован:

- Алгоритм belt-block в режиме ECB;
- Интерфейс шифрования;
- Проверка корректности по официальным тест-векторам;
- Отладка модуля с использованием dmesg.

2.2 Интеграция модуля в систему

Следующим этапом стала **интеграция модуля в систему**, что подразумевает его загрузку в ядро и подготовку к тестированию с пользовательского пространства.

Для интеграции модуля использовалась стандартная инфраструктура Kbuild. Исходный код модуля расположен в отдельной директории. Создан следующий Makefile:

```
obj-m := belt.o

KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
    make -C $(KDIR) M=$(PWD) modules

clean:
    make -C $(KDIR) M=$(PWD) clean
```

make

В результате сборки получается файл belt.ko — загружаемый модуль ядра.

Динамическая загрузка модуля

После успешной сборки модуль можно загрузить в работающую систему с помощью insmod:

```bash
sudo insmod belt.ko
```

Проверка загрузки:

```bash
lsmod | grep belt
```

Проверка регистрации алгоритма в Crypto API:

```bash
cat /proc/crypto | grep belt
```

Пример ожидаемого вывода:

```yaml
name         : belt-ecb
driver       : belt-ecb-generic
module       : belt
priority     : 100
```

Если модуль не загрузился, отладка производится с помощью:

```bash
dmesg | tail
```

Для организации взаимодействия между пользовательским пространством и модулем ядра с реализованным алгоритмом belt было решено использовать **символьное устройство**. Такой подход удобен тем, что предоставляет стандартный интерфейс (файловый дескриптор) и позволяет организовать простой протокол обмена данными.

2.3 Реализация интерфейса для взаимодействия с моделем из пользовательского пространства

Для организации взаимодействия между пользовательским пространством и модулем ядра в данной работе было реализовано **символьное устройство**. Это один из наиболее простых и удобных способов обмена данными между приложениями и драйверами в Linux.

Определение и назначение символьного устройства

Символьное устройство (character device) — это тип устройства, который предоставляет поочередный (байтовый) доступ к данным. В отличие от блочных устройств, символьные не буферизуются ядром и позволяют напрямую вызывать обработчики read, write и другие операции [7].

Структура file_operations

Ядро Linux использует структуру file_operations для определения набора операций, которые можно выполнять над устройством. В рамках разработки модуля был реализован минимальный набор функций:

```
static struct file_operations fops = {
   .read = dev_read,
   .write = dev_write,
};
```

Здесь:

- .read — функция, вызываемая при чтении из устройства (например, через read() в пользовательском коде).
- .write — функция, вызываемая при записи в устройство (write()).

При необходимости структура может быть дополнена такими функциями, как open, release, ioctl и другими.

Регистрация устройства

Регистрация символьного устройства осуществляется при инициализации модуля:

```
#define DEVICE_NAME "NTcrypto"


static int __init test_module_init(void) {
   major_num = register_chrdev(0, DEVICE_NAME, &fops);
   if (major_num < 0) {
      printk(KERN_ERR "Device registration failure\n");
      return major_num;
   }


   printk(KERN_INFO "Module installed. Use 'mknod /dev/%s c %d 0'\n",
DEVICE_NAME, major_num);
   return 0;
}
```

Здесь:

- register_chrdev(0, DEVICE_NAME, &fops) — регистрирует устройство с динамически выделяемым **основным номером (major number)**.
- При успешной регистрации в переменной major_num сохраняется назначенный номер.
- При ошибке возвращается отрицательное значение.
- Через printk выводится подсказка для создания символьного файла устройства.

Создание символьного файла

После загрузки модуля в систему (через insmod), символьное устройство необходимо отобразить в файловой системе /dev с помощью команды:

```
sudo mknod /dev/NTcrypto c <major_num> 0
```

где:

- c — указывает, что создается **символьное** устройство;
- <major_num> — это значение, выведенное printk при загрузке модуля.

После этого можно использовать стандартные инструменты (например, echo, cat, собственные утилиты) для взаимодействия с устройством.

Удаление устройства и освобождение номера

При выгрузке модуля необходимо освободить ресурсы:

```
static void __exit test_module_exit(void) {
   unregister_chrdev(major_num, DEVICE_NAME);
   printk(KERN_INFO "Module removed\n");
}
```

Функция unregister_chrdev() снимает регистрацию символьного устройства и освобождает занятый major-номер.

Вывод

Регистрация символьного устройства обеспечивает прозрачный и удобный интерфейс взаимодействия между пользовательским приложением и ядром. Использование структуры file_operations позволяет реализовать конкретную бизнес-логику чтения и записи, а создание файла устройства в /dev делает доступ к модулю возможным через стандартные механизмы операционной системы. Это ключевой шаг в организации доступа к криптографическому функционалу, реализованному в ядре Linux.

Протокол взаимодействия

Для реализации взаимодействия между пользовательским пространством и ядром через символьное устройство был реализован простой, но эффективный **пользовательский протокол обмена данными**. Основная цель — передача параметров для шифрования или дешифрования (режим работы, ключ и данные), а также получение результата обратно в пользовательскую программу.

Описание логики

- **Функция write()** — пользователь передаёт в ядро необходимые параметры: режим работы (шифрование/дешифрование), ключ и путь к файлу или сами данные.

Структура данных

Поскольку в Linux отсутствует автоматическая сериализация структур между пространствами ядра и пользователя, вся передача осуществляется в виде **байтового массива**. На стороне пользователя строка компонуется в определённой последовательности:

- 1 байт — режим работы (0 — шифрование, 1 — дешифрование);
- 64 байта — шестнадцатеричное представление ключа (256 бит);
- оставшаяся часть — путь к файлу или данные, с которыми будет производиться операция.

Реализация write()

Ниже приведён фрагмент кода, реализующий приём данных с пользовательской стороны и дальнейший запуск криптографической обработки:

```
static ssize_t dev_write(struct file *filp, const char __user *buf, size_t count,
loff_t *off) {
   if (count > MAX_PATH_LEN) {
  printk(KERN_ERR "Veru long path\n");
      return -EINVAL;
   }

   if (copy_from_user(file_pat, buf, count)) {
      printk(KERN_INFO "Copy from user error\n");
      return -EFAULT;
   }

   file_pat[43] = '\0';  // Гарантируем строку

   const u8* key = hex_string_to_bytes(k,64);

   ktime_t start, end;
   s64 elapsed_ns;

   start = ktime_get();
   kernel_encrypt_file(file_pat, key);
```

```
    end = ktime_get();

    elapsed_ns = ktime_to_ns(ktime_sub(end, start));

    u64 seconds = elapsed_ns / NSEC_PER_SEC;
    u64 remainder_ns = elapsed_ns % NSEC_PER_SEC;
    printk(KERN_INFO "Время выполнения: %llu.%09llu сек (%lld
наносекунд)\n",
      seconds, remainder_ns, elapsed_ns);
    return count;
}
```

Особенности реализации:

- copy_from_user() — безопасно копирует данные из пространства пользователя. При этом осуществляется проверка на корректность указателей, чтобы избежать исключений в ядре.
- hex_string_to_bytes() — вспомогательная функция, преобразующая строку (в hex-формате) в массив байтов.
- kernel_encrypt_file() — основная функция криптографической обработки данных. В зависимости от переданных параметров выполняет шифрование или дешифрование.
- ktime_get() и ktime_sub() — API ядра для точного измерения времени выполнения операции, вплоть до наносекунд.

Общий сценарий работы:

Модуль ядра получает строку, содержащую директорию файла для шифрования, считывает файл по блокам, вызывает функцию шифрования для каждого блока и в конце выводит в журнал ядра общее время работы алгоритма для заданного файла.

2.4 Тестирование модуля через символьное устройство

Для взаимодействия с символьным устройством /dev/NTcrypto, созданным в рамках разработанного модуля ядра, была реализована утилита командной строки belt_test. Её основная задача — передача пути к файлу, подлежащему криптографической обработке, непосредственно в модуль ядра через системный вызов write().

Ниже представлен исходный код утилиты:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define DEVICE "/dev/NTcrypto"
int main(int argc, char **argv) {
   if (argc != 2) {
      printf("Использование: %s <путь_к_файлу>\n", argv[0]);
      return 1;
   }
   int fd = open(DEVICE, O_WRONLY);
   if (fd < 0) {
      perror("Ошибка открытия устройства");
      return 2;
   }
   // Отправляем путь к файлу в модуль ядра
   if (write(fd, argv[1], strlen(argv[1])) < 0) {
      perror("Ошибка записи");
      gclose(fd);
      return 3;
   }
   close(fd)
   return 0;
}
```

Функциональность:

Утилита получает один аргумент командной строки — путь к файлу, который нужно зашифровать или расшифровать. Затем она открывает

символьное устройство и передаёт путь к файлу с помощью write() — этот путь далее обрабатывается в модуле ядра.

Назначение и удобство:

Такой подход позволяет быстро и просто запускать криптографическую обработку без необходимости интеграции с библиотеками шифрования в пользовательском пространстве. Это полезно при тестировании ядра и измерении производительности, так как позволяет минимизировать накладные расходы и сосредоточиться на производительности самого ядра.

Проверка шифрования

Разработанный модуль был протестирован на различных наборах данных с целью убедиться в корректности реализации алгоритма, стабильности модуля, а также в соответствии с государственными стандартами.

Тестирование различных длин сообщений:

Модуль был протестирован на входных данных различной длины:

- **Кратные блоку (16 байт)** — стандартный случай, когда входная строка совпадает по размеру с длиной блока алгоритма belt-block.
- **Некратные блоку** — важный случай, так как алгоритм должен корректно обрабатывать данные путём добавления паддинга (дополнения).
- **Пустые строки** — проверка корректной обработки краевых случаев.
- **Длинные файлы (> 1 МБ)** — тестирование производительности и отсутствия утечек памяти или сбоев ядра.

Результат: модуль устойчиво и корректно обрабатывал все входные данные, не приводя к краху ядра. Все случаи некратных блоков корректно дополнялись до нужной длины.

Сравнение с тест-векторами из СТБ 34.101.31

СТБ 34.101.31 — официальный белорусский стандарт, описывающий параметры алгоритма belt-block, включая его режимы работы и тестовые векторы (входные данные, ключи и ожидаемые результаты).

Для проверки соответствия:

- Были использованы официальные тест-векторы из приложения к стандарту.

- Модуль шифровал заранее заданные строки и сравнивал полученные байты с эталонными значениями.
- Для каждого вектора сравнивались:
    - Входное сообщение (P)
    - Ключ (K)
    - Результат шифрования (C)
    - Результат расшифрования (P')

Результат: значения, полученные от модуля ядра, полностью совпадали с тестовыми векторами, представленными в СТБ 34.101.31. Это доказывает корректную реализацию алгоритма belt-block и её соответствие стандарту.

Вывод по результатам тестирования

Проведённое тестирование показало:

- устойчивую работу модуля даже на больших объёмах данных;
- корректную реализацию симметричного шифрования;
- полное соответствие результатам официальных векторов, подтверждающее правильность реализации;
- удобство использования интерфейса /dev/NTcrypto через консольную утилиту belt_test.

Это свидетельствует о готовности модуля к использованию в продуктивной среде или как надёжного прототипа для дальнейшего внедрения.

2.5 Интеграция модуля в ядро статически

После успешной динамической загрузки, тестирования и отладки модуля с реализацией алгоритма **belt**, следующим этапом стала его **статическая интеграция в ядро Linux**. Такой подход позволяет включить функциональность модуля напрямую в состав ядра, без необходимости загрузки его вручную через insmod или modprobe.

Исходный файл belt.c был перенесён в директорию ядра, содержащую другие криптографические алгоритмы, например:

linux-6.8/crypto/belt.c

Также в Kconfig и Makefile соответствующего уровня были добавлены строки:

crypto/Kconfig:

config CRYPTO_BELT

bool "Belt block cipher algorithm"

default y

help

Implementation of belt block cipher according to STB 34.101.31.

crypto/Makefile:

obj-$(CONFIG_CRYPTO_BELT) += belt.o

Перед сборкой ядра необходимо было включить поддержку belt в конфигурации:

Конфигурация ядра

Перед сборкой ядра необходимо выполнить его конфигурацию — определить, какие подсистемы, драйверы и модули будут включены в состав ядра, и в каком виде: **встроенными** (статически) или **загружаемыми** (динамически). Для этого используется система настройки ядра, основанная на Kconfig-файлах и интерактивных интерфейсах.

Инициализация конфигурации

Была использована команда:

make oldconfig

Эта команда позволяет взять текущую (или базовую) конфигурацию ядра, например, из файла /boot/config-<version> или .config, и обновить её в соответствии с новой версией исходников ядра. Это удобно, так как не требует ручного прохождения всех параметров с нуля.

Также был использован интерфейс:

make menuconfig

Это интерактивная ncurses-форма, позволяющая вручную просматривать и редактировать параметры ядра. С его помощью был найден и включён интересующий модуль.

Раздел Cryptographic API

Для интеграции разработанного криптографического алгоритма belt-block необходимо перейти в следующую категорию настроек:

-> Cryptographic API

В данном разделе отображается список всех поддерживаемых криптографических алгоритмов и механизмов ядра. Среди них был найден и активирован пункт:

[*] Belt block cipher algorithm

Символ [*] означает, что компонент будет **встроен в ядро статически** — то есть не как отдельный модуль, а как часть самого исполняемого ядра. Это особенно важно для повышения надёжности, безопасности и производительности, так как такой модуль нельзя случайно выгрузить, и он будет инициализирован сразу при старте системы.

Аналогичная запись в файле конфигурации .config выглядит так:

CONFIG_CRYPTO_BELT=y

Если бы модуль собирался как динамический, то запись имела бы вид:

CONFIG_CRYPTO_BELT=m

где m означает, что модуль будет компилироваться как отдельный .ko-файл и загружаться в ядро вручную или по мере необходимости.

Особенности статической интеграции

Статическая интеграция криптографического алгоритма имеет следующие преимущества:

- **Надёжность**: модуль не может быть удалён из памяти во время выполнения;
- **Безопасность**: отсутствует риск подмены или загрузки вредоносного модуля с тем же именем;

- **Производительность**: отсутствует накладной код динамической загрузки и регистрации;
- **Инициализация на старте**: модуль доступен сразу после загрузки ядра, без необходимости ручной или автоматической загрузки.

Однако, это также увеличивает размер ядра и требует пересборки ядра при изменении или обновлении кода модуля.

Проверка конфигурации

Для проверки того, что параметр действительно включён, можно воспользоваться следующей командой:

grep CONFIG_CRYPTO_BELT .config

Если она возвращает CONFIG_CRYPTO_BELT=y, это означает, что модуль belt-block будет встроен в ядро как часть его основной структуры.

Вывод

Этап конфигурации ядра имеет критическое значение для корректной интеграции и последующей сборки модуля. Включение модуля belt-block в раздел Cryptographic API как встроенного компонента (CONFIG_CRYPTO_BELT=y) обеспечивает его наличие в системе сразу после загрузки, что особенно важно в случае, если криптографические функции должны быть доступны на ранних этапах инициализации или в средах с ограниченным доверием к внешним модулям.

Сборка ядра

После завершения настройки параметров ядра (см. пункт 9.2), была выполнена полноценная сборка модифицированного ядра Linux с интегрированным модулем криптографического алгоритма belt-block.

Процесс сборки

Для сборки использовались стандартные утилиты GNU Make и инструменты компиляции, входящие в состав кросс-компилятора (или natively, если ядро собирается под текущую архитектуру). Последовательность команд была следующей:

```
make -j$(nproc)
make modules_install
make install
```

Пояснение к командам:

- make -j$(nproc) — выполняет параллельную сборку ядра, где $(nproc) автоматически подставляет количество доступных ядер процессора. Это существенно ускоряет процесс компиляции, особенно на многоядерных системах.
- make modules_install — устанавливает собранные модули ядра в каталог /lib/modules/<версия_ядра>, включая как штатные модули, так и встроенный модуль с реализацией belt-block, если он не был встроен статически.
- make install — копирует собранное ядро (bzImage), файл конфигурации (config), и системный загрузочный образ (initrd, если используется) в загрузочный раздел, обновляя загрузчик (например, GRUB) автоматически или вручную, в зависимости от дистрибутива.

Время сборки

Время сборки ядра существенно зависит от аппаратной конфигурации машины:

- На современных системах с SSD и 8+ потоками сборка может занимать **от 20 до 40 минут**;
- На менее производительных системах (например, на виртуальных машинах или старом "железе") этот процесс может затянуться **до 2–3 часов**.

Также важную роль играет объём поддерживаемых драйверов и включённых опций: минимальная сборка (с минимальным числом драйверов) может быть выполнена существенно быстрее.

Установка ядра и перезагрузка

После установки ядра:

1. Проверяется, что в /boot/ появился файл нового ядра, например vmlinuz-6.8-custom.
2. Удостоверяемся, что в конфигурации загрузчика (GRUB или др.) добавлена соответствующая запись.
3. Выполняется перезагрузка системы:
   Reboot
4. В меню загрузчика выбирается новая сборка ядра (если автозагрузка не настроена), либо система автоматически запускается с новым ядром.

Загрузка и инициализация модуля

Если криптографический модуль belt-block был встроен **статически** в ядро, то его инициализация происходит **автоматически при загрузке** системы. При этом можно убедиться в его работе, просмотрев системный лог:

dmesg | grep belt

Если же модуль был собран как **отдельный объект (модуль)**, то он загружается во время выполнения следующей командой:

modprobe belt_block

или

insmod belt_block.ko

После чего устройство /dev/NTcrypto становится доступным для взаимодействия.

Вывод

Сборка ядра — важный этап интеграции разработанного модуля, требующий аккуратной настройки и понимания процесса компиляции. Успешная сборка и запуск системы с новым ядром свидетельствуют о корректности внесённых изменений и готовности модуля к практическому использованию. Полученное ядро может быть использовано как основа для дальнейших экспериментов, а также как демонстрация встроенной поддержки специфических криптографических алгоритмов.

2.6 Сравнение производительности алгоритма в ядре и в пользовательском пространстве

Заключительным этапом практической части стало сравнение производительности алгоритма шифрования **belt** в двух вариантах реализации:

- **В ядре Linux** — через разработанный модуль с символьным интерфейсом;
- **В пользовательском пространстве** — с использованием идентичной логики шифрования, реализованной как обычная библиотека на языке C.

Для корректной оценки производительности алгоритма шифрования belt-block, реализованного как в пространстве пользователя, так и в пространстве ядра, была разработана и применена строго формализованная методика измерений. Целью было получить объективные, воспроизводимые и статистически устойчивые результаты, позволяющие сравнить обе реализации по скорости выполнения базовых операций шифрования.

Объём данных

В качестве входных данных использовался массив размером **1 мегабайт** (1 048 576 байт), наполненный **псевдослучайными байтами**. Такой объём был выбран как достаточно репрезентативный: он позволяет выявить стабильные тенденции в производительности, нивелируя флуктуации, возможные на малых размерах ввода. Данные формировались с помощью генератора случайных чисел стандартной библиотеки rand() в пользовательском пространстве и с использованием get_random_bytes() в ядре при необходимости.

Условия идентичности параметров

Для обеспечения **сравнимости** результатов как в user space, так и в kernel space использовались одинаковые параметры алгоритма:

- **Режим шифрования**: ECB (Electronic Code Book) — самый простой блочный режим, не требующий инициализационного вектора (IV), что исключает влияние побочных факторов;
- **Ключ**: заранее зафиксированный 256-битный (32-байтный) ключ, одинаковый для всех тестов;
- **Алгоритм шифрования**: belt-block, реализованный по стандарту СТБ 34.101.31;
- **Архитектура CPU**: x86_64, без активных фоновых задач;
- **Условия нагрузки**: тесты проводились в одиночном пользовательском сеансе, без параллельной загрузки CPU другими приложениями.

Методика измерения времени

Для точного измерения времени выполнения были использованы высокоточные таймеры, соответствующие среде выполнения:

- **В пользовательском пространстве** применялась функция clock_gettime() с флагом CLOCK_MONOTONIC, обеспечивающая наносекундную точность и устойчивость к изменениям системного времени;

Пример вызова:

```
struct timespec start, end;

clock_gettime(CLOCK_MONOTONIC, &start);

encrypt(data, size);
```

clock_gettime(CLOCK_MONOTONIC, &end);

В пространстве ядра использовалась функция ktime_get() и ktime_to_ns() — специализированные средства подсистемы timekeeping ядра Linux, обеспечивающие точный и низкоуровневый контроль времени.

Пример:

```
ktime_t start, end;

start = ktime_get();

kernel_encrypt_file(...);

end = ktime_get();

s64 elapsed_ns = ktime_to_ns(ktime_sub(end, start));
```

Статистическая устойчивость результатов

Каждое измерение производилось **10 раз** для каждого способа шифрования (в ядре и в пользовательском пространстве), чтобы нивелировать влияние возможных случайных скачков нагрузки, прерываний или асинхронных системных событий. Полученные значения времени (в наносекундах) **усреднялись**, и вычислялось стандартное отклонение для оценки стабильности результатов.

Прочие условия тестирования

- **Кэширование**: данные, используемые в тестах, сбрасывались из кэша перед каждым запуском, чтобы минимизировать влияние Page Cache (в пользовательском пространстве использовался posix_fadvise() с POSIX_FADV_DONTNEED);
- **Повторяемость**: на всех этапах обеспечивалась возможность автоматизированного повторного запуска тестов;
- **Система**: тесты проводились на дистрибутиве Ubuntu 22.04 LTS с ядром Linux 6.8, на физическом ПК (не в виртуальной машине), с отключёнными службами энергосбережения и частотного масштабирования (frequency scaling).

Вывод

Такая методика позволила провести корректное сравнение двух реализаций алгоритма belt-block, с высокой точностью оценив влияние уровня

выполнения (ядро против пользовательского пространства) на производительность криптографических операций.

Результаты

В результате серии экспериментальных измерений производительности алгоритма belt-block при реализации в двух различных контекстах — в пространстве пользователя и в пространстве ядра Linux — были получены следующие средние значения времени выполнения операции шифрования на объёме данных в 1 МБ:

Реализация	**Среднее время выполнения (мс)**
Пользовательское пространство	~62 мс
Пространство ядра	~52 мс

Таким образом, реализация в пространстве ядра показала **примерно 15–16% ускорение** по сравнению с аналогичной реализацией в пользовательском пространстве. Это является **предсказуемым и логически обоснованным результатом** по следующим причинам:

1. **Отсутствие переключения контекста.**
   В пользовательской реализации каждый вызов криптографической функции требует перехода из пользовательского пространства в ядро (системный вызов), что влечёт за собой определённые накладные расходы на контекст-переключение.
2. **Прямой доступ к системным ресурсам.**
   Код, работающий в ядре, имеет непосредственный доступ к физической памяти и системным ресурсам, минуя абстракции пользовательского API, что сокращает задержки и увеличивает эффективность работы.

Практическое значение результатов

Указанный прирост производительности может показаться относительно умеренным, однако он **существенен** в ряде сценариев, где:

- Операции шифрования выполняются **массово и непрерывно**, например, в случае шифрования трафика в VPN-сервере, при обработке логов или в криптографических прокси;

- Используются **ограниченные ресурсы**, как в случае встраиваемых или мобильных устройств, где каждая миллисекунда и цикл CPU критичны;
- Требуется **низкая задержка** при обеспечении безопасности: например, в системах реального времени, защитных шлюзах, файрволлах и пр.

Таким образом, даже кажущаяся незначительной экономия времени на одной операции приводит к ощутимому эффекту при масштабировании на большие объёмы или при непрерывной работе.

Риски и ограничения ядровой реализации

Несмотря на демонстрируемые преимущества, реализация криптографических функций в пространстве ядра **имеет ряд особенностей и рисков**, которые необходимо учитывать:

1. **Сложность отладки.**
   Ошибки в пользовательском приложении, как правило, не влияют на стабильность всей системы. В отличие от этого, ошибка в коде ядра (например, выход за границы буфера, неправильный указатель, гонка потоков) может привести к **kernel panic** и полной остановке системы.
2. **Отсутствие гибкости.**
   Ядровой модуль менее гибок в плане обновлений и модификаций: его изменение требует пересборки и перезагрузки системы, в то время как пользовательские библиотеки можно обновлять "на лету".
3. **Повышенные требования к безопасности.**
   Учитывая, что ядро работает с максимальными привилегиями, ошибки безопасности в криптографическом модуле могут **открыть путь для эскалации привилегий**, утечек данных или нарушений целостности ядра.

Вывод

Полученные результаты подтверждают, что реализация криптографического алгоритма belt-block в пространстве ядра Linux:

- **Эффективнее** с точки зрения времени обработки;
- **Целесообразна** в задачах, где скорость критична;
- **Требует повышенного внимания** к безопасности, архитектуре и качеству кода.

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломной работы была решена комплексная задача по проектированию, разработке и интеграции криптографического алгоритма шифрования belt-block, определённого в стандарте **СТБ 34.101.31**, непосредственно в ядро операционной системы **Linux**. Работа включала как теоретические исследования архитектуры ядра и встроенной криптографии, так и практическую реализацию модуля, его тестирование, интеграцию и оценку производительности.

В процессе выполнения работы были достигнуты следующие ключевые результаты:

Анализ архитектуры ядра Linux и Crypto API

На начальном этапе был проведён подробный анализ **архитектуры ядра Linux**, с акцентом на его **модульность**, поддержку **динамической и статической загрузки модулей**, структуру пространства памяти, а также принципы взаимодействия между подсистемами.

Особое внимание было уделено **подсистеме Crypto API**, которая представляет собой расширяемую и модульную архитектуру для работы с криптографическими алгоритмами. Были изучены:

- Слои абстракции Crypto API;
- Алгоритмы, встроенные в ядро Linux 6.8 (включая симметричные шифры, хэш-функции и генераторы случайных чисел);
- Принципы аппаратного ускорения с использованием SIMD-инструкций;
- Механизмы регистрации и использования криптографических трансформов.

Полученные знания позволили понять, как разрабатываются и встраиваются собственные криптографические модули, совместимые с общими интерфейсами ядра.

Разработка и реализация криптографического модуля belt

На основе стандарта **СТБ 34.101.31** был разработан модуль ядра, реализующий блочный симметричный алгоритм **belt-block**. Реализация опиралась на собственную криптографическую библиотеку, адаптированную для использования в ядре Linux с учётом его ограничений (например, отказ от

стандартной библиотеки, работа в контексте ядра, безопасность доступа к памяти).

Модуль включает:

- Примитивы инициализации, обработки блоков, шифрования и дешифрования;
- Обработку входных параметров (режим работы, ключ, путь к файлу);
- Интеграцию с механизмами времени и отладочной печати ядра.

Таким образом, был реализован полнофункциональный низкоуровневый модуль шифрования, способный работать как в качестве загружаемого модуля, так и как часть статически собранного ядра.

Создание интерфейса взаимодействия с модулем через символьное устройство

Для обеспечения взаимодействия между модулем ядра и пользовательским пространством был реализован интерфейс **символьного устройства**, зарегистрированного в системе через вызов register_chrdev().

Были реализованы и протестированы функции read() и write(), которые позволяют:

- Передавать данные и параметры шифрования из пользовательского пространства в модуль;
- Получать результаты обработки обратно в пользовательское пространство.

Данный подход обеспечивает гибкость, модульность и универсальность — утилиты или сторонние программы могут использовать /dev/NTcrypto как интерфейс взаимодействия с шифровальщиком.

Тестирование функциональности модуля

Для проверки работоспособности была разработана утилита **belt_test**, позволяющая отправлять путь к файлу в символьное устройство, запускать процесс шифрования и получать диагностические сообщения. Тестирование проводилось на разных входных данных, с различными размерами блоков и вариантами ключей.

Основные цели тестирования:

- Проверка целостности и корректности реализованного алгоритма;
- Сравнение результата шифрования и дешифрования;

- Проверка симметричности (дефункция шифрования восстанавливает исходные данные);
- Сравнение с эталонными векторами, приведёнными в стандарте.

Результаты тестирования подтвердили корректность и устойчивость реализации.

Интеграция в ядро и сборка

Модуль был встроен в ядро **статически**, через включение параметра CONFIG_CRYPTO_BELT=y в конфигурации ядра в разделе Cryptographic API. Это означает, что модуль загружается автоматически вместе с ядром, не требует ручной инициализации, и не может быть выгружен в процессе работы, что важно для систем, предъявляющих повышенные требования к целостности и безопасности.

Сборка ядра включала:

- Настройку конфигурации через make menuconfig;
- Компиляцию ядра и всех модулей;
- Установку и последующую загрузку в новое ядро с включённым модулем belt.

Оценка производительности и сравнение с пользовательской реализацией

Была проведена серия измерений времени выполнения операций шифрования на объёме 1 МБ, как в пользовательском пространстве, так и в ядре. Для этого использовались:

- В пользовательском пространстве: clock_gettime() с CLOCK_MONOTONIC;
- В ядре: ktime_get() с последующим преобразованием в наносекунды.

Каждое измерение проводилось **10 раз** и усреднялось. Результаты показали, что реализация в ядре работает быстрее: в среднем, **на 15–16%** эффективнее по сравнению с реализацией в пространстве пользователя. Это объясняется:

- Отсутствием накладных расходов на системные вызовы и переключения контекста;
- Оптимизированным доступом к памяти в ядре;
- Отсутствием дополнительных уровней абстракции и буферизации.

Практическая значимость и перспективы

Полученные результаты демонстрируют, что **встраивание криптографических алгоритмов непосредственно в ядро Linux** может быть оправданным решением для следующих категорий задач:

- Обработка защищённых данных в реальном времени;
- Применение в системах, чувствительных к задержкам (встроенные устройства, криптомаршрутизаторы);
- Повышенные требования к безопасности, где нежелателен перенос данных в пользовательское пространство.

В то же время подобный подход требует строгого соблюдения принципов **безопасной разработки ядра**, поскольку ошибки или уязвимости в модуле могут повлиять на стабильность всей системы.

Приобретённые знания и навыки

В процессе выполнения работы был приобретён ценный опыт в следующих областях:

- Низкоуровневое программирование в пространстве ядра;
- Работа с архитектурой и механизмами ядра Linux;
- Использование и расширение Crypto API;
- Разработка символьных устройств и интерфейсов взаимодействия между ядром и пользовательским пространством;
- Сборка и конфигурация ядра Linux;
- Оптимизация производительности и измерение эффективности реализованных решений.

Итог

Таким образом, дипломная работа не только реализует прикладную задачу, но и демонстрирует **возможность эффективной интеграции современных отечественных криптографических алгоритмов** в архитектуру ядра Linux. Результаты могут быть использованы как основа для дальнейших исследований, создания защищённых операционных систем или внедрения в продуктах, где критична производительность и безопасность.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Бовет Д. П., Чезати М. Understanding the Linux Kernel. – 3rd ed., O'Reilly, 2005. – 942 p. (Бовет Д. П., Чезати М. Понимание ядра Linux)

2 Браун Н. Введение в системное программирование на Linux. – М.: ДМК Пресс, 2020. – 302 с.

3 Грибачев К.В. Ядро Linux: описание процесса разработки. – URL: https://habr.com/ru/articles/551966/ (дата обращения: 30.04.2024)

4 Керриск М. Linux API. Исчерпывающее руководство. – М.: ДМК Пресс, 2021. – 1080 с.

5 Кодзима С. Внутреннее устройство Linux. – М.: ДМК Пресс, 2022. – 280 с.

6 Корбет Дж., Рубини А., Хартман Г. Разработка ядра Linux. – М.: Вильямс, 2008. – 448 с.

7 Кузьминский М.Ю. Программирование Linux-драйверов. – М.: ДМК Пресс, 2019. – 320 с.

8 Купер М. Погружение в Linux: от пользователя к разработчику. – М.: Эксмо, 2023. – 512 с.

9 Лав Р. Linux. Системное программирование. – 2-е изд., СПб.: Питер, 2020. – 448 с.

10 Магда Ю.С. Программирование в Linux. Самоучитель. – СПб.: БХВ-Петербург, 2018. – 480 с.

11 Родригес К., Фишер Г., Смолски С. Linux. Администрирование и системное программирование. – СПб.: БХВ-Петербург, 2019. – 1104 с.

12 Стивенс Р. UNIX. Разработка сетевых приложений. – 3-е изд., СПб.: Питер, 2021. – 1056 с.

13 Linux Kernel Documentation. – URL: https://www.kernel.org/doc/html/latest/ (дата обращения: 30.04.2024) (Документация ядра Linux)

14 Love R. Linux Kernel Development. – 3rd ed., Addison-Wesley, 2010. – 440 p. (Лав Р. Разработка ядра Linux)

15 Venkateswaran S. Essential Linux Device Drivers. – Prentice Hall, 2008. – 744 p. (Венкатесваран С. Основные драйверы устройств Linux)

ПРИЛОЖЕНИЕ А

```
#include "linux/init.h"
#include "linux/kernel.h"
#include "linux/module.h"
#include "linux/uaccess.h"
#include "linux/fs.h"
#include "linux/file.h"
#include "linux/ktime.h"
#include "encrypt.h"
#include "linux/types.h"

typedef uint64_t u64;
typedef uint32_t u32;
typedef uint16_t u16;
typedef uint8_t u8;

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ivan");
MODULE_DESCRIPTION("ENCRYPT_MODULE");
MODULE_VERSION("1.0");

#define DEVICE_NAME "NTcrypto"
#define MAX_PATH_LEN 256

static int major_num;
static const u8* k =
"E9DEE72C8F0C0FA62DDB49F46F73964706075316ED247A3739CBA38303A
98BF6";
static char file_pat[MAX_PATH_LEN];

//Таблица подстановки H
static const u16 h_table[16][16] = {

    {0xB1, 0x94, 0xBA, 0xC8, 0x0A, 0x08, 0xF5, 0x3B, 0x36, 0x6D, 0x00, 0x8E,
0x58, 0x4A, 0x5D, 0xE4},
    {0x85, 0x04, 0xFA, 0x9D, 0x1B, 0xB6, 0xC7, 0xAC, 0x25, 0x2E, 0x72, 0xC2,
0x02, 0xFD, 0xCE, 0x0D},
    {0x5B, 0xE3, 0xD6, 0x12, 0x17, 0xB9, 0x61, 0x81, 0xFE, 0x67, 0x86, 0xAD,
0x71, 0x6B, 0x89, 0x0B},
    {0x5C, 0xB0, 0xC0, 0xFF, 0x33, 0xC3, 0x56, 0xB8, 0x35, 0xC4, 0x05, 0xAE,
0xD8, 0xE0, 0x7F, 0x99},
    {0xE1, 0x2B, 0xDC, 0x1A, 0xE2, 0x82, 0x57, 0xEC, 0x70, 0x3F, 0xCC, 0xF0,
0x95, 0xEE, 0x8D, 0xF1},
    {0xC1, 0xAB, 0x76, 0x38, 0x9F, 0xE6, 0x78, 0xCA, 0xF7, 0xC6, 0xF8, 0x60,
0xD5, 0xBB, 0x9C, 0x4F},
    {0xF3, 0x3C, 0x65, 0x7B, 0x63, 0x7C, 0x30, 0x6A, 0xDD, 0x4E, 0xA7, 0x79,
0x9E, 0xB2, 0x3D, 0x31},
    {0x3E, 0x98, 0xB5, 0x6E, 0x27, 0xD3, 0xBC, 0xCF, 0x59, 0x1E, 0x18, 0x1F,
0x4C, 0x5A, 0xB7, 0x93},
    {0xE9, 0xDE, 0xE7, 0x2C, 0x8F, 0x0C, 0x0F, 0xA6, 0x2D, 0xDB, 0x49, 0xF4,
0x6F, 0x73, 0x96, 0x47},
    {0x06, 0x07, 0x53, 0x16, 0xED, 0x24, 0x7A, 0x37, 0x39, 0xCB, 0xA3, 0x83,
0x03, 0xA9, 0x8B, 0xF6},
    {0x92, 0xBD, 0x9B, 0x1C, 0xE5, 0xD1, 0x41, 0x01, 0x54, 0x45, 0xFB, 0xC9,
0x5E, 0x4D, 0x0E, 0xF2},
    {0x68, 0x20, 0x80, 0xAA, 0x22, 0x7D, 0x64, 0x2F, 0x26, 0x87, 0xF9, 0x34,
0x90, 0x40, 0x55, 0x11},
```

```
    {0xBE, 0x32, 0x97, 0x13, 0x43, 0xFC, 0x9A, 0x48, 0xA0, 0x2A, 0x88, 0x5F,
0x19, 0x4B, 0x09, 0xA1},
    {0x7E, 0xCD, 0xA4, 0xD0, 0x15, 0x44, 0xAF, 0x8C, 0xA5, 0x84, 0x50, 0xBF,
0x66, 0xD2, 0xE8, 0x8A},
    {0xA2, 0xD7, 0x46, 0x52, 0x42, 0xA8, 0xDF, 0xB3, 0x69, 0x74, 0xC5, 0x51,
0xEB, 0x23, 0x29, 0x21},
    {0xD4, 0xEF, 0xD9, 0xB4, 0x3A, 0x62, 0x28, 0x75, 0x91, 0x14, 0x10, 0xEA,
0x77, 0x6C, 0xDA, 0x1D}

};


//Перевод 16-ричного символа в биты
static u8 hex_to_bi(const u8 ch)
{
    if (ch >= '0' && ch <= '9')
        return ch - '0';
    if (ch >= 'A' && ch <= 'F')
        return 10 + (ch - 'A');
    if (ch >= 'a' && ch <= 'f')
        return 10 + (ch - 'a');
    return 0;
}

//Конвертация 4 бит в 16-ричный эквивалент
static u8 hex_asc_up(u8 val)
{
    const u8 *hex_upper = "0123456789ABCDEF";
    return hex_upper[val & 0x0F];
}

//конвертация 16-ричной строки в биты
static u8* hex_string_to_bytes(const u8* hex_str,const size_t len)
{
    u8* bytes;
    size_t i;
    u8 val;

    bytes = (u8*)kmalloc(len / 2, GFP_KERNEL);
    if(!bytes)
        printk(KERN_ERR "Memory allocation error");

    for (i = 0; i < len / 2; i++) {
        val = hex_to_bi(hex_str[2*i]) << 4;
        val |= hex_to_bi(hex_str[2*i + 1]);
        bytes[i] = val;
    }
    return bytes;
}

//Конвертация бит в 16-ричную строку
static u8* bytes_to_hex_string(const u8 *data, const size_t len)
{
    u8 *hex_str =  kmalloc(len * 2, GFP_KERNEL);
    if(!hex_str)
        printk(KERN_ERR "Memory allocation error");
    size_t i;

    for (i = 0; i < len; i++) {
        hex_str[i*2] = hex_asc_up(data[i] >> 4);
```

```
        hex_str[i*2+1] = hex_asc_up(data[i] & 0x0F);
    }

    return hex_str;
}

//Функция подстановки H
static inline u16 h(const u16 value){
    return h_table[value>>4][value&0xF];
}

//Побитовое ИЛИ двух слов длины size байт
static u8* xor(const u8* left, const u8* right,const size_t size){
    u8* result = (u8*)kmalloc(size, GFP_KERNEL);
    if(!result)
        printk(KERN_ERR "Memory allocation error");
    size_t i = 0;
    while(i < size){
        *(result+i) = *(left + i) ^ *(right+i);
        ++i;
    }
    return result;
}

//Численное представление для слова длины size байт
static u64 u_hat(const u8* u,const size_t size){
    size_t i = 0;
    u64 result = 0;
    u64 mult = 1;
    while(i<size){
        mult<<=(8*i);
        result += mult*(*(u+i));
        mult=1;
        ++i;
    }
    return result;
}

//U угловые скобки 4 байта
static u8* u_angle_32(const u64 U){
    u32 u_mod = U%0x100000000;
    u8* u = (u8*)kmalloc(4, GFP_KERNEL);
    if(!u)
        printk(KERN_ERR "Memory allocation error");
    u8* u_mod_ptr = (u8*)&u_mod;
    size_t i =0;
    while(i<4){
        *(u+i) = *(u_mod_ptr+i);
        ++i;
    }
    return u;
}

//Shlo
static u8* shlo(const u8* u){
    return u_angle_32(u_hat(u, 4)/2);
}

//Shhi
static u8* shhi(const u8* u){
```

```
    return u_angle_32(u_hat(u, 4) * 2);
}

//Shlo_k
static u8* shlok(const u8* u,const u32 k){
    size_t i =1;
    u8* inter_res;
    u8* result = (u8*)kmalloc(4, GFP_KERNEL);
    if(!result)
        printk(KERN_ERR "Memory allocation error");
    memcpy(result, u, 4);
    while(i++<=k){
        inter_res = shlo(result);
        memcpy(result, inter_res,4);
        kfree(inter_res);
    }
    return result;
}

//Rothi
static u8* rothi(const u8* u){
    u8* sh = shhi(u);
    u8* sl = shlok(u, 31);
    u8* result = xor(sh,sl,4);
    kfree(sh); kfree(sl);
    return result;
}

//Rothi_r
static u8* rothir(const u8* u,const u32 r){
    size_t i =0 ;
    u8* inter_res;
    u8* result = (u8*)kmalloc(4, GFP_KERNEL);
    if(!result)
        printk(KERN_ERR "Memory allocation error");
    memcpy(result,u,4);
    while(i++<r){
        inter_res = rothi(result);
        memcpy(result,inter_res,4);
        kfree(inter_res);
    }
    return result;
}

//Gru
static u8* Gru(const u8* u,const u32 r){
    u8* Hu = (u8*)kmalloc(4, GFP_KERNEL);
    if(!Hu)
        printk(KERN_ERR "Memory allocation error");
    size_t i =0;
    while(i<4){
        *(Hu + i) = h(*(u+i));
        ++i;
    }
    u8* result = rothir(Hu,r);
    kfree(Hu);
    return result;
}

//Слово от суммы 2-х численных представлений
```

```
static u8* box_2_rows(const u8* u, const u8* v){
    u64 uvh = u_hat(u, 4) + u_hat(v, 4);
    return u_angle_32(uvh);
}

//Слово от разницы 2-х численных представлений
static u8* box_1_row(const u8* u, const u8* v){
    if(u_hat(u,4)>=u_hat(v, 4))
        return u_angle_32(u_hat(u,4)-u_hat(v, 4));
    else
        return u_angle_32(0x100000000-(u_hat(v, 4)-u_hat(u, 4)));
}

//Обмен содержимым 2-х адресов памяти
static void swapi(u8 *a, u8 *b, const size_t size) {
    void *temp = kmalloc(size, GFP_KERNEL);
    if(!temp)
        printk(KERN_ERR "Memory allocation error");
    memcpy(temp, a, size);
    memcpy(a, b, size);
    memcpy(b, temp, size);
    kfree(temp);
}

//Шифрование СТБ 34.101.31-2011(шифрование блока)
static u8* encrypt(const u8* X, const u8* Teta){

    //Задание переменных a, b, c, d, e
    u8 *a, *b, *c, *d, *e;
    a = (u8*)kmalloc(4, GFP_KERNEL); b = (u8*)kmalloc(4, GFP_KERNEL); c =
(u8*)kmalloc(4, GFP_KERNEL); d = (u8*)kmalloc(4, GFP_KERNEL), e =
(u8*)kmalloc(4, GFP_KERNEL);
    if(!a || !b || !c || !d || !e)
        printk(KERN_ERR "Memory allocation error");
    memcpy(a,X,4); memcpy(b,X+4,4); memcpy(c,X+8,4);memcpy(d,X+12,4);

    size_t i = 1, j =1;

    //Задание тактовых ключей
    u8* K = (u8*)kmalloc(228, GFP_KERNEL);
    if(!K)
        printk(KERN_ERR "Memory allocation error");
    while(j<57){
        memcpy(K+j*4,Teta+((j-1)%8)*4,4);
        ++j;
    }
    //Указатели для хранения промежуточных результатов
    u8 *result1, *result2, *result3, *result4, *result5;

    while(i < 9){
        result1 = box_2_rows(a,K+(7*i-6)*4);
        result2 = Gru(result1, 5);
        result3 = xor(b,result2,4);
        memcpy(b,result3,4);
        kfree(result1); kfree(result2);kfree(result3);

        result1 = box_2_rows(d,K+(7*i-5)*4);
        result2 = Gru(result1, 21);
        result3 = xor(c,result2,4);
        memcpy(c,result3,4);
```

```
        kfree(result1); kfree(result2);kfree(result3);

        result1 = box_2_rows(b,K+(7*i-4)*4);
        result2 = Gru(result1, 13);
        result3 = box_1_row(a,result2);
        memcpy(a,result3,4);
        kfree(result1); kfree(result2);kfree(result3);

        result1 = box_2_rows(b,c);
        result2 = box_2_rows(result1, K+(7*i-3)*4);
        result3 = Gru(result2, 21);
        result4 = u_angle_32(i);
        result5 = xor(result3,result4,4);
        memcpy(e,result5,4);
        kfree(result1); kfree(result2);kfree(result3);kfree(result4); kfree(result5);

        result1 = box_2_rows(b,e);
        memcpy(b,result1,4);
        kfree(result1);

        result1 = box_1_row(c,e);
        memcpy(c,result1,4);
        kfree(result1);

        result1 = box_2_rows(c,K+(7*i-2)*4);
        result2 = Gru(result1, 13);
        result3 = box_2_rows(d,result2);
        memcpy(d,result3,4);
        kfree(result1); kfree(result2);kfree(result3);

        result1 = box_2_rows(a,K+(7*i-1)*4);
        result2 = Gru(result1, 21);
        result3 = xor(b,result2,4);
        memcpy(b,result3,4);
        kfree(result1); kfree(result2);kfree(result3);

        result1 = box_2_rows(d,K+7*i*4);
        result2 = Gru(result1, 5);
        result3 = xor(c,result2,4);
        memcpy(c,result3,4);
        kfree(result1); kfree(result2);kfree(result3);

        swapi(a,b,4);
        swapi(c,d,4);
        swapi(b,c,4);
        ++i;

    }

    u8* result = (u8*)kmalloc(16, GFP_KERNEL);
    if(!result)
        printk(KERN_ERR "Memory allocation error");
    memcpy(result,b,4);          memcpy(result+4,d,4);          memcpy(result+8,a,4);
memcpy(result+12,c,4);
    kfree(K); kfree(a); kfree(b); kfree(c); kfree(d); kfree(e);

    return result;

}
```

```
//Шифрование файла
static void kernel_encrypt_file(const u8 *input_path, const u8 *key) {
    struct file *input_file;
    loff_t pos = 0;
    const size_t block_size = 16;
    u8 block[block_size];
    ssize_t bytes_read;

    input_file = filp_open(input_path, O_RDONLY, 0);
    if (IS_ERR(input_file)) {
        printk(KERN_ERR "Ошибка открытия файла: %ld\n",
PTR_ERR(input_file));
        return;
    }

    while (bytes_read = kernel_read(input_file, block, block_size, &pos)) {
        if (bytes_read < 0) {
            printk(KERN_ERR "Ошибка чтения файла\n");
            break;
        }

        if (bytes_read < block_size) {
            break;
        }

        u8 *encrypted = encrypt(block, key);
        kfree(encrypted);
    }

    printk(KERN_INFO "Block processed: %lld\n", pos / block_size);
    filp_close(input_file, NULL);
}

// Обработчик записи в устройство
static ssize_t dev_write(struct file *filp, const char __user *buf, size_t count, loff_t
*off) {
    if (count > MAX_PATH_LEN) {
        printk(KERN_ERR "Veru long path\n");
        return -EINVAL;
    }

    if (copy_from_user(file_pat, buf, count)) {
        printk(KERN_INFO "Copy from user error\n");
        return -EFAULT;
    }

    file_pat[43] = '\0';  // Гарантируем строку

    const u8* key = hex_string_to_bytes(k,64);

    ktime_t start, end;
    s64 elapsed_ns;

    start = ktime_get();
    kernel_encrypt_file(file_pat, key);
    end = ktime_get();

    elapsed_ns = ktime_to_ns(ktime_sub(end, start));

    u64 seconds = elapsed_ns / NSEC_PER_SEC;
```

```
    u64 remainder_ns = elapsed_ns % NSEC_PER_SEC;
    printk(KERN_INFO"Время выполнения: %llu.%09llu сек\n",
        seconds, remainder_ns, elapsed_ns);

    return count;
}

// Обработчик чтения (возвращаем статус)
static ssize_t dev_read(struct file *filp, char __user *buf, size_t count, loff_t *off) {
    const char *msg = "Ready\n";
    size_t len = strlen(msg);
    if (copy_to_user(buf, msg, len)) {
        return -EFAULT;
    }
    return len;
}

static struct file_operations fops = {
    .read = dev_read,
    .write = dev_write,
};

//Инициализация модуля
static int __init test_module_init(void){

    major_num = register_chrdev(0, DEVICE_NAME, &fops);
    if (major_num < 0) {
        printk(KERN_ERR"Device registration failure\n");
        return major_num;
    }
    printk(KERN_INFO "Module installed. Use 'mknod /dev/%s c %d 0'\n",
DEVICE_NAME, major_num);

    return 0;

}

//Выгрузка модуля
static void __exit test_module_exit(void){
    printk(KERN_INFO "Test module exit\n");
}

module_init(test_module_init);
module_exit(test_module_exit);
```