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:

«Драйвер устройства сканирования отпечатков пальцев Shenzhen Goodix Technology для операционной системы Linux»

Студент ИУ7-71Б (Группа) _____ (Подпись, дата) Корниенко К. Ю. (И. О. Фамилия)

Руководитель курсовой работы _____ (Подпись, дата) Рязанова Н. Ю. (И. О. Фамилия)

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель данной работы — написать драйвер сканера отпечатка пальца.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с техническим заданием, требуется написать драйвер сканера отпечатка пальца Shenzhen Goodix Technology для операционной системы Linux.

Для решения поставленной задачи необходимо:

— проанализировать особенности работы сканера отпечатка пальца;

— рассмотреть способы получения данных от устройства с использованием подсистемы USB;

— разработать драйвер в виде загружаемого модуля ядра;

— провести исследование разработанного драйвера.

1.2 Конфигурация устройства

Ниже приведена конфигурация устройства, полученная из отладочной файловой системы по пути `/sys/kernel/debug/devices`.

```
T:  Bus=03 Lev=01 Prnt=01 Port=03 Cnt=01 Dev#=  2 Spd=12   MxCh= 0
D:  Ver= 2.00 Cls=02(comm.) Sub=01 Prot=01 MxPS=64 #Cfgs=  1
P:  Vendor=27c6 ProdID=5125 Rev= 2.00
S:  Manufacturer=Shenzhen Goodix Technology Co.,Ltd.
S:  Product=Goodix Fingerprint Device
S:  SerialNumber=00000000001A
C:* #Ifs= 2 Cfg#= 1 Atr=60 MxPwr=100mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=02(comm.) Sub=01 Prot=01 Driver=(none)
E:  Ad=82(I) Atr=03(Int.) MxPS=   8 Ivl=16ms
I:* If#= 1 Alt= 0 #EPs= 2 Cls=0a(data ) Sub=00 Prot=00 Driver=(none)
E:  Ad=01(O) Atr=02(Bulk) MxPS=  64 Ivl=0ms
E:  Ad=81(I) Atr=02(Bulk) MxPS=  64 Ivl=0ms
```

Данное устройство имеет одну конфигурацию с двумя активными интерфейсами, первый из которых предназначен для передачи управляющих комманд, а второй – для передачи данных.

1.3 Подсистема USB

USB драйвер имеет две точки входа – `probe` и `disconnect`, которые вызываются подсистемой USB при подключении и отключении устройства, соответствующего данному драйверу.

Обмен информацией между подключенным устройством и драйвером осуществляется пакетами по запросу. Блок запроса USB (USB Request Block – URB) представляется структурой ядра `urb`:

```
struct urb {
    struct list_head urb_list;
    struct usb_device *dev; /* (in) pointer to associated device */
    unsigned int pipe; /* (in) pipe information */
    int status; /* (return) non-ISO status */
    void *transfer_buffer; /* (in) associated data buffer */
    dma_addr_t transfer_dma; /* (in) dma addr for transfer_buffer */
    u32 transfer_buffer_length; /* (in) data buffer length */
    u32 actual_length; /* (return) actual transfer length */
    int interval; /* (modify) transfer interval */
    void *context; /* (in) context for completion */
    usb_complete_t complete; /* (in) completion routine */
    /* ... */
};
```

1.3.1 Способы передачи URB пакетов

Передача URB пакетов по шине USB является дуплексной и может происходить в одной из четырех форм, в зависимости от подключенного устройства:

— `Control` – используется для передачи управляющих команд для настройки устройства или получения его статуса.

— `Bulk` – используется для обмена большими пакетами данных. Зачастую именно эта форма передачи используется устройствами на подобие сканеров и SCSI адаптеров.

— `Interrupt` – используется для запроса передачи небольших пакетов в режиме опроса устройства. Если был запрошен пакет в режиме

прерывания, то драйвер хост-контроллера автоматически будет повторять этот запрос с определенным интервалом (1–255 мс).

— `Isochronous` – используется для передачи данных в реальном времени с гарантированной пропускной способностью шины, но ненадежно. В общем случае изохронный тип используется для передачи аудио и видео информации.

Рассматриваемое устройство – сканер отпечатка пальца – поддерживает форматы передачи `Control` (для передачи управляющих команд) и `Bulk` (для передачи изображения считанного отпечатка пальца).

1.4 Особенности взаимодействия с устройством с использованием протокола TLS

Сканер отпечатков пальцев Shenzhen Goodix Technology для Linux требует использования протокола TLS 1.2 для защиты передачи данных.

Для аутентификации в данном протоколе используются асимметричные алгоритмы шифрования (открытый ключ – закрытый ключ), а для сохранения конфиденциальности – симметричные (с одним, секретным, ключом), также используются и сеансовые ключи, которые необходимы для каждого отдельного уникального защищенного сеанса.

В соответствии с протоколом TLS процесс инициации сеанса, также называемый «рукопожатием», состоит из следующих шагов:

1. Клиент (в данном случае – драйвер) отправляет запрос на безопасное соединение с сервером (устройством).

2. Сервер предоставляет цифровой сертификат, подтверждающий подлинность сервера.

3. Используя открытый ключ сервера, клиент и сервер устанавливают сеансовый ключ, с помощью которого шифруются данные на протяжении всего сеанса.

Установка защищенного соединение с устройством производится через отправку URB Control блоков.

1.5 Взаимодействие драйвера с пользовательскими программами

Одним из возможных способов передачи информации из пространства ядра в пространство пользователя является использование виртуальной файловой системы proc [1].

Операции, которые могут быть осуществлены с файлом определяются структурой `proc_ops`. В данной работе необходимо и достаточно реализовать следующие операции: `proc_open`, `proc_read`, `proc_lseek`, `proc_poll`, `proc_release`.

Так как изображение отпечатка пальца имеет существенный размер, пользовательское приложение может не обработать все данные за одну операцию чтения. В ядре Linux есть интерфейс файлов последовательностей (sequence file). Использование данного интерфейса избавляет разработчика загружаемого модуля ядра от необходимости учитывать описанную проблему.

В случае, если пользователь не прикладывает палец к сканеру, процесс должен быть заблокирован при попытке чтения. Необходимость блокировки процесса и ожидания появления нового события приводит к использованию очередей ожидания, которые представляются в ядре структурой `wait_queue_head`:

```
struct wait_queue_entry {
    unsigned int flags;
    void *private;
    wait_queue_func_t func;
    struct list_head entry;
};

struct wait_queue_head {
    spinlock_t lock;
    struct list_head head;
};
```

Блокировка процесса и ожидание возникновения события осуществляется вызовом макроса

`wait_event_interruptible(wq_head, condition)`.

Все ждущие в данной очереди процессы пробуждаются посредством вызова макроса `wake_up_all(wq_head)`, после чего будет анализировано вы-

ражение `condition` переданное при блокировке. Если оно ложно, то процесс вновь блокируется.

Для поддержки механизма опроса реализуется функция `proc_poll`.

Выводы

В данном разделе была представлена конфигурация устройства сканера, изложены основные моменты работы с подсистемой USB при проектировании драйверов USB-устройств в ОС Linux. Также рассмотрены особенности взаимодействия драйвера с устройством с использованием протокола TLS и обеспечением защищенного соединения. Обсуждены способы передачи данных из пространства ядра в пространство пользователя.

2 Конструкторский раздел

В данном разделе приведены ключевые алгоритмы использованные при написании драйвера сканера отпечатка пальца.

2.1 Структура ПО

Разрабатываемое ПО состоит из драйвера сканера отпечатка пальца и пользовательского приложения для тестирования работы драйвера.

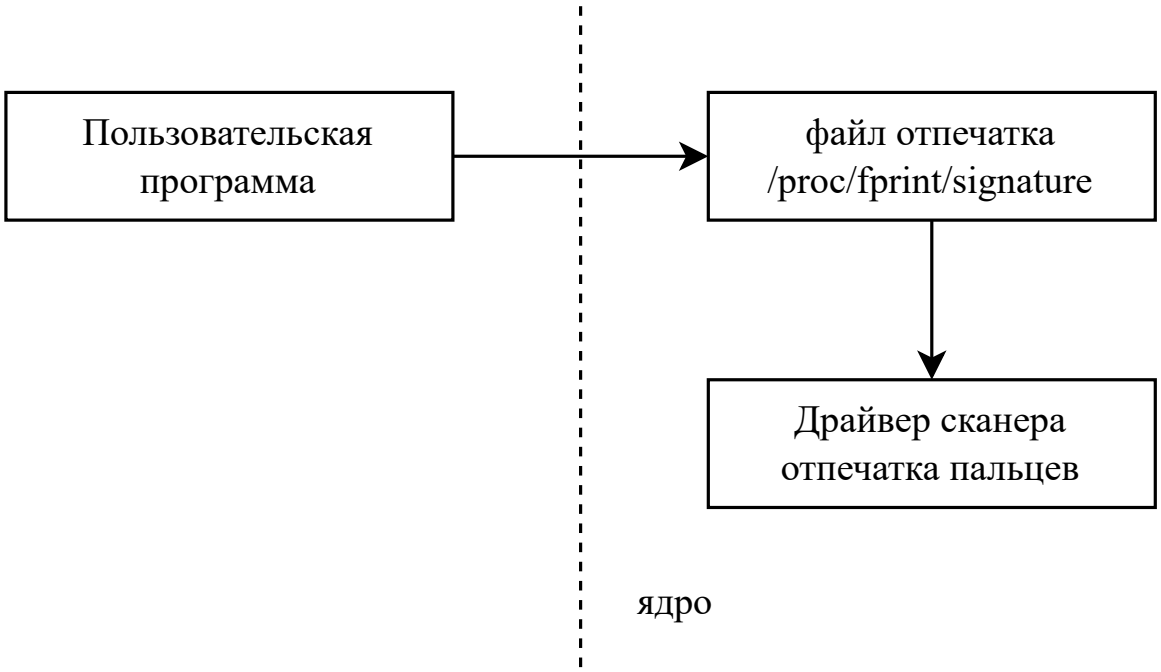

Рисунок 2.1 – Структура ПО

2.2 Алгоритм обработки URB

На рисунке 2.2 представлен алгоритм инициализации TLS соединения с устройством. PKS – pre-shared key cichersuites – предварительный общий ключ безопасности используемый в качестве сертификата.

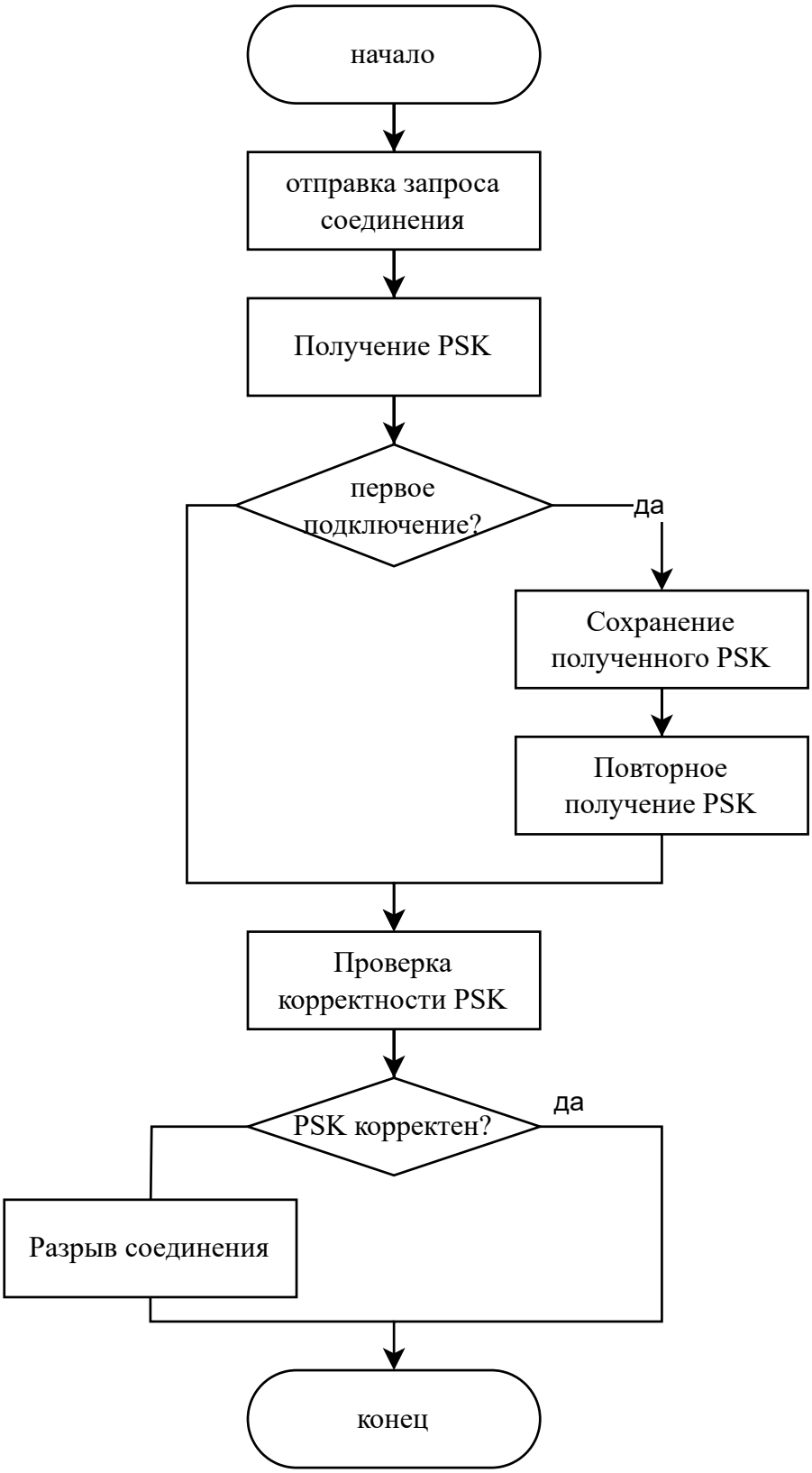

Рисунок 2.2 – Алгоритм инициализации соединения

3 Технологический раздел

3.1 Выбор языка и среды разработки

Для написания драйвера был выбран язык программирования C.

Для сборки драйвера будет использована утилита Make. В качестве среды разработки был выбран редактор VSCode.

Драйвер разрабатывается для ОС Linux версии 6.7.

3.2 Точки входа драйвера

Листинг 3.1 – функция probe драйвера

```
int fprint_probe(struct usb_interface *intf, const struct
   usb_device_id *id)
{
    printk(KERN_INFO LOG_PREFIX "fingerprint usb interface
       probed\n");

    struct usb_host_interface *iface = intf->cur_altsetting;
    printk(KERN_INFO LOG_PREFIX "fingerprint usb interface
       number: %d\n", iface->desc.bInterfaceNumber);
    if (iface->string != NULL) {
        printk(KERN_INFO LOG_PREFIX "fingerprint usb interface
           string: %s\n", iface->string);
    } else {
        printk(KERN_WARNING LOG_PREFIX "fingerprint usb
           interface string is NULL\n");
    }

    if (iface->desc.bNumEndpoints != 3) {
        printk(KERN_INFO LOG_PREFIX "skipping this interface\n");
        return 0;
    }

    if (iface->desc.bInterfaceNumber != 0) {
        printk(KERN_INFO LOG_PREFIX "skipping this interface\n");
        return 0;
    }

    struct usb_endpoint_descriptor *endpoint =
       &iface->endpoint[0].desc;
```

```
    if (endpoint->bEndpointAddress != 0x82) {
        printk(KERN_WARNING LOG_PREFIX "endpoint[1] must have
           adress 0x81\n");
        return 0;
    }

    int res = initialize_fprint_endpoint(intf, endpoint);
    if (res != 0) {
        printk(KERN_ERR LOG_PREFIX "failed to initialize
           endpoint\n");
    }
    return res;
}
```

Листинг 3.2 – функция `disconnect` драйвера

```
void fprint_disconnect(struct usb_interface *intf)
{
    printk(KERN_INFO LOG_PREFIX "fingerprint usb interface
       disconnected\n");

    struct usb_host_interface *iface = intf->cur_altsetting;
    printk(KERN_INFO LOG_PREFIX "fingerprint usb interface
       number: %d\n", iface->desc.bInterfaceNumber);
    printk(KERN_INFO LOG_PREFIX "fingerprint usb interface
       string: %d\n", iface->desc.iInterface);

    // free allocated structures here
    if (iface->desc.bInterfaceNumber != 0) {
        printk(KERN_INFO LOG_PREFIX "skipping this interface\n");
        return;
    }

    struct usb_endpoint_descriptor *endpoint =
       &iface->endpoint[0].desc;
    if (endpoint->bEndpointAddress != 0x82) {
        printk(KERN_WARNING LOG_PREFIX "endpoint[1] must have
           adress 0x81\n");
        return;
    }

    struct usb_device* usbdev = interface_to_usbdev(intf);
    struct fprint_drv_data* drvdata =
```

```
        dev_get_drvdata(&usbdev->dev);
    if (drvdata != NULL) {
        // usb_free_urb(drvdata->urb);
        kfree(drvdata->transfer_buffer);
        kfree(drvdata);
        printk(KERN_INFO LOG_PREFIX "driver data free'd\n");
    }
}
```

3.3 Работа с файловой системой proc

Листинг 3.3 – функции работы с VFS proc

```
static atomic_t reader_available = ATOMIC_INIT(1);
static DECLARE_WAIT_QUEUE_HEAD(reader_queue);
static DECLARE_WAIT_QUEUE_HEAD(poll_wait_queue);

static const struct seq_operations fprint_seq_ops = {
    .start = start,
    .next = next,
    .stop = stop,
    .show = show
};

static const struct proc_ops fprint_proc_ops = {
    .proc_open    = open,
    .proc_read    = seq_read,
    .proc_lseek   = seq_lseek,
    .proc_release = seq_release,
    .proc_poll    = poll,
};

void* start(struct seq_file *m, loff_t *pos)
{
    seqfile_iterator = 0;
    return &seqfile_iterator;
}

void stop(struct seq_file *m, void *v)
{
    atomic_set(&reader_available, 1);
}
```

```
void* next(struct seq_file *m, void *v, loff_t *pos)
{
    wait_event_interruptible(reader_queue,
       signature_data_available);
    signature_data_available = false;

    *pos += signature_data_size;
    return NULL;
}

int show(struct seq_file *m, void *v)
{
    if (signature_data == NULL) {
        printk(KERN_WARNING LOG_PREFIX "signature data is
           NULL\n");
        return 0;
    }

    seq_write(m, signature_data, signature_data_size);
    return 0;
}

int open(struct inode* inode, struct file* file)
{
    signature_data_available = false;
    return seq_open(file, &fprint_seq_ops);
}

static unsigned int poll(struct file *file, poll_table *wait)
{
    poll_wait(file, &poll_wait_queue, wait);
    if (signature_data_available)
        return POLLIN | POLLRDNORM;
    return 0;
}
```

4 Исследовательский раздел

4.1 Постановка исследования

Исследование скорости обработки запроса отпечатка пальца было проведено для значений размера буфера чтения от 16 до 512 байт. Замер времени осуществлялся с использованием функции `clock_gettime` стандартной библиотеки языка C. При этом в качестве идентификатора часов использовался `CLOCK_PROCESS_CPUTIME_ID` для измерения времени, затраченного непосредственно процессом.

4.2 Результаты

На рисунке 4.1 приведен график зависимости времени обработки запроса отпечатка пальца пользовательского приложения от размера буфера чтения.

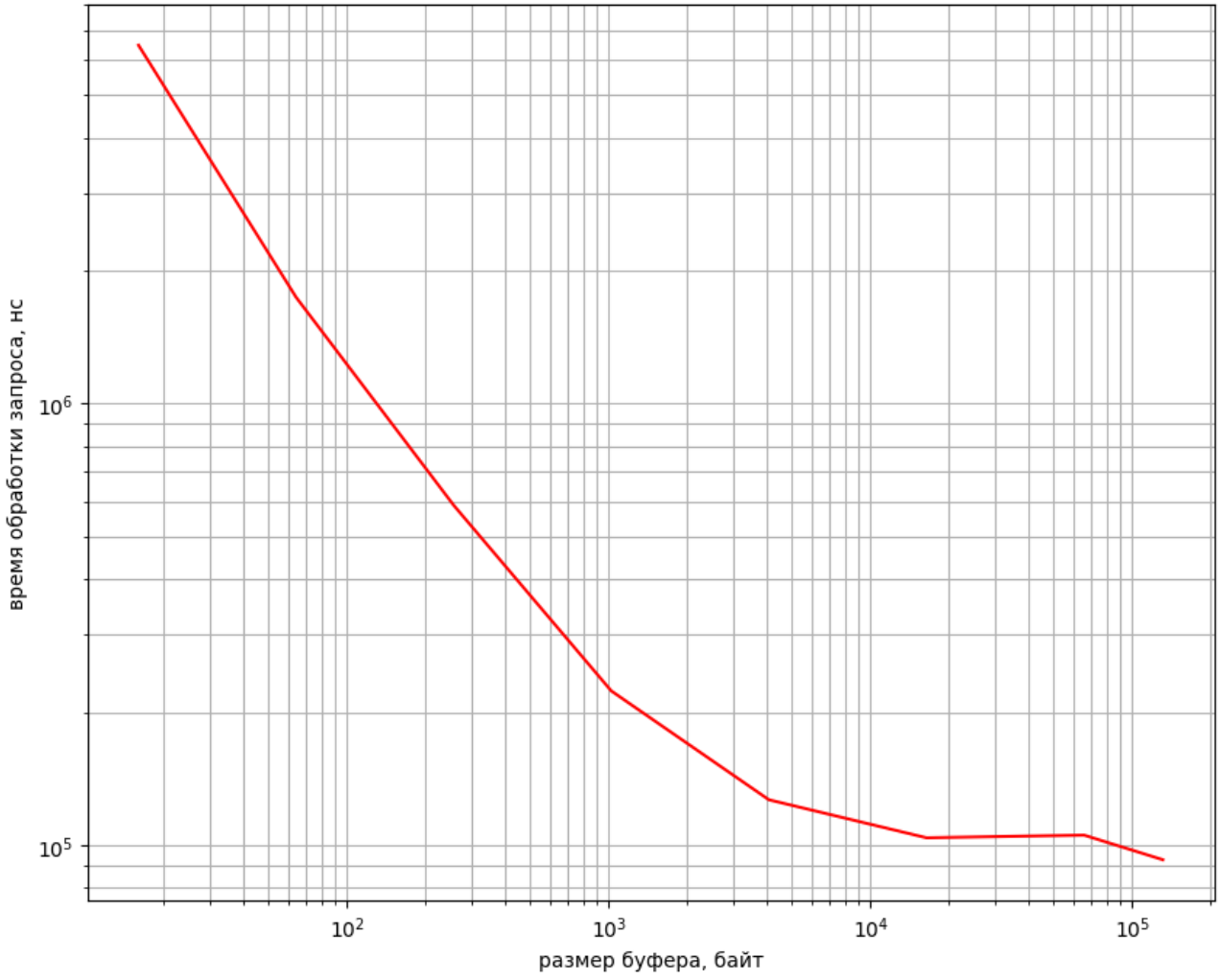

Рисунок 4.1 – Зависимость времени обработки запроса от размера буфера

ЗАКЛЮЧЕНИЕ

В ходе работы была изучена подсистема USB, рассмотрены и решены проблемы связанные с работой сканера отпечатка пальца.

Был разработан драйвер для сканера отпечатка пальцев.

Было проведено исследование зависимости времени обработки запроса отпечатка пальца от пользовательского приложения для различных размеров буферов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. The /proc Filesystem – The Linux Kernel documentation. [Электронный ресурс]. URL: https://docs.kernel.org/filesystems/proc.html (Дата обращения: 21.11.2023).

ПРИЛОЖЕНИЕ А