



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

*«Модуль ядра для подмены пользовательских данных,
передаваемых на съёмные USB-носители»*

Студент ИУ7-75Б
(Группа)

(Подпись, дата)

Смирнов П. И.
(И. О. Фамилия)

Руководитель курсовой работы

(Подпись, дата)

Рязанова Н. Ю.
(И. О. Фамилия)

2026 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ

Заведующий кафедрой ИУ7

_____ И. В. Рудаков

«__» сентября 2025 г.

З А Д А Н И Е
на выполнение курсовой работы

по теме

Модуль ядра для подмены пользовательских данных, передаваемых на съёмные USB-носители

Студент группы **ИУ7-75Б**

Смирнов Пётр Ильич

Направленность КуР (учебная, исследовательская, практическая, производственная, др.):
учебная.

Источник тематики (кафедра, предприятие, НИР): кафедра.

График выполнения КуР: 25% к 5 нед., 50% к 8 нед., 75% к 11 нед., 100% к 15 нед.

Техническое задание

Разработать модуль ядра для подмены пользовательских данных, передаваемых на USB-накопители с файловой системой FAT32. Использовать механизм kprobes.

Оформление курсовой работы:

Расчетно-пояснительная записка на 40-50 листах формата А4.

Дата выдачи задания «__» сентября 2025 г.

Руководитель НИР

_____ **Н. Ю. Рязанова**

Студент

_____ **П. И. Смирнов**

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 Аналитический раздел	6
1.1 Постановка задачи	6
1.2 Архитектура подсистемы USB в Linux	6
1.3 Анализ основных структур ядра для работы с USB	7
1.3.1 struct User Request Block (URB)	7
1.3.2 struct usb_device	9
1.4 Анализ основных функций ядра для работы с USB	9
1.4.1 usb_submit_urb	9
1.4.2 usb_hcd_link_urb_to_ep	10
1.5 Анализ методов динамической отладки функций ядра	10
1.5.1 kprobes	10
1.5.2 Livepatch	11
1.5.3 Выбор метода динамической отладки функций ядра . . .	12
1.6 Файловая система FAT32	12
1.7 Выводы	15
2 Конструкторский раздел	16
2.1 IDEF0 диаграммы	16
2.2 Схема алгоритма фильтрации перехваченных URB	16
2.3 Схема алгоритма обработки URB	17
2.4 Схема алгоритма изменения данных в URB	18
3 Технологический раздел	20
3.1 Выбор языка и среды программирования	20
3.2 Конфигурация USB-накопителя	20
3.3 Функция проверки буфера на соответствие таблице FAT	20
3.4 Функции, вызываемые при загрузке и выгрузке модуля	21
3.5 Функция обработки структуры URB	22
3.6 Функция-обработчик точки останова	23
3.7 Makefile	23

4 Исследовательский раздел	25
4.1 Исследование работы программы	25
4.1.1 Тест №1 — текстовый файл	25
4.1.2 Тест №2 — директория с поддиректориями и файлами . .	26
4.1.3 Тест №3 — файл PDF	27
4.2 Выводы	27
ЗАКЛЮЧЕНИЕ	28
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	29
ПРИЛОЖЕНИЕ А	30

ВВЕДЕНИЕ

Предотвращение утечек данных через USB-устройства является одной из важнейших задач Data Leak Prevention программно-аппаратных комплексов [1]. Основными механизмами для решения этой задачи в ядре Linux являются драйверы устройств и перехват функций ядра. Данная работа посвящена созданию и исследованию загружаемого модуля ядра для подмены данных, передаваемых на USB-накопители, с использованием перехвата управления у функций.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу необходимо разработать модуль ядра для подмены пользовательских данных, передаваемых на USB-накопители. Подмена данных осуществляется изменением значения каждого байта данных, с использованием XOR операции. Для достижения поставленной цели необходимо решить следующие задачи:

1. провести обзор функций и структур ядра, представляющих возможность реализации разрабатываемого модуля;
2. провести анализ методов динамической отладки функций ядра;
3. провести анализ работы файловой системы FAT32;
4. реализовать и протестировать модуль.

1.2 Архитектура подсистемы USB в Linux

Основные компоненты, реализующие работу с USB-устройствами [2]:

1. драйвер устройства;
2. ядро USB (USB Core) – общий код, управляющий всей подсистемой;
3. драйвер хост-контроллера (HCD) – реализует программно-аппаратное взаимодействие.

Схема взаимодействия драйверов USB:

1. драйвер устройства создает или получает запрос на передачу информации;
2. драйвер инициализирует запрос всей необходимой информацией:
 - тип запроса (чтение, запись, управление);
 - адрес устройства и номер конечной точки (endpoint);
 - указатель на буфер данных;

- размер буфера данных;
 - функция обратного вызова (callback) – которая будет вызвана, когда операция завершится.
3. драйвер отправляет запрос в ядро USB;
 4. ядро и HCD выполняют необходимую низкоуровневую работу, чтобы выполнить этот запрос на физической шине USB;
 5. когда операция завершена (данные получены, отправлены или произошла ошибка), HCD вызывает функцию обратного вызова для этого запроса, чтобы уведомить драйвер устройства о результате.

1.3 Анализ основных структур ядра для работы с USB

Для управления внешним накопителем, таким как USB-устройство в операционной системе представлено несколько структур.

1.3.1 struct User Request Block (URB)

URB — структура, инкапсулирующая запрос на передачу данных, передаваемый драйвером USB-устройства низкоуровневому драйверу хост-контроллера. Каждая операция ввода-вывода через USB шину оформляется в виде отдельного URB, что обеспечивает единый интерфейс для работы с различными типами USB-передач.

Типы передач, поддерживаемые URB

URB поддерживает все четыре типа передач, определенные в спецификации USB [3]:

1. управляющие послыки (Control Transfers), используемые для конфигурирования во время подключения и в процессе работы для управления устройствами.
2. сплошные передачи (Bulk Data Transfers) сравнительно больших пакетов без жестких требований ко времени доставки. Пакеты имеют поле данных

размером 8, 16, 32 или 64 байт. Приоритет этих передач самый низкий, они могут приостанавливаться при большой загрузке шины.

3. прерывания (Interrupt) - короткие передачи, имеют спонтанный характер и должны обслуживаться не медленнее, чем того требует устройство.
4. изохронные передачи (Isochronous Transfers) - непрерывные передачи в реальном времени, занимающие предварительно согласованную часть пропускной способности шины и имеющие заданную задержку доставки.

Основные поля структуры URB [4]

- `struct usb_device *dev` – представление устройства USB;
- `int status` – статус выполнения;
- `void *transfer_buffer` – указатель на область данных для передачи;
- `u32 transfer_buffer_length` – размер передаваемого буфера;
- `struct scatterlist *sg` – указатель на начало списка разрозненных буферов для передачи;
- `int num_sgs` – длина списка разрозненных буферов;
- `u32 actual_length` – фактически переданное количество байт;
- `unsigned int transfer_flags` – флаги передачи. Флаг `URB_NO_TRANSFER_DMA_MAP` сообщает драйверу хост-контроллера, что драйвер устройства поддерживает DMA (Direct Memory Access);
- `dma_addr_t transfer_dma` – физический адрес передаваемых данных. Если установлен флаг `URB_NO_TRANSFER_DMA_MAP` драйвер устройства сообщает, что он предоставил этот адрес DMA, который драйвер хост-контроллера должен использовать предпочтительнее, чем `transfer_buffer`;
- `usb_complete_t complete` – функция обратного вызова. Используется для освобождения или повторной отправки URB.

1.3.2 struct usb_device

Структура `usb_device` отвечает за представление USB-устройства в ядре.

Основные поля структуры `usb_device`

- `int devnum` – номер устройства в системе. Уникальный идентификатор, присваиваемый устройству при его подключении;
- `char devpath[16]` – путь к устройству в дереве USB. Строка, представляющая путь от корневого хаба до устройства;
- `struct usb_device_descriptor descriptor` – содержит стандартный дескриптор устройства (идентификатор производителя, идентификатор продукта, версия устройства и т.д.);
- `enum usb_device_state state` – текущее состояние устройства;
- `unsigned can_submit_urbs` – флаг, указывающий, можно ли отправлять URBs.

1.4 Анализ основных функций ядра для работы с USB

Для работы с устройствами USB в ядре существует множество функций, описанных в файлах `/drivers/usb/core/urb.c`, `/drivers/usb/core/usb.c`, `/drivers/usb/core/hcd.c`.

1.4.1 usb_submit_urb

Функция имеет следующую сигнатуру:

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags).
```

Функция отправляет запрос на передачу, передавая URB, описывающее этот запрос, в подсистему USB. Завершение запроса идентифицируется позднее, асинхронно, при вызове обработчика завершения. URBs могут отправляться конечным точкам до завершения предыдущих, чтобы минимизировать задержки. Таким образом очередь на конечной точке никогда не будет пустой. Такая организация позволяет максимально эффективно использовать

полосу пропускания, позволяя USB контроллерам начинать работу с более поздними запросами до того, как программное обеспечение драйвера завершит обработку более ранних запросов.

В случае успеха функция возвращает 0, иначе – отрицательный номер ошибки.

1.4.2 `usb_hcd_link_urb_to_ep`

Функция имеет следующую сигнатуру:

```
int usb_hcd_link_urb_to_ep(struct usb_hcd *hcd, struct urb
*urb).
```

Функция добавляет URB в очередь конечной точки. Принимает структуру хост-контроллера, на который был отправлен URB и сам URB. Драйверы хост-контроллера должны вызывать эту процедуру в своем методе `enqueue()`. Возвращает значение 0, если ошибки нет, в противном случае возвращается отрицательный код ошибки (в этом случае метод `enqueue()` должен завершиться ошибкой).

1.5 Анализ методов динамической отладки функций ядра

В ядре Linux существует несколько механизмов отладки функций ядра.

1.5.1 `kprobes`

Kprobes – это средство динамической отладки ядра, позволяющее ставить breakpoints на доступные для записи участки памяти и самостоятельно обрабатывать их. Описывается структурой `kprobes`, имеющей следующие поля:

- `const char *symbol_name` – имя функции, на которую ставится breakpoint;
- `kprobe_pre_handler_t pre_handler` – указатель на функцию обработки точки останова до выполнения основной функции;
- `kprobe_post_handler_t post_handler` – указатель на функцию обработки точки останова после выполнения основной функции;

- `kprobe_opcode_t *addr` – адрес функции, на которую ставится breakpoint. Должно быть указано либо имя функции, либо её адрес, если заполнены оба поля, то регистрация kprobe вернёт ошибку -EINVAL [5].

Добавление и удаление точек останова выполняют следующие функции:

Листинг 1.1 – Функции добавления и удаления kprobes

```
#include <linux/kprobes.h>
int register_kprobe(struct kprobe *kp);
int register_kretprobe(struct kretprobe *rp);
void unregister_kprobe(struct kprobe *kp);
void unregister_kretprobe(struct kretprobe *rp);
```

1.5.2 Livepatch

Livepatch – механизм для исправление критических уязвимостей безопасности и критических ошибок в системах с высокими требованиями к доступности. Основан на подмене целых функций ядра на их исправленные версии во время выполнения системы.

Структура `klp_func` описывает подменяемую функцию:

- `const char *old_name` – имя оригинальной функции;
- `void *new_func` – указатель на исправленную функцию;
- `struct list_head stack_node` – узел списка для отслеживания вызовов функции;
- `bool patched` – флаг, указывающий, применён ли патч к данной функции.

Структура `klp_object` описывает модуль, содержащий функции для замены:

- `const char *name` – имя модуля;
- `struct module *mod` – указатель на структуру модуля;
- `struct klp_func *funcs` – список функций для патчинга;

- `bool patched` – флаг, указывающий, применён ли патч ко всем функциям объекта.

Структура `klp_patch` описывает единый патч, который может включать несколько объектов:

- `struct list_head list` – узел для включения в глобальный список патчей;
- `struct klp_object *objs` – объекты, входящие в патч;
- `struct module *mod` – указатель на модуль, реализующий патч;
- `bool enabled` – флаг активности патча;
- `bool replace` – флаг, указывающий, должен ли патч заменять предыдущие.

Для включения патча необходимо вызвать следующую функцию:

```
int klp_enable_patch(struct klp_patch *patch)
```

Она инициализирует структуру `klp_patch`, выполняет необходимый поиск символов и перемещение кода, регистрирует исправленные функции с помощью `ftrace`.

1.5.3 Выбор метода динамической отладки функций ядра

Для реализации поставленной задачи `Livepatch` избыточен, так как необходимо не исправлять ошибку в ядре, а добавлять новую функциональность. К тому же подменённая функция вызывалась бы для всех USB устройств, а не для конкретной флешки, что невыгодно и может нарушить работу других устройств. Единственным вариантом остаётся `kprobes`.

1.6 Файловая система FAT32

Файловая система получила название от одноимённой таблицы размещения файлов – File Allocation Table, FAT. 32 в названии указывает на размер элемента таблицы (32 бита). В FAT32 четыре старших двоичных разряда зарезервированы и игнорируются в процессе работы операционной системы [6].

На рисунке 1.1 изображена структура раздела FAT:



Рисунок 1.1 – Структура раздела FAT32

В файловой системе FAT дисковое пространство логического раздела делится на две области – системную и область данных. Системная область создается и инициализируется при форматировании, а впоследствии обновляется при манипулировании файловой структурой.

Область данных логического диска содержит файлы и каталоги, подчиненные корневому, и разделена на участки одинакового размера – кластеры. Кластер может состоять из одного или нескольких последовательно расположенных на диске секторов. Число секторов в кластере должно быть кратно степени двойки и может принимать значения от 1 до 64.

Каждому кластеру соответствует элемент таблицы FAT, содержащий

информацию о том, свободен данный кластер или занят данными файла. Если кластер занят под файл, то в соответствующем элементе таблицы размещения файлов указывается адрес кластера, содержащего следующую часть файла. Номер начального кластера, занятого файлом, хранится в элементе каталога, содержащего запись об этом файле. Последний элемент списка кластеров содержит признак конца файла. Первые два элемента FAT являются резервными.

В FAT32 корневой каталог может быть расположен в любом месте области данных раздела и иметь произвольный размер. Структура элемента каталога файлов показана на рисунке 1.2. Элемент начинается с 11-байтного поля, содержащего «короткое имя» файла, по которому операционная система обычно осуществляет поиск файла в каталоге. «Короткое имя» состоит из двух полей: 8-байтного поля, содержащего собственно имя файла и 3-байтного поля, содержащего расширение. Разделительная точка между именем и расширением файла не хранится в структуре данных.

Смещение	Размер (байт)	Содержание
0x00	11	Короткое имя файла.
0x0B	1	Атрибуты файла.
0x0C	1	Зарезервировано для Windows NT. Поле обрабатывается только в FAT32.
0x0D	1	Поле, уточняющее время создания файла (содержит десятки миллисекунд). Поле обрабатывается только в FAT32.
0x0E	1	Время создания файла. Поле обрабатывается только в FAT32.
0x10	2	Дата создания файла. Поле обрабатывается только в FAT32.
0x12	2	Дата последнего обращения к файлу для записи или считывания данных. Поле обрабатывается только в FAT32.
0x14	2	Старшее слово номера первого кластера файла. Поле обрабатывается только в FAT32.
0x16	2	Время выполнения последней операции записи в файл.
0x18	2	Дата выполнения последней операции записи в файл.
0x1A	2	Младшее слово номера первого кластера файла.
0x1C	4	Размер файла в байтах.

Рисунок 1.2 – Структура элемента каталога

4 бит в поле атрибутов файла, равный единице, определяет файл как каталог. Если первый байт короткого имени равен 0xE5, то элемент каталога свободен и его можно использовать при создании нового файла.

1.7 Выводы

В результате проведённого анализа был выбран механизм динамической отладки kprobes для реализации модуля ядра. Была рассмотрена архитектура подсистемы USB в Linux, основные структуры и функции работы и представления USB, а также изучено строение файловой системы FAT32, на которой работает USB-накопитель.

2 Конструкторский раздел

2.1 IDEF0 диаграммы

На рисунках 2.1 и 2.2 представлены IDEF0 диаграммы нулевого и первого уровней.

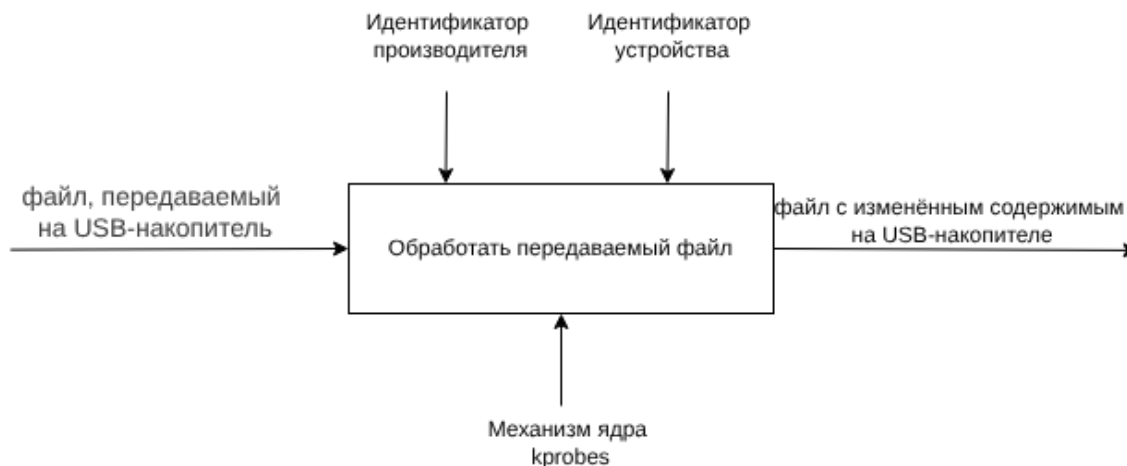


Рисунок 2.1 – Диаграмма IDEF0 нулевого уровня

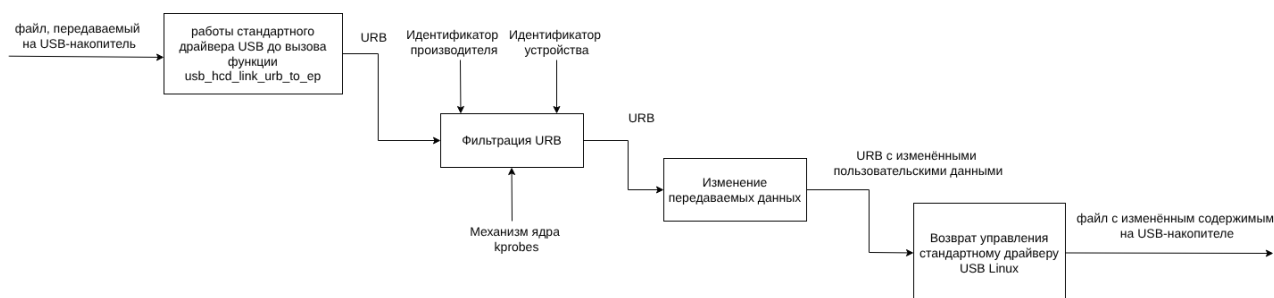


Рисунок 2.2 – Диаграмма IDEF0 первого уровня

2.2 Схема алгоритма фильтрации перехваченных URB

На рис. 2.3 представлена схема работы алгоритма фильтрации перехваченных URB.

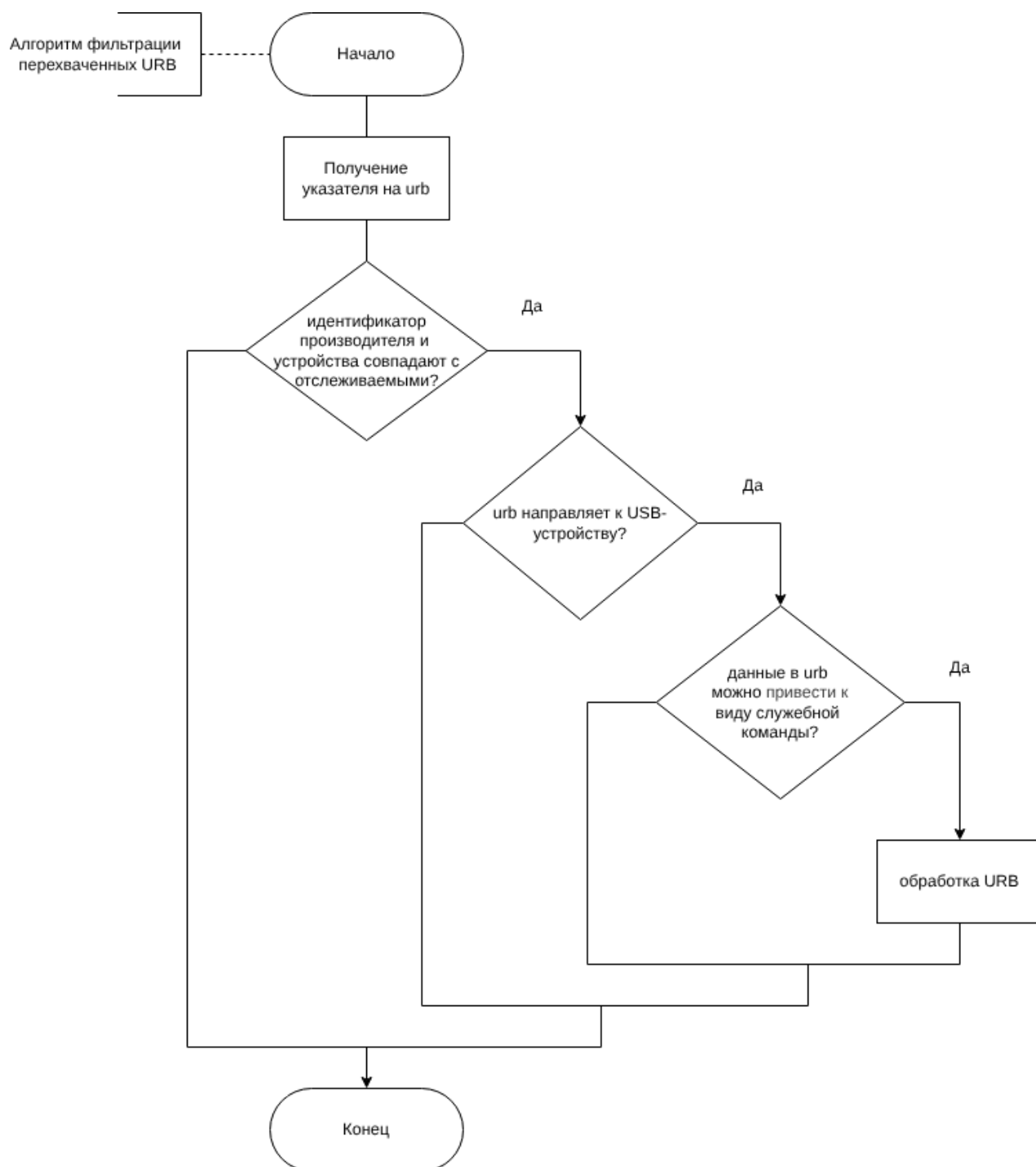


Рисунок 2.3 – Алгоритм фильтрации перехваченных URB

2.3 Схема алгоритма обработки URB

На рис. 2.4 представлена схема работы алгоритма обработки перехваченного URB.

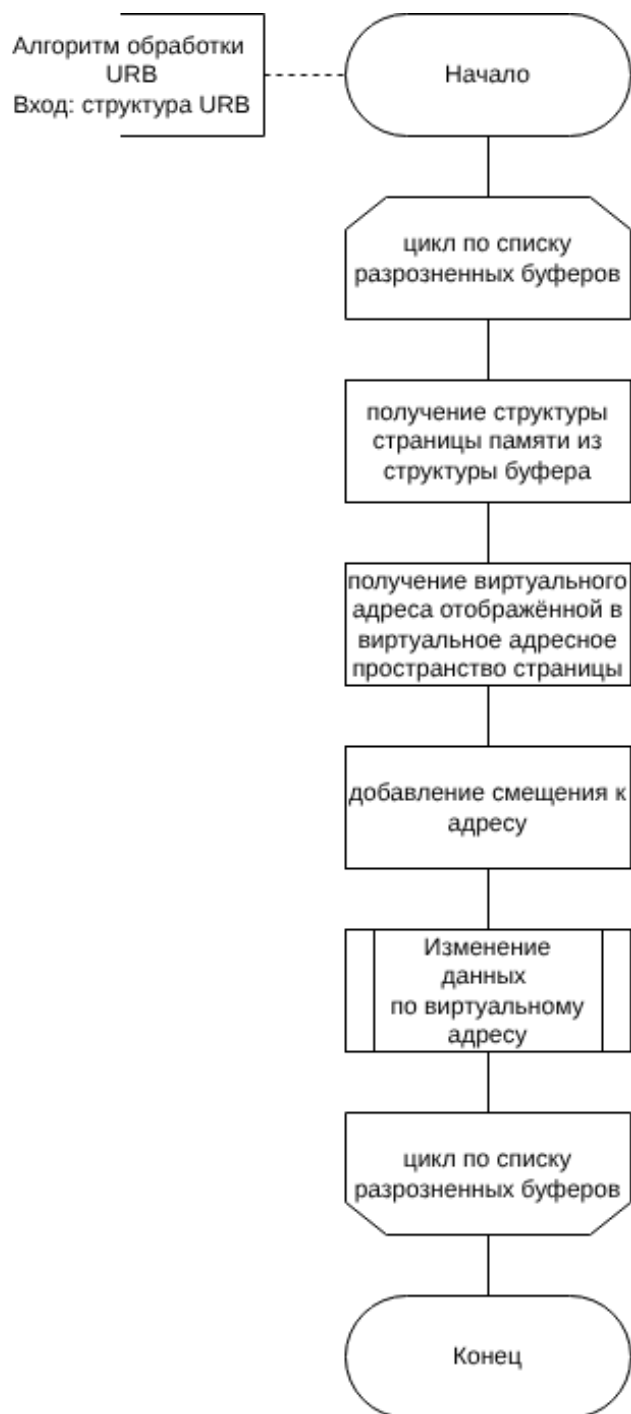


Рисунок 2.4 – Алгоритм обработки перехваченного URB

2.4 Схема алгоритма изменения данных в URB

На рис. 2.5 представлена схема работы алгоритма изменения данных в URB.

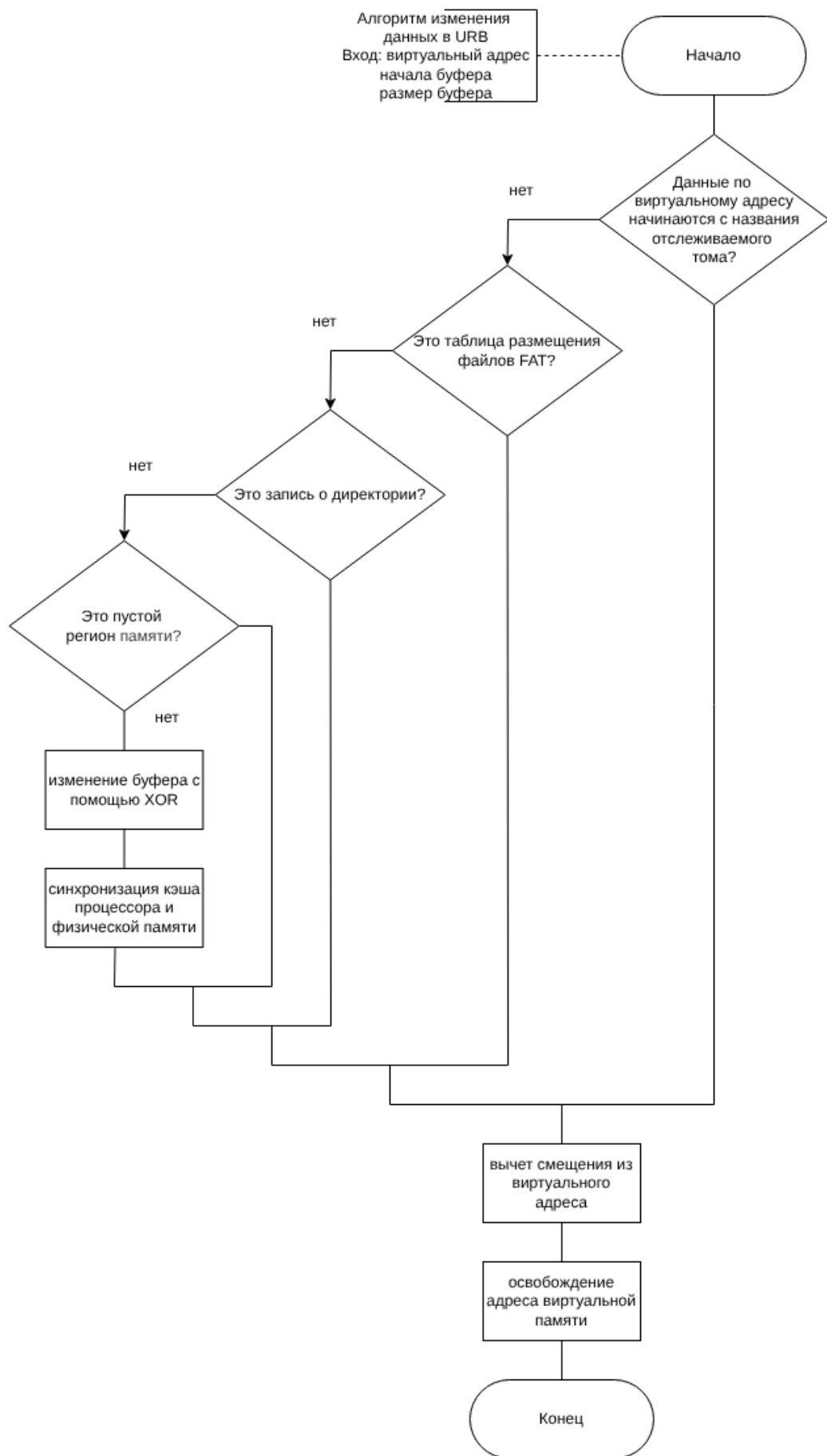


Рисунок 2.5 – Алгоритм изменения данных в URB

3 Технологический раздел

3.1 Выбор языка и среды программирования

Для написания модуля был выбран язык программирования C. Для сборки модуля была использована утилита Make [7]. В качестве среды разработки была использована среда VSCode [8].

3.2 Конфигурация USB-накопителя

На листинге 3.1 представлены параметры отслеживаемого флеш накопителя, такие как идентификаторы производителя и устройства и имя тома, заданные в программе через макросы.

Листинг 3.1 – Параметры отслеживаемого USB-накопителя

```
#define VENDOR_ID    0x0dd8
#define PRODUCT_ID   0x3801
#define VOLUME_NAME  "MYFLASH"
```

3.3 Функция проверки буфера на соответствие таблице FAT

На листинге 3.2 представлен код функции, проверяющей, что буфер является таблицей размещения файлов.

Листинг 3.2 – Функция проверки буфера на соответствие таблице FAT

```
static int is_fat_table(const unsigned char *buffer)
{
    if (buffer[0] == 0xF8 &&
        buffer[1] == 0xFF &&
        buffer[2] == 0xFF &&
        buffer[3] == 0x0F &&
        buffer[4] == 0xFF &&
        buffer[5] == 0xFF &&
        buffer[6] == 0xFF &&
        buffer[7] == 0x0F)
        return 1;
    return 0;
}
```

3.4 Функции, вызываемые при загрузке и выгрузке модуля

На листинге 3.3 представлен код функций, выполняющихся при загрузке и выгрузке модуля. При входе регистрируется `kprobe`, при выгрузке удаляется.

Листинг 3.3 – Функции, вызываемые при загрузке и выгрузке модуля

```
static struct kprobe kp_submit;

static int __init monitor_init(void)
{
    kp_submit.symbol_name = "usb_hcd_link_urb_to_ep";
    kp_submit.pre_handler = handler_usb_hcd_link_urb_to_ep_pre;
    int ret = register_kprobe(&kp_submit);
    if (ret < 0) {
        printk(KERN_ERR "Failed kprobe: %d\n", ret);
        return ret;
    }
    printk(KERN_INFO "USB FAKER: Loaded\n");
    return 0;
}

static void __exit monitor_exit(void)
{
    unregister_kprobe(&kp_submit);
    printk(KERN_INFO "USB FAKER: Unloaded\n");
}

module_init(monitor_init);
module_exit(monitor_exit);
MODULE_LICENSE("GPL");
```

3.5 Функция обработки структуры URB

На листинге 3.4 представлен код функции, которая проверяет передаваемые в URB данные и меняет их с помощью XOR в случае, если они являются внутренним содержимым передаваемых на USB-накопитель файлов.

Листинг 3.4 – Функция обработки структуры URB

```
static void process(struct urb *urb){
    struct scatterlist *sg;
    int i;
    for_each_sg(urb->sg, sg, urb->num_sgs, i) {
        size_t len = sg->length;
        struct page *page = sg_page(sg);
        if (page) {
            void *virt_addr = kmap_atomic(page);
            if (virt_addr) {
                virt_addr += sg->offset;
                if (memcmp(virt_addr, VOLUME_NAME,
                           strlen(VOLUME_NAME)) == 0 ||
                    is_fat_table(virt_addr) ||
                    ((char*)virt_addr)[11] == 0x10 ||
                    ((char*)virt_addr)[0] == 0xE5)
                    goto unmap;
                unsigned char *data = virt_addr;
                for (size_t j = 0; j < len; j++)
                    data[j] ^= 0xAA;
                if (urb->dev->bus && urb->dev->bus->sysdev) {
                    dma_sync_single_for_device(
                        urb->dev->bus->sysdev,
                        sg_dma_address(sg),
                        len, DMA_TO_DEVICE);
                }
unmap:
                virt_addr -= sg->offset;
                kunmap_atomic(virt_addr);
            }
            else printk(KERN_INFO "  SG[%d]: kmap failed\n", i);
        }
        else printk(KERN_INFO "  SG[%d]: NULL page\n", i);
    }
}
```

3.6 Функция-обработчик точки останова

На листинге 3.5 представлен код функции, которая выполняется при достижении точки останова. Указатель на структуру URB берётся из регистра `si`. Проверяются параметры устройства, направление передачи, является ли передаваемый URB блоком команды. После всех проверок вызывается функция обработки URB.

Листинг 3.5 – Функция-обработчик точки останова

```
static int handler_usb_hcd_link_urb_to_ep_pre(struct kprobe *p,
struct pt_regs *regs)
{
    struct urb *urb = (struct urb *)regs->si;
    if (!urb || !urb->dev)
        return 0;
    struct usb_device *udev = urb->dev;

    if (udev->descriptor.idVendor == VENDOR_ID &&
        udev->descriptor.idProduct == PRODUCT_ID &&
        usb_urb_dir_out(urb) == 1) {
        struct bulk_cb_wrap *cbw = (struct bulk_cb_wrap
            *)urb->transfer_buffer;
        if (cbw &&
            cbw->Signature &&
            le32_to_cpu(cbw->Signature) == US_BULK_CB_SIGN) {
            return 0;
        }
        process(urb);
    }
    return 0;
}
```

3.7 Makefile

Для сборки проекта была использована утилита Make и написан Makefile, код которого представлен на листинге 3.6.

Листинг 3.6 – Код Makefile

```
obj-m += mon.o
mon-objs := monitor.o
KDIR ?= /lib/modules/$(shell uname -r)/build

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

.PHONY: all
```

Выполнение команды make без параметров приведет к сборке файла загружаемого модуля ядра mon.ko. Для загрузки модуля используется команда `sudo insmod mon.ko`. Для выгрузки модуля используется команда `sudo rmmod mon`.

4 Исследовательский раздел

4.1 Исследование работы программы

Для проверки модуля ядра на USB-накопитель перемещали файлы и папки с разным содержимым.

4.1.1 Тест №1 — текстовый файл

На флеш-накопитель был скопирован файл с содержимым, представленным на листинге 4.1:

Листинг 4.1 – Текст передаваемого файла

```
version: '3.8'
services:
  express:
    build: ./express
    container_name: express-app
    environment:
      - DATABASE_NAME=anekdot_test
      - HOST=postgres
      - PORT=5432
      - USER=postgres
      - PASSWORD=password
    depends_on:
      postgres:
        condition: service_healthy
    deploy:
      resources:
        limits:
          cpus: '1'
          memory: 256M
    ports:
      - "3000:3000"
    networks:
      - benchmark
```

На накопителе содержимое файла выглядело следующим образом:

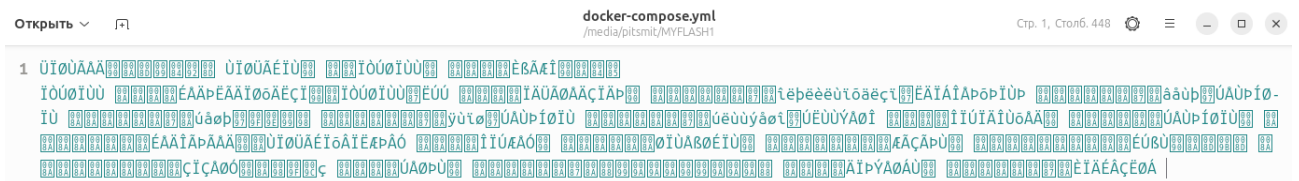


Рисунок 4.1 – Содержимое записанного на USB-накопитель файла

4.1.2 Тест №2 — директория с поддиректориями и файлами

На накопитель была скопирована директория с поддиректориями. В поддиректориях хранились текстовые файлы.

При записи на накопитель структура папок осталась неизменной. На листинге 4.2 представлена структура папок, полученная с помощью команды `tree`.

Листинг 4.2 – Структура переданной папки

```
* MYFLASH1 tree
.
  jest
    config
      jest.config.factory.js
      jest.config.js
    setup
      jest.polyfills.js
      jest.setup.ts

4 directories, 4 files
```

Во всех текстовых файлах данные были изменены, что видно на рисунке 4.2.



Рисунок 4.2 – Содержимое записанного на USB-накопитель файла

4.1.3 Тест №3 — файл PDF

На накопитель был скопирован одностраничный PDF файл. При попытке открыть его с USB-накопителя с помощью стандартного приложения Linux Ubuntu Просмотрщик Документов, было выведено следующее сообщение:

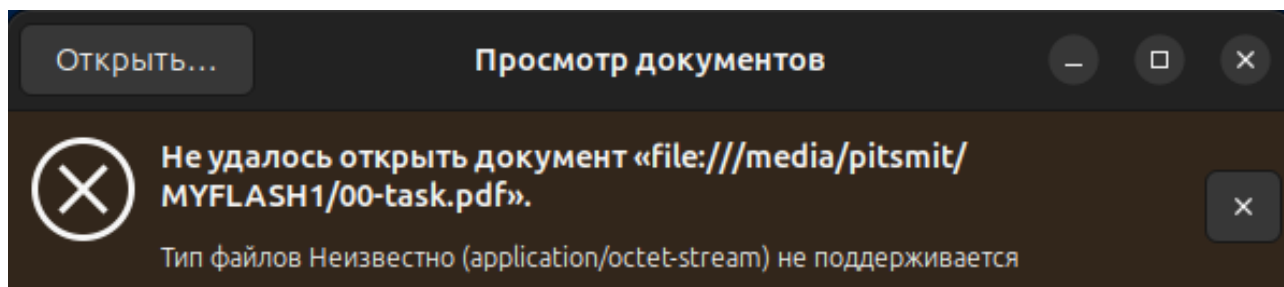


Рисунок 4.3 – Сообщение об ошибке при попытке открыть PDF-файл

4.2 Выводы

Проведенное исследование разработанного модуля ядра показало, что модуль удовлетворяет всем заявленным требованиям ТЗ: подменяет данные в передаваемых файлах на USB-накопитель.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. DLP: предотвращаем утечки; [Электронный ресурс]. — Режим доступа: <https://habr.com/ru/companies/otus/articles/798787/> (дата обращения: 03.12.2025).
2. Linux-USB Host Side API; [Электронный ресурс]. — Режим доступа: <https://microsin.net/programming/arm-working-with-usb/linux-usb-host-side-api.html> (дата обращения: 03.12.2025).
3. Шина USB и FireWire; [Электронный ресурс]. — Режим доступа: <https://схем.net/comp/comp56.php> (дата обращения: 03.12.2025).
4. Код ядра Linux; [Электронный ресурс]. — Режим доступа: <https://elixir.bootlin.com/linux/v6.12.6/source> (дата обращения: 03.12.2025).
5. Kernel Probes (Kprobes); [Электронный ресурс]. — Режим доступа: <https://docs.kernel.org/trace/kprobes.html> (дата обращения: 03.12.2025).
6. Мешков В. Архитектура файловой системы FAT // Системный администратор. — 2004. — №. 2. — С. 42-54.
7. Документация Make; [Электронный ресурс]. — Режим доступа: <https://www.gnu.org/software/make/manual/make.html> (дата обращения: 03.12.2025).
8. Документация VSCode; [Электронный ресурс]. — Режим доступа: <https://code.visualstudio.com/> (дата обращения: 03.12.2025).

ПРИЛОЖЕНИЕ А