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 и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Загружаемый модуль ядра для мониторинга сетевого трафика процесса»

Студент ИУ7-73Б (Группа) _____ (Подпись, дата) В. Марченко (И. О. Фамилия)

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

2023 г.

РЕФЕРАТ

Расчетно-пояснительная записка 43 с., 16 рис., 1 табл., 6 источн., 2 прил.
ЗАГРУЖАЕМЫЙ МОДУЛЬ ЯДРА, ОПЕРАЦИОННЫЕ СИСТЕМЫ, СЕТЕВОЙ ТРАФИК, СИСТЕМНЫЕ ВЫЗОВЫ, СОКЕТЫ, ФУНКЦИИ ЯДРА, FTRACE, LINUX

Объектом разработки является загружаемый модуль ядра для мониторинга сетевого трафика процесса.

Цель работы: реализация загружаемого модуля ядра, с помощью которого можно получить суммарный размер полученных и отправленных каким-либо процессом пакетов по сети.

В результате выполнения работы был реализован и протестирован загружаемый модуль ядра для мониторинга сетевого трафика процесса.

Область применения результатов — мониторинг сетевого трафика процессов и расширение функционала реализованного загружаемого модуля ядра.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей расчетно-пояснительной записке к курсовой работе применяют следующие сокращения и обозначения:

ОС	Операционная система
ПО	Программное обеспечение
ЗМЯ	Загружаемый модуль ядра
PID	Идентификатор процесса (от англ. process identifier)

ВВЕДЕНИЕ

Сетевой трафик — объем информации, передаваемой через компьютерную сеть за определенный период времени. Количество трафика измеряется как в пакетах, так и в битах, байтах и их производных. Трафик подразделяется на:

1) исходящий (информация, поступающая во внешнюю сеть);
2) входящий (информация, поступающая из внешней сети);
3) внутренний (в пределах определенной сети, чаще всего локальной);
4) внешний (за пределами определенной сети).

В настоящее время сетевой трафик является неотъемлемой частью повседневной жизни, и его отслеживание играет важную роль в различных сферах. Отслеживание сетевого трафика позволяет анализировать передаваемые данные, идентифицировать нарушения безопасности, оптимизировать сеть и повышать эффективность работы.

Существует множество программ для подсчета сетевого трафика. Например, TMeter, BWMeter, BitMeter II и GabNetStats — это утилиты для операционных систем семейства Microsoft Windows. Для Linux доступны программы vnStat, Nload и slurm. На рисунке 1 показан пример работы утилиты vnStat. Вышеперечисленные программы позволяют получать информацию о скорости передачи данных в сети и об объеме передаваемых по сети пакетов. Существенный недостаток некоторых из этих инструментов — они отслеживают сетевой трафик не для конкретного процесса, а для всей системы.

Для отслеживания сетевого трафика необходимо перехватывать функции ядра. Существует несколько способов это сделать:

1) модификация таблицы системных вызовов (данный способ устарел в новых версиях ядра);
2) использование утилиты strace;
3) использование системного вызова ptrace();
4) использование фреймворка ftrace.

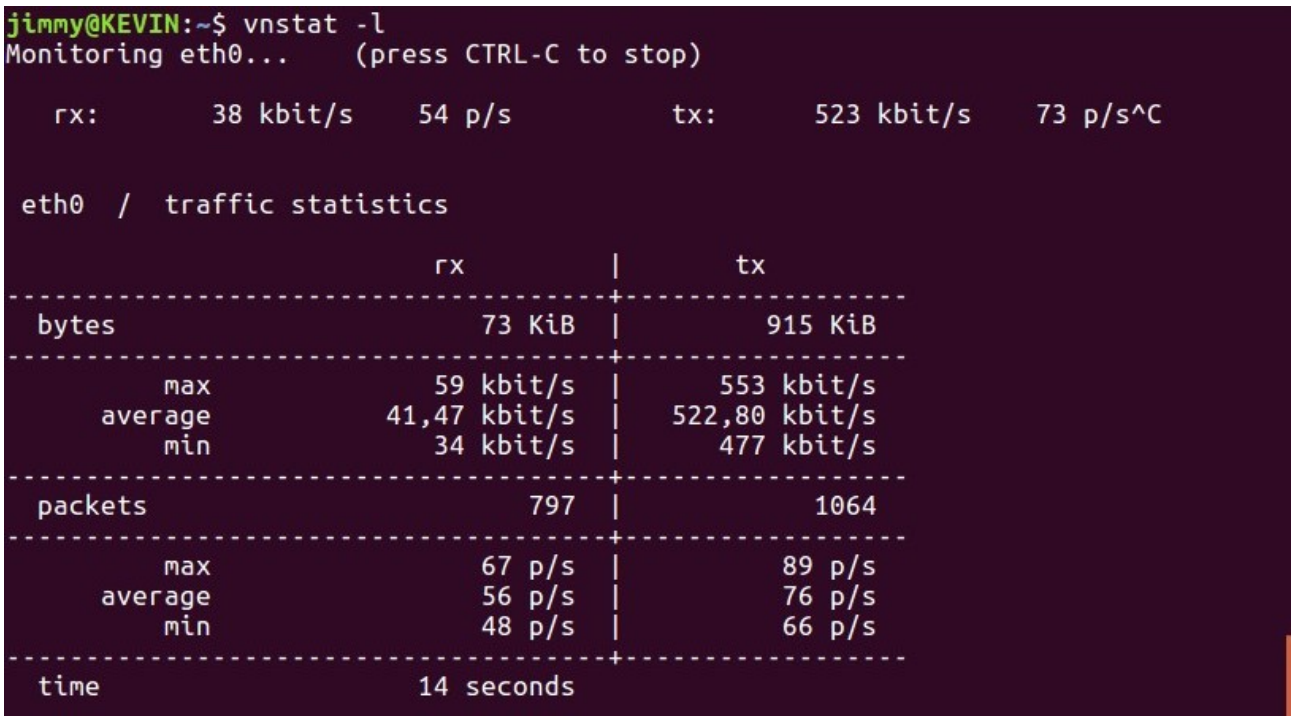

Рисунок 1 – Пример работы утилиты vnStat

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу целью является реализация загружаемого модуля ядра, с помощью которого можно получить суммарный размер полученных и отправленных каким-либо процессом пакетов по сети.

Задачи курсовой работы:

1) провести сравнительный анализ существующих методов перехвата функций ядра Linux и выбрать наиболее подходящий для достижения поставленной цели;

2) разработать программное обеспечение для мониторинга сетевого трафика процесса;

3) реализовать ПО для мониторинга сетевого трафика процесса;

4) протестировать реализованное программное обеспечение.

1.2 Ограничения и требования к ПО

Разрабатываемый загружаемый модуль ядра должен перехватывать функции, относящиеся к передачи пакетов по сети. Подсчет объема сетевого трафика должен вестись для конкретного процесса, поэтому перед загрузкой модуля ему на вход нужно передать имя программы.

Для хранения объема отправленных и полученных пакетов в байтах можно использовать переменную типа uint64_t. Этот тип данных обеспечивает наибольший диапазон беззнаковых чисел в языке C. Таким образом, ограничение к разрабатываемому модулю: наибольший объем полученных и отправленных по сети пакетов — 16384 пебибайт (18446744 терабайт).

1.3 Методы перехвата функций ядра Linux

1.3.1 strace

В Linux есть несколько средств для перехвата функций ядра. Первое средство — утилита strace.

strace — это диагностическая и отладочная утилита, работающая в пространстве пользователя. Она используется для мониторинга и изменения взаимодействия между процессами и ядром Linux, включая системные вызовы, доставку сигналов и изменения состояния процесса. Данная утилита подходит для решения проблем с программами, исходный код которых недоступен, поскольку их не нужно перекомпилировать для отладки. Работа strace основана на системном вызове ptrace() [1].

Достоинства strace — утилиту можно запустить для трассировки уже работающий программы; можно фильтровать системные вызовы, которые необходимо отслеживать, по типу; а также можно собирать статистику (количество вызовов, время выполнения и ошибки) [1].

В листинге 1.1 показан пример работы утилиты strace.

Листинг 1.1 – Пример работы утилиты strace

```
$ strace -P /etc/ld.so.cache ls /var/empty
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=22446, ...}) = 0
mmap(NULL, 22446, PROT_READ, MAP_PRIVATE, 3, 0) = 0x2b7ac2ba9000
close(3) = 0
+++ exited with 0 +++
```

1.3.2 ptrace

ptrace (от англ. process trace) — системный вызов в некоторых UNIX-подобных системах (в том числе в Linux и FreeBSD), который позволяет трассировать или отлаживать выбранный процесс. Можно сказать, что ptrace() дает полный контроль над процессом: можно изменять ход выполнения программы, смотреть и изменять значения в памяти или состояния регистров. Никаких дополнительных прав нет — возможные действия ограничены правами запущенного процесса. К тому же, при трассировке программы с setuid битом, этот самый бит не работает — привилегии не повышаются [2].

В листинге 1.2 показан прототип системного вызова ptrace().

Листинг 1.2 – Прототип системного вызова ptrace()

```
#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

Значения параметров системного вызова ptrace(): request — это действие, которое необходимо осуществить, например PTRACE_CONT, PTRACE_PEEKTEXT, pid — идентификатор трассируемого процесса, addr и data зависят от request [2].

Начать трассировку можно двумя способами: для уже запущенного процесса (PTRACE_ATTACH), либо запустить процесс самому с помощью PTRACE_TRACEME. Для управления трассировкой можно использовать следующие аргументы [2]:

1) PTRACE_SINGLESTEP — пошаговое выполнение программы, управление будет передаваться после выполнения каждой инструкции (такая трассировка достаточно медленная);

2) PTRACE_SYSCALL — продолжить выполнение программы до входа или выхода из системного вызова;

3) PTRACE_CONT — продолжить выполнение программы.

1.3.3 ftrace

ftrace — это внутренний трассировщик, призванный помочь разработчикам и проектировщикам систем обнаружить, что происходит внутри ядра. Его можно использовать для отладки или анализа задержек и проблем с производительностью, возникающих за пределами пространства пользователя [3].

Хотя ftrace обычно считается средством трассировки функций, на самом деле это совокупность нескольких различных утилит трассировки. Существует трассировка задержки для проверки того, что происходит между отключенными и включенными прерываниями, а также с момента пробуждения задачи до ее фактического планирования [3].

Одним из наиболее распространенных применений ftrace является трассировка событий. По всему ядру есть сотни статических событий, которые

можно включить через файловую систему tracefs, чтобы увидеть, что происходит в определенных частях ядра [3].

Инфраструктура ftrace изначально была создана для прикрепления функций обратного вызова (callbacks) к началу функций с целью записи и отслеживания потока ядра. Но обратные вызовы в начале функции могут иметь и другие варианты использования. Либо для оперативного исправления ядра, либо для мониторинга безопасности [3].

Чтобы зарегистрировать функции обратного вызова, нужно использовать структуру struct ftrace_ops. Эта структура используется, чтобы сообщить ftrace, какую функцию следует вызывать в качестве обратного вызова [3].

Необходимо заполнить только одно поле при регистрации struct ftrace_ops — func. Поля flags и private не являются обязательными [3]. В листинге 1.3 показан пример заполнения полей структуры struct ftrace_ops.

Листинг 1.3 – Пример заполнения полей структуры struct ftrace_ops

```
struct ftrace_ops fops = {
    .func    = my_callback_func,
    .flags   = MY_FTRACE_FLAGS,
    .private = any_private_data_structure,
};
```

Чтобы начать трассировку, нужно вызвать функцию register_ftrace_function(&fops), а чтобы завершить — unregister_ftrace_function(&fops) [3].

Структура struct ftrace_ops и необходимые для трассировки функции определены в заголовочном файле <linux/ftrace.h> [3].

1.4 Сравнительный анализ методов перехвата функций ядра

В таблице 1.1 приведено сравнение рассмотренных методов перехвата функций ядра Linux.

Таблица 1.1 – Сравнение методов перехвата функций ядра

Метод	**Возможность перехвата функций, вызванных конкретным процессом**	**Тип**	**Пространство**
strace	+	утилита	пользователя
ptrace()	+	системный вызов	пользователя
ftrace	+	фреймворк	ядра

Все рассмотренные методы обеспечивают возможность перехвата функций ядра для конкретного процесса, однако только ftrace работает в пространстве ядра, что и делает его подходящим средством для реализации загружаемого модуля.

1.5 Аргументы командной строки для загружаемого модуля ядра

Как и любая другая программа на языке C, загружаемый модуль ядра может принимать аргументы командной строки. Чтобы разрешить передачу аргументов в модуль, нужно объявить переменные, которые будут принимать значения аргументов командной строки, как глобальные, а затем использовать макрос module_param(), который определен в <linux/moduleparam.h>. Во время выполнения insmod проинициализирует переменные значениями аргументов командной строки. Объявления переменных и макросы размещаются в начале модуля. Макрос module_param() принимает 3 аргумента: имя переменной, ее тип и права доступа к соответствующему файлу в sysfs. Если необходимо использовать массивы целых чисел или строк, нужно использовать макросы module_param_array() и module_param_string() [4].

В листинге 1.4 показан пример объявления глобальных переменных, которые могут быть проинициализированы с помощью аргументов командной строки.

Листинг 1.4 – Пример объявления глобальных переменных для использования аргументов командной строки

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

MODULE_LICENSE("GPL");

static short int myshort = 1;
static int myint = 420;
static long int mylong = 9999;
static char *mystring = "test";
static int myintArray[2] = {-1, -1};
static int arr_argc = 0;

module_param(myshort, short, S_IRUSR | S_IWUSR
             | S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");
module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");
module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");
module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");
module_param_array(myintArray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintArray, "An array of integers");
```

1.6 Определение функций ядра для перехвата

Прототипы функций ядра, которые могут быть перехвачены с помощью ftrace, находятся в заголовочном файле <linux/syscalls.h> [5].

Чтобы определить функции ядра, которые выполняются при вызове конкретных системных вызовов, можно написать программу, вызывающую соответствующие системные вызовы, и воспользоваться утилитой strace.

1.6.1 Определение целевого процесса

Для того, чтобы мониторить сетевой трафик конкретного процесса, после передачи имени программы на вход загружаемому модулю ядра, нужно запустить эту программу. При этом необходимо перехватить функцию sys_execve(), которая вызывается при запуске программы. Затем с помощью функции copy_from_user() модуль сможет получить имя запущенной программы и сравнить с тем, которое было передано в качестве аргумента командной строки. Вторая функция ядра — sys_exit_group() — будет вызвана при завершении отслеживаемого процесса. Перехватив эту функцию, загружаемый модуль ядра запишет в системный лог статистику.

1.6.2 Определение размера пакета

Также нужно перехватывать функции ядра, которые вызываются в системных вызовах, связанных с передачей пакетов по сети. Это следующие функции ядра: sys_send(), sys_sendto(), sys_sendmsg(), sys_recv(), sys_recvfrom() и sys_recvmsg().

В листинге 1.5 показаны прототипы перечисленных функций ядра [5].

Листинг 1.5 – Прототипы функций ядра, которые нужно перехватывать для мониторинга сетевого трафика процесса

```
#include <linux/syscalls.h>

asmlinkage long sys_execve(const char __user *filename,
        const char __user *const __user *argv,
        const char __user *const __user *envp);
asmlinkage long sys_send(int, void __user *, size_t, unsigned);
asmlinkage long sys_sendto(int, void __user *, size_t,
    unsigned, struct sockaddr __user *, int);
asmlinkage long sys_sendmsg(int fd,
    struct user_msghdr __user *msg, unsigned flags);
asmlinkage long sys_recv(int, void __user *, size_t, unsigned);
asmlinkage long sys_recvfrom(int, void __user *,
    size_t, unsigned, struct sockaddr __user *, int __user *);
asmlinkage long sys_recvmsg(int fd,
    struct user_msghdr __user *msg, unsigned flags);
asmlinkage long sys_exit_group(int error_code);
```

Выводы из аналитического раздела

В результате проведения сравнительного анализа методов перехвата функций ядра Linux был выбран фреймворк ftrace. Для определения процесса, от имени которого выполняются функции ядра, необходимо воспользоваться механизмом передачи аргументов командной строки в загружаемый модуль ядра и перехватить функцию sys_execve(). Определены прототипы функций ядра для перехвата.

2 Конструкторский раздел

2.1 Последовательность выполняемых ПО действий

На рисунках 2.1–2.2 показана последовательность выполняемых программным обеспечением действий в виде IDEF0-диаграмм нулевого и первого уровней.

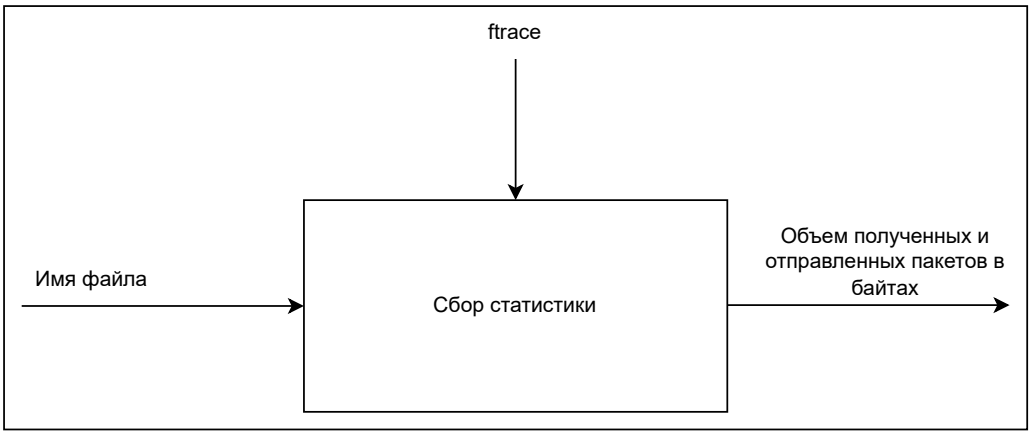

Рисунок 2.1 – IDEF0-диаграмма нулевого уровня

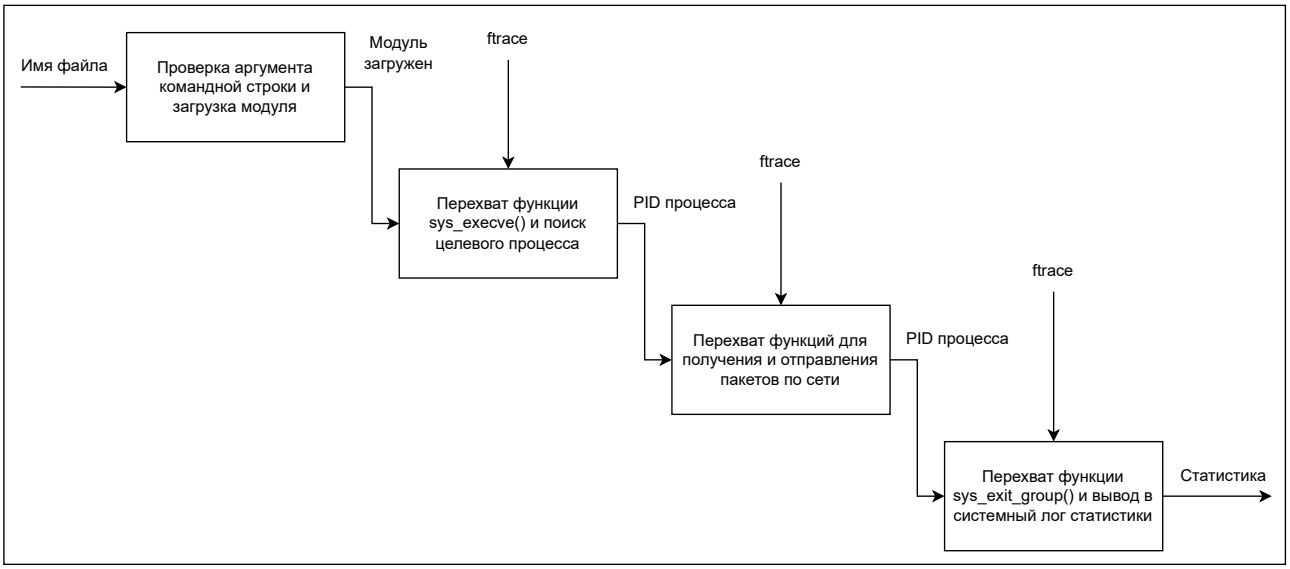

Рисунок 2.2 – IDEF0-диаграмма первого уровня

На вход загружаемому модулю ядра поступает имя файла в виде аргумента командной строки. Далее происходит проверка этого аргумента. Если ошибка не возникла, регистрируются перехватчики функций ядра и модуль загружается.

За счет перехвата функции sys_execve() загружаемый модуль ядра получает PID целевого процесса. Перехватывая функции, связанные с передачей пакетов по сети, модуль считает размер отправленных и полученных процессом пакетов. За счет перехвата sys_exit_group() загружаемый модуль ядра выводит в системный лог статистику при завершении отслеживаемого процесса. Затем модуль выгружается.

2.2 Алгоритм получения аргументов командной строки в режиме ядра

На рисунке 2.3 показана схема алгоритма получения аргументов командной строки в режиме ядра. Сначала проверяется, было ли указано имя файла в качестве аргумента командной строки. Затем, если проверка прошла успешно, регистрируются перехватчики функций ядра, и модуль загружается.

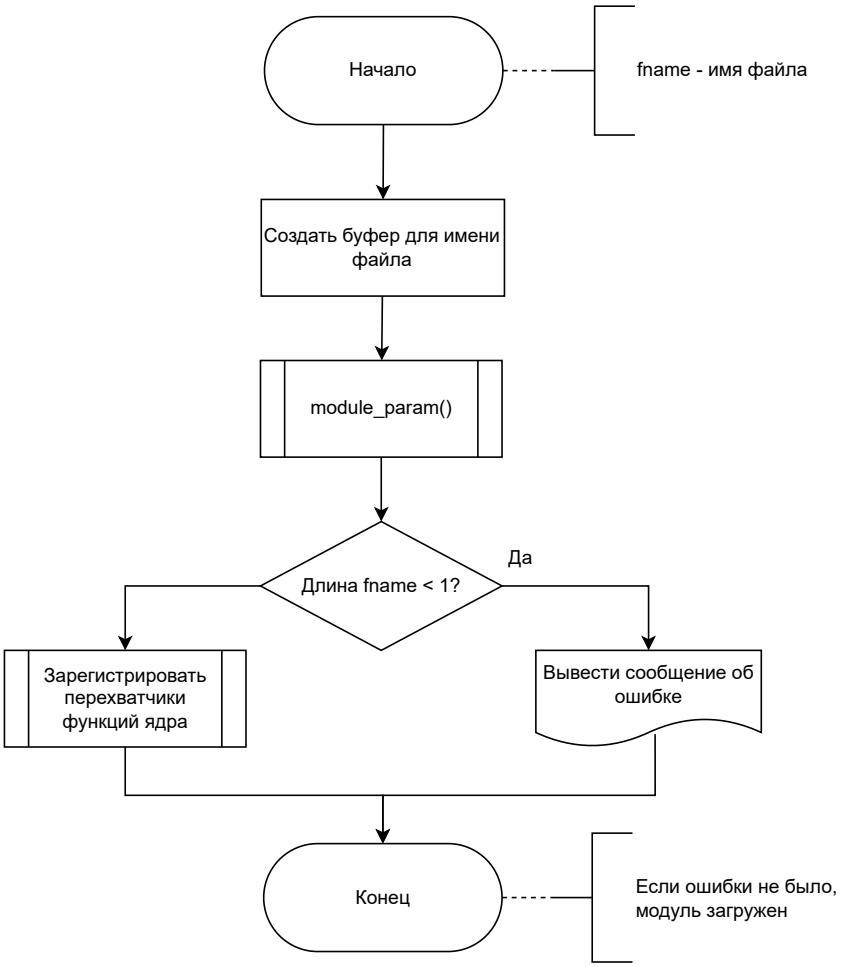

Рисунок 2.3 – Схема алгоритма получения аргументов командной строки в режиме ядра

2.3 Алгоритм определения целевого процесса

На рисунке 2.4 показана схема алгоритма определения целевого процесса. Перед выполнением sys_execve() проверяется, проинициализирован ли идентификатор целевого процесса. Если нет, имя файла копируется из пространства пользователя в пространство ядра. Затем происходит сравнение с именем, полученным в качестве аргумента командной строки. При успешном сравнении идентификатор целевого процесса инициализируется полученным с помощью макроса current значением процесса, от имени которого выполняется sys_execve().

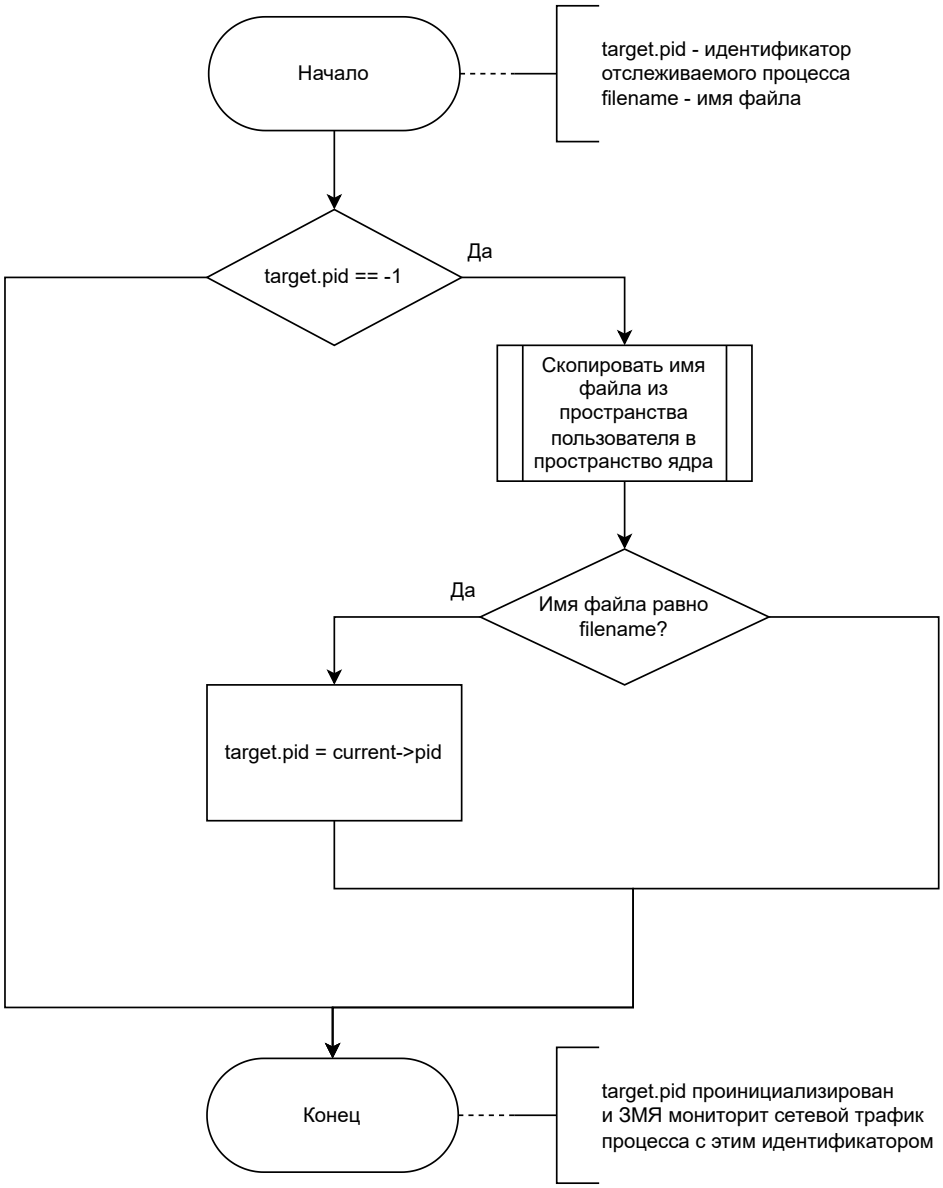

Рисунок 2.4 – Схема алгоритма определения целевого процесса

2.4 Алгоритм получения размера пакета

На рисунке 2.5 показана схема алгоритма получения размера отправленного пакета на примере функции ядра sys_send().

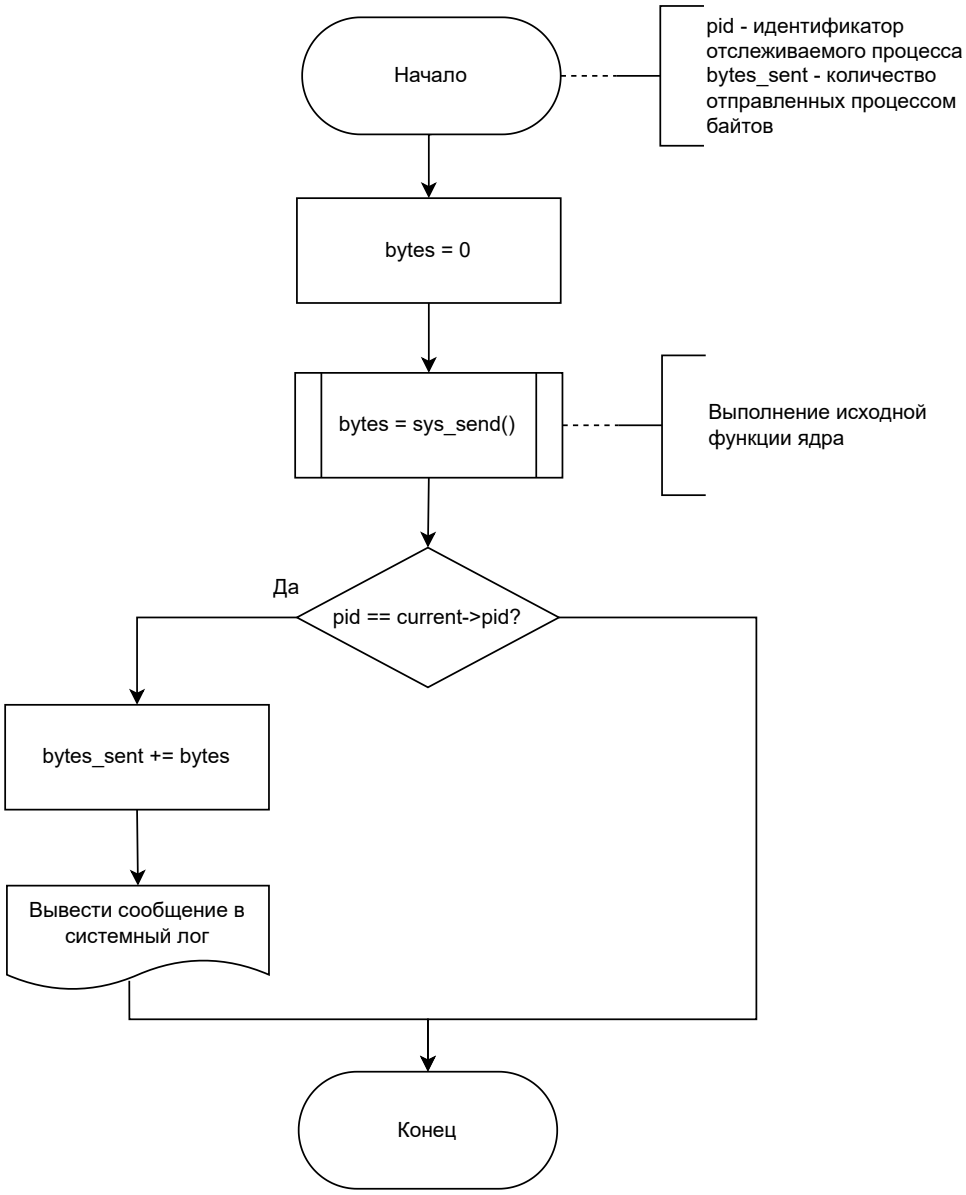

Рисунок 2.5 – Схема алгоритма получения размера пакета

2.5 Структура ПО

На рисунке 2.6 показана структура разрабатываемого программного обеспечения для мониторинга сетевого трафика процесса.

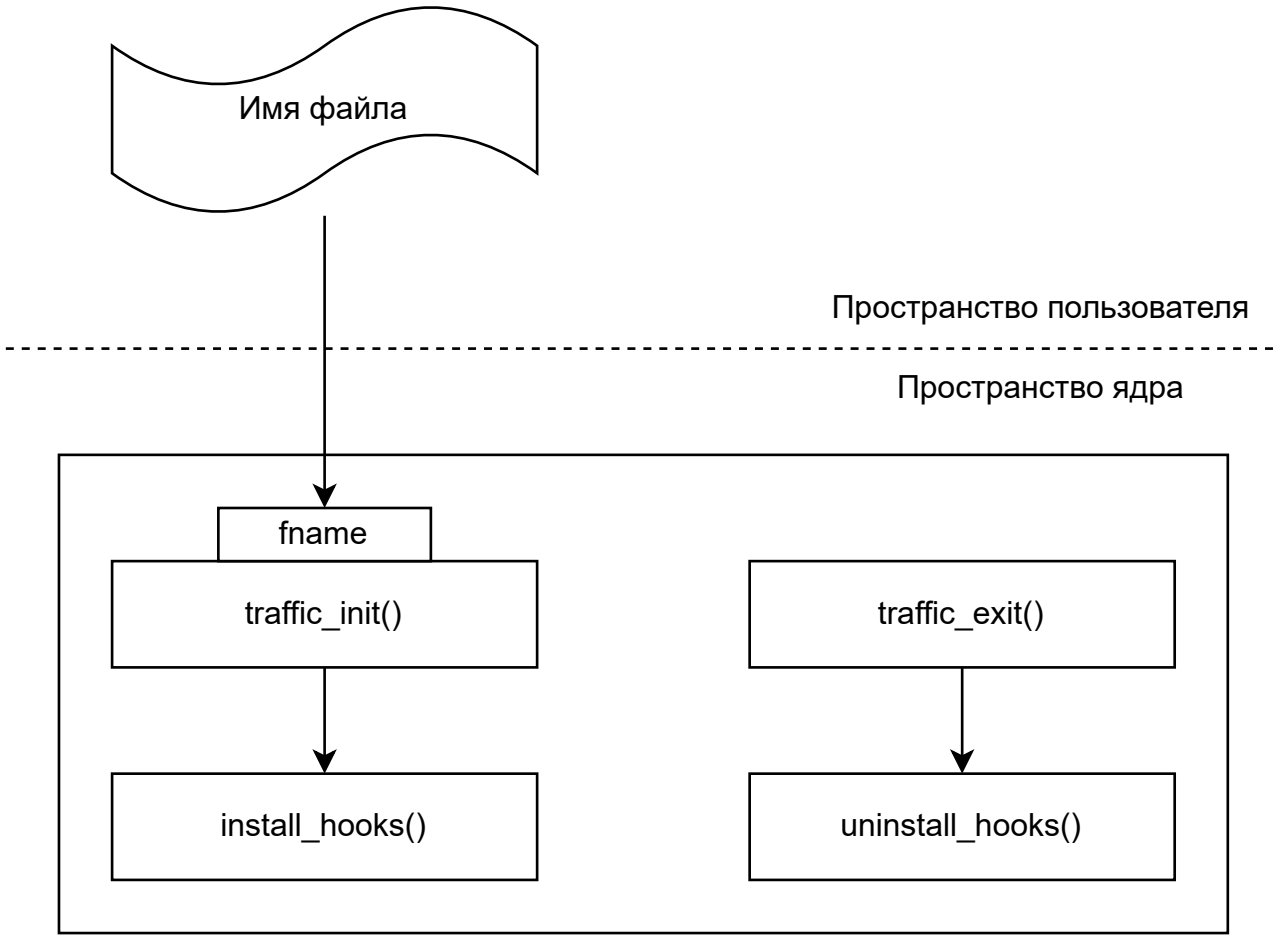

Рисунок 2.6 – Структура ПО

3 Технологический раздел

3.1 Выбор языка и среды программирования

Для реализации программного обеспечения был выбран язык программирования C, так как на нем написано ядро Linux, и он используется для написания загружаемых модулей ядра.

Для написания кода был выбран текстовый редактор Sublime Text 3, т. к. он обладает следующими преимуществами [6]:

1) кроссплатформенность (доступность на Linux Ubuntu);

2) подсветка синтаксиса (в частности, для языка C);

3) настраиваемый интерфейс;

4) множественное выделение (например, переменных);

5) автоматическое заполнение.

3.2 Используемые структуры

Для реализации загружаемого модуля ядра были разработаны две структуры:

— struct hook (листинг 3.1). Поле name содержит название функции ядра (например, sys_send). Поле function — указатель на функцию, которую нужно вызвать вместо исходной. Поле original указывает на исходную функцию ядра. Поле address — адрес исходной функции ядра. А поле ops — экземпляр структуры struct ftrace_ops.

Листинг 3.1 – Структура struct hook

```
struct hook {
    const char*       name;
    void*             function;
    void*             original;
    unsigned long     address;
    struct ftrace_ops ops;
};
```

— struct net_traffic (листинг 3.2). Поле pid содержит идентификатор отслеживаемого процесса. Изначально значение этого поля равно -1. После

того, как нужная программа будет запущена, поле pid проинициализируется идентификатором отслеживаемого процесса. Поля bytes_received и bytes_sent хранят объем соответственно входящих и исходящих пакетов в байтах.

Листинг 3.2 – Структура struct net_traffic

```
struct net_traffic {
    pid_t    pid;
    uint64_t bytes_received;
    uint64_t bytes_sent;
};
```

3.3 Реализация алгоритма получения аргументов командной строки в режиме ядра

В листинге 3.3 показана реализация алгоритма получения аргументов командной строки в режиме ядра.

Листинг 3.3 – Реализация алгоритма получения аргументов командной строки в режиме ядра

```
static int __init traffic_init(void)
{
    if (strlen(fname) < 1)
    {
        printk(KERN_INFO "Traffic module: error - incorrect
           filename.\n");
        return -1;
    }
    int err = install_hooks(hook_array, ARRAY_SIZE(hook_array));
    if (!err)
    {
        printk(KERN_INFO "Traffic module: loaded.\n");
        printk(KERN_INFO "Traffic module: program %s is
           monitored.\n", fname);
    }
    return err;
}
```

3.4 Реализация алгоритма определения целевого процесса

В листинге 3.4 показана реализация алгоритма поиска процесса, чей сетевой трафик должен отслеживать загружаемый модуль ядра.

Листинг 3.4 – Реализация алгоритма определения целевого процесса

```
static asmlinkage long (*original_sys_execve)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_execve(struct pt_regs *regs)
{
    if (statistics.pid == -1)
    {
        char *kernel_fname = dup_filename((void *)regs->di);
        char *fname_ = kernel_fname + strlen(kernel_fname) - 1;
        while (fname_ >= kernel_fname && *fname_ != '/')
            --fname_;
        if (strcmp(fname, ++fname_) == 0)
        {
            statistics.pid = current->pid;
            printk(KERN_INFO "Traffic module: program %s with
               PID = %d executed.\n", fname, statistics.pid);
        }
        kfree(kernel_fname);
    }
    return original_sys_execve(regs);
}
```

3.5 Реализация алгоритма получения размера пакета

В листинге 3.5 показана реализация алгоритма получения размера отправленного пакета на примере перехвата функции ядра sys_send().

Листинг 3.5 – Реализация алгоритма получения размера пакета

```
static asmlinkage long (*original_sys_send)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_send(struct pt_regs *regs)
{
    long bytes = original_sys_send(regs);
    pid_t pid = current->pid;
```

```
    if (pid == statistics.pid && bytes > 0)
    {
        statistics.bytes_sent += bytes;
        printk(KERN_INFO "Traffic module: process sent %ld
           bytes.\n", bytes);
    }
    return bytes;
}
```

3.6 Реализация алгоритма регистрации перехватчика

В листинге 3.6 показана реализация алгоритма регистрации перехватчика.

Листинг 3.6 – Реализация алгоритма регистрации перехватчика

```
int install_hook(struct hook *hook)
{
    int err = resolve_hook_address(hook);
    if (err)
        return err;
    hook->ops.func = ftrace_thunk;
    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                    | FTRACE_OPS_FL_RECURSION
                    | FTRACE_OPS_FL_IPMODIFY;
    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err)
    {
        printk(KERN_INFO "Traffic module: can't
           ftrace_set_filter_ip: %d\n", err);
        return err;
    }
    err = register_ftrace_function(&hook->ops);
    if (err)
    {
        printk(KERN_INFO "Traffic module: can't
           register_ftrace_function: %d\n", err);
        ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
        return err;
    }
    return 0;
}
```

3.7 Makefile для сборки загружаемого модуля ядра

В листинге 3.7 показан Makefile для сборки загружаемого модуля ядра.

Листинг 3.7 – Makefile для сборки загружаемого модуля ядра

```
ifneq ($(KERNELRELEASE),)
    obj-m := net_traffic.o
else
    CURRENT = $(shell uname -r)
    KDIR = /lib/modules/$(CURRENT)/build
    PWD = $(shell pwd)
default:
    echo $(MAKE) -C $(KDIR) M=$(PWD) modules
    $(MAKE) -C $(KDIR) M=$(PWD) modules
    make clean
clean:
    @rm -f *.o .*.cmd .*.flags *.mod.c *.order
    @rm -f .*.*.cmd *~ *.*~ TODO.*
    @rm -fR .tmp*
    @rm -rf .tmp_versions
disclean: clean
    @rm *.ko *.symvers
endif
```

4 Исследовательский раздел

4.1 Технические характеристики устройства

Технические характеристики устройства, на котором было проведено тестирование реализованного программного обеспечения:

1) операционная система Ubuntu 22.04.3 LTS;
2) оперативная память 4 ГБ;
3) процессор Intel® Core™ i5-4790K @ 4.00 ГГц;
4) версия ядра Linux 6.2.0.

4.2 Тестирование программного обеспечения

Для тестирования реализованного программного обеспечения были написаны две тестовые программы, имеющие архитектуру «клиент—сервер». В первой программе для взаимодействия используются системные вызовы recv() и send(), а во второй — recvfrom() и sendto(). В обеих программах клиенты сообщают серверу свой PID. То же делает сервер — сообщает клиентам свой идентификатор. И сервер, и клиенты выводят длину полученного сообщения. Таким образом можно проверить суммарный размер пакетов, который вывел в системный лог реализованный загружаемый модуль ядра. Также реализованное ПО было протестировано с помощью браузера Mozilla Firefox.

На рисунке 4.1 показан результат работы загружаемого модуля ядра при отсутствии аргумента командной строки.

```
vladyslav@vladyslav-pc:~/ics7-os-cp/src$ sudo insmod net_traffic.ko
insmod: ERROR: could not insert module net_traffic.ko: Operation not p
ermitted
vladyslav@vladyslav-pc:~/ics7-os-cp/src$ sudo dmesg
[ 3364.793229] Traffic module: error - incorrect filename.
vladyslav@vladyslav-pc:~/ics7-os-cp/src$
```

Рисунок 4.1 – Результат работы загружаемого модуля ядра при отсутствии аргумента командной строки

На рисунке 4.4 показан результат работы загружаемого модуля ядра, который мониторил сетевой трафик процесса-сервера, использующего систем-

ные вызовы recv() и send(). На рисунках 4.2–4.3 показаны размеры пакетов, которые соответственно получал и отправлял отслеживаемый процесс.

```
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/send_recv$ ./server.out
Server received: message from client with PID = 5098.
Client message length: 35.
Server received: message from client with PID = 5099.
Client message length: 35.
Server received: message from client with PID = 5100.
Client message length: 35.
Server received: message from client with PID = 5101.
Client message length: 35.
Server received: message from client with PID = 5102.
Client message length: 35.
^CServer exited.
```

Рисунок 4.2 – Размеры пакетов, которые получал отслеживаемый процесс

```
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/send_recv$ ./client.out
Client received: message from server with PID = 5089.
Server message length: 35.
Client exited.
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/send_recv$ ./client.out
Client received: message from server with PID = 5089.
Server message length: 35.
Client exited.
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/send_recv$ ./client.out
Client received: message from server with PID = 5089.
Server message length: 35.
Client exited.
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/send_recv$ ./client.out
Client received: message from server with PID = 5089.
Server message length: 35.
Client exited.
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/send_recv$ ./client.out
Client received: message from server with PID = 5089.
Server message length: 35.
Client exited.
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/send_recv$
```

Рисунок 4.3 – Размеры пакетов, которые отправлял отслеживаемый процесс

```
vladyslav@vladyslav-pc:~/ics7-os-cp/src$ sudo dmesg
[ 3466.073857] Traffic module: loaded.
[ 3466.073860] Traffic module: program server.out is monitored.
[ 3469.037256] Traffic module: program server.out with PID = 5089 executed.
[ 3475.449400] Traffic module: process received 35 bytes.
[ 3475.449598] Traffic module: process sent 35 bytes.
[ 3476.644215] Traffic module: process received 35 bytes.
[ 3476.644311] Traffic module: process sent 35 bytes.
[ 3477.253262] Traffic module: process received 35 bytes.
[ 3477.253348] Traffic module: process sent 35 bytes.
[ 3477.886604] Traffic module: process received 35 bytes.
[ 3477.886702] Traffic module: process sent 35 bytes.
[ 3478.569248] Traffic module: process received 35 bytes.
[ 3478.569455] Traffic module: process sent 35 bytes.
[ 3481.878480] Traffic module: process with PID = 5089 exited.
[ 3481.878487] Traffic module: received 175 bytes.
[ 3481.878488] Traffic module: sent     175 bytes.
[ 3537.399637] Traffic module: unloaded.
```

Рисунок 4.4 – Результат работы загружаемого модуля ядра на примере системных вызовов recv() и send()

На рисунке 4.7 показан результат работы загружаемого модуля ядра, который мониторил сетевой трафик процесса-сервера, использующего системные вызовы recvfrom() и sendto(). На рисунках 4.5–4.6 показаны размеры пакетов, которые соответственно получал и отправлял отслеживаемый процесс.

```
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/sendto_recvfrom$ ./server.out
Server received: message from client with PID = 5650.
Client message length: 35.
Server received: message from client with PID = 5651.
Client message length: 35.
Server received: message from client with PID = 5652.
Client message length: 35.
^CServer exited.
```

Рисунок 4.5 – Размеры пакетов, которые получал отслеживаемый процесс

```
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/sendto_recvfrom$ ./client.out
Client received: message from server with PID = 5649.
Server message length: 35.
Client exited.
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/sendto_recvfrom$ ./client.out
Client received: message from server with PID = 5649.
Server message length: 35.
Client exited.
vladyslav@vladyslav-pc:~/ics7-os-cp/tests/sendto_recvfrom$ ./client.out
Client received: message from server with PID = 5649.
Server message length: 35.
Client exited.
```

Рисунок 4.6 – Размеры пакетов, которые отправлял отслеживаемый процесс

```
[ 3944.937358] Traffic module: loaded.
[ 3944.937363] Traffic module: program server.out is monitored.
[ 3947.877179] Traffic module: program server.out with PID = 5649 executed.
[ 3950.445750] Traffic module: process received 35 bytes.
[ 3950.445945] Traffic module: process sent 35 bytes.
[ 3951.328660] Traffic module: process received 35 bytes.
[ 3951.328746] Traffic module: process sent 35 bytes.
[ 3951.830600] Traffic module: process received 35 bytes.
[ 3951.830741] Traffic module: process sent 35 bytes.
[ 3953.334512] Traffic module: process with PID = 5649 exited.
[ 3953.334517] Traffic module: received 105 bytes.
[ 3953.334517] Traffic module: sent     105 bytes.
[ 3958.714959] Traffic module: unloaded.
```

Рисунок 4.7 – Результат работы загружаемого модуля ядра на примере системных вызовов recvfrom() и sendto()

На рисунках 4.8–4.9 показан результат работы загружаемого модуля ядра, который мониторил сетевой трафик браузера Mozilla Firefox.

```
vladyslav@vladyslav-pc:~/ics7-os-cp/src$ sudo insmod net_traffic.ko fname=firefox
vladyslav@vladyslav-pc:~/ics7-os-cp/src$ sudo dmesg
[ 4073.529217] Traffic module: loaded.
[ 4073.529221] Traffic module: program firefox is monitored.
[ 4075.579227] Traffic module: program firefox with PID = 5675 executed.
[ 4075.640844] Traffic module: process sent 1 bytes.
```

Рисунок 4.8 – Вывод в системный лог при загрузке модуля

```
[ 4123.068450] Traffic module: process sent 72 bytes.
[ 4123.068453] Traffic module: process sent 72 bytes.
[ 4123.068550] Traffic module: process sent 76 bytes.
[ 4123.068564] Traffic module: process sent 72 bytes.
[ 4123.068568] Traffic module: process sent 72 bytes.
[ 4123.068572] Traffic module: process sent 72 bytes.
[ 4123.068576] Traffic module: process sent 72 bytes.
[ 4123.068579] Traffic module: process sent 72 bytes.
[ 4123.083292] Traffic module: process sent 72 bytes.
[ 4123.083315] Traffic module: process sent 64 bytes.
[ 4123.084432] Traffic module: process sent 72 bytes.
[ 4123.084738] Traffic module: process sent 64 bytes.
[ 4123.085305] Traffic module: process sent 72 bytes.
[ 4123.085349] Traffic module: process sent 64 bytes.
[ 4123.085716] Traffic module: process sent 72 bytes.
[ 4123.085732] Traffic module: process sent 64 bytes.
[ 4123.404946] Traffic module: process sent 72 bytes.
[ 4123.466668] Traffic module: process sent 72 bytes.
[ 4123.466693] Traffic module: process sent 64 bytes.
[ 4123.587847] Traffic module: process with PID = 5675 exited.
[ 4123.587852] Traffic module: received 283565 bytes.
[ 4123.587853] Traffic module: sent     1317693 bytes.
[ 4172.728330] Traffic module: unloaded.
```

Рисунок 4.9 – Результат работы загружаемого модуля ядра на примере браузера Mozilla Firefox

Выводы из исследовательского раздела

Для тестирования реализованного ПО были написаны две программы, имеющие архитектуру «клиент—сервер», и взаимодействующие с помощью системных вызовов send() и recv() в первом случае и sendto() и recvfrom() во втором. Также загружаемый модуль ядра был протестирован с помощью браузера Mozilla Firefox. Все тесты пройдены успешно.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы был реализован загружаемый модуль ядра для мониторинга сетевого трафика процесса. Реализованное ПО принимает на вход имя файла в качестве аргумента командной строки, перехватывает функцию sys_execve(), определяет целевой процесс и мониторит его сетевой трафик.

Выполнены следующие задачи:

1) проведен сравнительный анализ существующих методов перехвата функций ядра Linux и выбран наиболее подходящий для достижения поставленной цели — фреймворк ftrace;

2) разработано программное обеспечение для мониторинга сетевого трафика процесса;

3) реализовано ПО для мониторинга сетевого трафика процесса;

4) протестировано реализованное программное обеспечение.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Левин Д. В.* Утилита strace. — (Дата обращения: 29.10.2023). https://strace.io/.
2. *Хелман А.* Перехват системных вызовов с помощью ptrace. — 2011. — (Дата обращения: 05.11.2023). https://habr.com/ru/articles/111266/.
3. *LinuxDocs.* Using ftrace to hook to functions. — 2011. — (Дата обращения: 05.11.2023). https://www.kernel.org/doc/html/v5.0/trace/ftrace-uses.html.
4. *Salzman P. J.*, *Burian M.*, *Pomerantz O.* Passing Command Line Arguments to a Module. — (Дата обращения: 29.10.2023). https://tldp.org/LDP/lkmpg/2.6/html/x323.html.
5. *Bootlin.* Elixir Cross Referencer (syscalls.h). — (Дата обращения: 05.11.2023). https://elixir.bootlin.com/linux/v4.8/source/include/linux/syscalls.h.
6. *SublimeLtd.* Sublime Text. — (Дата обращения: 10.11.2023). https://www.sublimetext.com/.

ПРИЛОЖЕНИЕ А

Руководство пользователя

Для мониторинга сетевого трафика процесса с помощью реализованного загружаемого модуля ядра необходимо:

1) с помощью утилиты make и файла Makefile собрать загружаемый модуль ядра;

2) загрузить модуль с помощью команды sudo insmod net_traffic.ko fname=NAME, где NAME — имя файла;

3) запустить файл с именем NAME;

4) после завершения отслеживаемого процесса, нужно посмотреть системный лог с помощью команды sudo dmesg;

5) с помощью команды sudo rmmod net_traffic выгрузить модуль.

В системном логе можно увидеть следующую информацию: имя файла, идентификатор отслеживаемого процесса, суммарный объем отправленных и полученных процессом пакетов в байтах. Также в системный лог выводится размер каждого отправленного и принятого процессом пакета.

ПРИЛОЖЕНИЕ Б

Исходный код загружаемого модуля ядра

В листинге 4.1 показан исходный код реализованного загружаемого модуля ядра для мониторинга сетевого трафика процесса.

Листинг 4.1 – Исходный код реализованного загружаемого модуля ядра

```
#include <linux/ftrace.h>
#include <linux/kallsyms.h>
#include <linux/kernel.h>
#include <linux/linkage.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/version.h>
#include <linux/kprobes.h>
#include <linux/sched.h>
#include <linux/net.h>
#include <linux/in.h>

#define OFFSET 0

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("LKM for monitoring network traffic of a
   process");

struct hook
{
    const char*       name;
    void*             function;
    void*             original;
    unsigned long     address;
    struct ftrace_ops ops;
};

struct net_traffic
{
    pid_t    pid;
    uint64_t bytes_received;
    uint64_t bytes_sent;
};
```

```
struct net_traffic statistics = { -1, 0, 0 };

static char *fname = "";
module_param(fname, charp, 0);

static unsigned long lookup_name(const char *name)
{
    struct kprobe kp =
    {
        .symbol_name = name
    };
    unsigned long address;
    if (register_kprobe(&kp) < 0)
        return 0;
    address = (unsigned long)kp.addr;
    unregister_kprobe(&kp);
    return address;
}

static int resolve_hook_address(struct hook *hook)
{
    hook->address = lookup_name(hook->name);
    if (!hook->address)
    {
        printk(KERN_INFO "Traffic module: can't hook syscall:
           %s.\n", hook->name);
        return -ENOENT;
    }
#if OFFSET
    *((unsigned long*) hook->original) = hook->address +
       MCOUNT_INSN_SIZE;
#else
    *((unsigned long*) hook->original) = hook->address;
#endif
    return 0;
}

static void notrace ftrace_thunk(unsigned long ip, unsigned long
   parent_ip,
    struct ftrace_ops *ops, struct ftrace_regs *fregs)
```

```
{
    struct pt_regs *regs = ftrace_get_regs(fregs);
    struct hook *hook = container_of(ops, struct hook, ops);
#if OFFSET
    regs->ip = (unsigned long)hook->function;
#else
    if (!within_module(parent_ip, THIS_MODULE))
        regs->ip = (unsigned long)hook->function;
#endif
}

int install_hook(struct hook *hook)
{
    int err = resolve_hook_address(hook);
    if (err)
        return err;
    hook->ops.func = ftrace_thunk;
    hook->ops.flags = FTRACE_OPS_FL_SAVE_REGS
                    | FTRACE_OPS_FL_RECURSION
                    | FTRACE_OPS_FL_IPMODIFY;
    err = ftrace_set_filter_ip(&hook->ops, hook->address, 0, 0);
    if (err)
    {
        printk(KERN_INFO "Traffic module: can't
           ftrace_set_filter_ip: %d\n", err);
        return err;
    }
    err = register_ftrace_function(&hook->ops);
    if (err)
    {
        printk(KERN_INFO "Traffic module: can't
           register_ftrace_function: %d\n", err);
        ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
        return err;
    }
    return 0;
}

void uninstall_hook(struct hook *hook)
{
    int err = unregister_ftrace_function(&hook->ops);
```

```
    if (err)
        printk(KERN_INFO "Traffic module: can't
           unregister_ftrace_function: %d\n", err);
    err = ftrace_set_filter_ip(&hook->ops, hook->address, 1, 0);
    if (err)
        printk(KERN_INFO "Traffic module: can't
           ftrace_set_filter_ip: %d\n", err);
}

int install_hooks(struct hook *hooks, size_t count)
{
    int err = 0;
    size_t i;
    for (i = 0; err == 0 && i < count; i++)
        err = install_hook(&hooks[i]);
    if (err)
        while (i != 0)
            uninstall_hook(&hooks[--i]);
    return err;
}

void uninstall_hooks(struct hook *hooks, size_t count)
{
    for (size_t i = 0; i < count; i++)
        uninstall_hook(&hooks[i]);
}

#if defined(CONFIG_X86_64) && (LINUX_VERSION_CODE >=
   KERNEL_VERSION(4,17,0))
#define PTREGS_SYSCALL_STUBS 1
#endif

static char *dup_filename(const char __user *filename)
{
    char *kernel_fname = kmalloc(4096, GFP_KERNEL);
    if (!kernel_fname)
        return NULL;
    if (strncpy_from_user(kernel_fname, filename, 4096) < 0)
    {
        kfree(kernel_fname);
        return NULL;
```

```
    }
    return kernel_fname;
}

static asmlinkage long (*original_sys_send)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_send(struct pt_regs *regs)
{
    long bytes = original_sys_send(regs);
    pid_t pid = current->pid;
    if (pid == statistics.pid && bytes > 0)
    {
        statistics.bytes_sent += bytes;
        printk(KERN_INFO "Traffic module: process sent %ld
           bytes.\n", bytes);
    }
    return bytes;
}

static asmlinkage long (*original_sys_sendto)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_sendto(struct pt_regs *regs)
{
    long bytes = original_sys_sendto(regs);
    pid_t pid = current->pid;
    if (pid == statistics.pid && bytes > 0)
    {
        statistics.bytes_sent += bytes;
        printk(KERN_INFO "Traffic module: process sent %ld
           bytes.\n", bytes);
    }
    return bytes;
}

static asmlinkage long (*original_sys_sendmsg)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_sendmsg(struct pt_regs *regs)
{
```

```
    long bytes = original_sys_sendmsg(regs);
    pid_t pid = current->pid;
    if (pid == statistics.pid && bytes > 0)
    {
        statistics.bytes_sent += bytes;
        printk(KERN_INFO "Traffic module: process sent %ld
           bytes.\n", bytes);
    }
    return bytes;
}

static asmlinkage long (*original_sys_recv)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_recv(struct pt_regs *regs)
{
    long bytes = original_sys_recv(regs);
    pid_t pid = current->pid;
    if (pid == statistics.pid && bytes > 0)
    {
        statistics.bytes_received += bytes;
        printk(KERN_INFO "Traffic module: process received %ld
           bytes.\n", bytes);
    }
    return bytes;
}

static asmlinkage long (*original_sys_recvfrom)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_recvfrom(struct pt_regs *regs)
{
    long bytes = original_sys_recvfrom(regs);
    pid_t pid = current->pid;
    if (pid == statistics.pid && bytes > 0)
    {
        statistics.bytes_received += bytes;
        printk(KERN_INFO "Traffic module: process received %ld
           bytes.\n", bytes);
    }
    return bytes;
```

```
}

static asmlinkage long (*original_sys_recvmsg)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_recvmsg(struct pt_regs *regs)
{
    long bytes = original_sys_recvmsg(regs);
    pid_t pid = current->pid;
    if (pid == statistics.pid && bytes > 0)
    {
        statistics.bytes_received += bytes;
        printk(KERN_INFO "Traffic module: process received %ld
           bytes.\n", bytes);
    }
    return bytes;
}

static asmlinkage long (*original_sys_execve)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_execve(struct pt_regs *regs)
{
    if (statistics.pid == -1)
    {
        char *kernel_fname = dup_filename((void *)regs->di);
        char *fname_ = kernel_fname + strlen(kernel_fname) - 1;
        while (fname_ >= kernel_fname && *fname_ != '/')
        {
            --fname_;
        }
        if (strcmp(fname, ++fname_) == 0)
        {
            statistics.pid = current->pid;
            printk(KERN_INFO "Traffic module: program %s with
               PID = %d executed.\n", fname, statistics.pid);
        }
        kfree(kernel_fname);
    }
    return original_sys_execve(regs);
}
```

```
static asmlinkage long (*original_sys_exit)(struct pt_regs
   *regs);

static asmlinkage long hook_sys_exit(struct pt_regs *regs)
{
    if (statistics.pid == current->pid)
    {
        printk(KERN_INFO "Traffic module: process with PID = %d
           exited.\n", statistics.pid);
        printk(KERN_INFO "Traffic module: received %d bytes.\n",
           statistics.bytes_received);
        printk(KERN_INFO "Traffic module: sent     %d bytes.\n",
           statistics.bytes_sent);
    }
    return original_sys_exit(regs);
}

#ifdef PTREGS_SYSCALL_STUBS
#define SYSCALL_NAME(name) ("__x64_" name)
#else
#define SYSCALL_NAME(name) (name)
#endif

#define HOOK(_name, _function, _original) \
{                                 \
    .name = SYSCALL_NAME(_name), \
    .function = (_function),     \
    .original = (_original),     \
}

static struct hook hook_array[] =
{
    HOOK("sys_execve", hook_sys_execve, &original_sys_execve),
    HOOK("sys_send", hook_sys_send, &original_sys_send),
    HOOK("sys_sendto", hook_sys_sendto, &original_sys_sendto),
    HOOK("sys_sendmsg", hook_sys_sendmsg, &original_sys_sendmsg),
    HOOK("sys_recv", hook_sys_recv, &original_sys_recv),
    HOOK("sys_recvfrom", hook_sys_recvfrom,
       &original_sys_recvfrom),
    HOOK("sys_recvmsg", hook_sys_recvmsg, &original_sys_recvmsg),
```

```
    HOOK("sys_exit_group", hook_sys_exit, &original_sys_exit),
};

static int __init traffic_init(void)
{
    if (strlen(fname) < 1)
    {
        printk(KERN_INFO "Traffic module: error - incorrect
           filename.\n");
        return -1;
    }
    int err = install_hooks(hook_array, ARRAY_SIZE(hook_array));
    if (!err)
    {
        printk(KERN_INFO "Traffic module: loaded.\n");
        printk(KERN_INFO "Traffic module: program %s is
           monitored.\n", fname);
    }
    return err;
}

static void __exit traffic_exit(void)
{
    uninstall_hooks(hook_array, ARRAY_SIZE(hook_array));
    printk(KERN_INFO "Traffic module: unloaded.\n");
}

module_init(traffic_init);
module_exit(traffic_exit);
```