Санкт-Петербургский государственный университет

Программная инженерия

Ковалев Марк Германович

Отслеживание и анализ пути сетевых пакетов в ядре Linux

Бакалаврская работа

Научный руководитель:
проф. каф. СП, д.ф.-м.н., проф. А.Н. Терехов

Научный консультант:
асс. каф. СП А.П. Козлов

Рецензент:
Инженер-программист ООО «Азул Системс» А.К. Петушков

Санкт-Петербург
2020

Saint Petersburg State University

Software Engineering

Mark Kovalev

Tracing and analysis of path of network packets in the Linux kernel

Bachelor's Thesis

Scientific supervisor:
Professor, PhD Andrey Terekhov

Scientific advisor:
Assistant Anton Kozlov

Reviewer:
Software engineer at Azul Systems, LLC Andrey Petushkov

Saint Petersburg
2020

Оглавление

Введение 5

1. Постановка задачи 7

2. Требования к реализации 8

3. Обзор 10

3.1 Подходы к отслеживанию пути сетевого пакета 11

3.1.1 Модификация кода ядра 11

3.1.2 Зондирование ядра 12

3.1.3 Точки трассировки 13

3.1.4 ftrace 13

3.1.5 Extended Berkeley Packet Filter 14

3.1.6 Вывод 14

3.2 Обзор технологии eBPF 15

3.3 Жизненный цикл eBPF программ 17

3.3.1 Компиляция и загрузка 17

3.3.2 Ограниченный язык программирования C 18

3.3.3 Верификатор 18

4. Реализация утилиты 20

4.1 Алгоритм работы 20

4.2 eBPF программы 22

4.2.1 eBPF классификатор 22

4.2.2 eBPF зонды 23

4.3 eBPF массивы 25

4.4 Загрузка и прикрепление eBPF программ 26

4.5 Разница контекстов точек крепления 26

4.6 Особенности реализации 27

4.7 Демонстрация работы прототипа 27

Заключение **29**

Список литературы **30**

Введение

Рост человеческих потребностей вкупе с развитием технологий способствуют созданию более продвинутых программных и аппаратных решений. Основополагающим элементом таких решений зачастую является обширная и комплексная сетевая инфраструктура. Множество различных устройств и виртуальных интерфейсов, сетевые пространства имён, настройки маршрутизации, фильтрации пакетов и межсетевого экрана, разнообразные сетевые протоколы — всё это позволяет создавать системы с ранее невозможной функциональностью и предоставлять пользователям новые сервисы. Естественным образом при разработке и сопровождении таких систем разработчики и администраторы сталкиваются с множеством проблем.

Обычно для отладки сетевых ошибок необходимо проверить все узлы, участвующие в сетевом взаимодействии: отправляющий, связующие и принимающий. Однако из-за сложной конфигурации, в сети возникают неочевидные связи и взаимодействий между различными элементами, что сильно осложняет процесс поиска источника неполадки даже в рамках одного узла. Неизбежно возникнет ситуация, в которой непонятно, с какой стороны подступиться проблеме. Тогда разработчику придётся перебирать возможные причины возникновения ошибки, используя весь набор доступных для сетевой отладки инструментов и полагаясь лишь на свой профессиональный опыт или интуицию. Такой бессистемный подход приводит к высокой стоимости устранения сбоев.

За всю историю развития технологий было создано множество утилит для конфигурации сетевой подсистемы, которые также используются в процессе поиска и решения сетевых проблем. Наиболее распространенным и часто используемыми из них являются `iproute2`, `ethtool`, `ping`, `traceroute`, `nslookup`, `netcat`, `iptables`, `tcpdump` [1]. С их помощью можно узнать конфигурацию сетевых интерфейсов, проверить наличие соединения и работоспособность DNS, просмотреть таблицу маршрутизации, изучить содержимое сетевых пакетов и многое другое. Однако все эти инструменты заточены под определённые задачи и их функциональность ограничена программным интерфейсом, предоставленным ядром ОС. Только при использовании комбинации из множества этих инструментов разработчик может получить общий взгляд на процесс обработки сетевого трафика в рамках

конкретного узла, что помогло бы определить область, в которой возникает проблема.

Частой причиной неполадок в комплексных сетевых инфраструктурах является неправильно определённая конфигурация для некоторого вида сетевых пакетов. Необходимый трафик в систему приходит, но обрабатываются не так, как было задумано архитектурой сетевой системы. В таких ситуациях скорость отладки можно было бы увеличить, используя знания о реализации сетевых механизмов и принимаемых ими решениях. Эту информацию можно получить, анализируя пути сетевых пакетов через ядро ОС. Таким образом можно точно определить, в какой момент процесс обработки пакетов отклоняется от предполагаемого, и станет известно, какую сетевую подсистему стоит изучать в поисках источника неполадки. Подход к созданию утилиты, способной отследить путь сетевого пакета будет рассмотрен в этой работе. Здесь и далее под путём сетевого пакета подразумевается порядок вызова функций ядра ОС при его обработке.

1. Постановка задачи

Целью данной дипломной работы является разработка прототипа утилиты, способной предоставить информацию о пути сетевого пакета в ядре Linux. Для достижения этой цели были сформулированы следующие задачи:

- Сформулировать требования к реализации утилиты.
- Выбрать подход к отслеживанию пути сетевого пакета, подходящий для реализации утилиты в рамках выставленных требований.
- Разработать прототип, демонстрирующий работоспособность утилиты.

2. Требования к реализации

Для решения сетевых проблем не существует формальных методик и алгоритмов. Создание такой документации осложнено многообразием потенциальных сетевых конфигураций. Поэтому, каждый сетевой администратор обычно имеет свои наборы инструментов и подходы к сетевой отладке. Эту информацию обычно можно найти только в личных блогах [2, 3, 4, 5].

После анализа источников, были выделены и сформулированы общие свойства, которыми обладают все полезные инструменты, которые чаще всего используются для сетевой отладки. Ниже перечислены требования, которым должна удовлетворять утилита, чтобы иметь возможность использования при решении проблем в реальных условиях.

Взаимодействие с утилитой должно быть реализовано через интерфейс командной строки и происходить по следующему сценарию:

1. Через стандартный поток ввода утилита получает выражение-фильтр, с помощью которого можно однозначно определить наблюдаемый сетевой трафик (например, задать протокол транспортного уровня, IP адрес отправителя или номер VLAN группы).

2. По завершению наблюдения, утилита выводит путь пакета из трафика, определённого фильтром, в текстовом виде в стандартный поток вывода.

Такую утилиту можно будет использовать как напрямую из командной строки, сразу получая путь сетевого пакета, так и интегрировать в сценарии развёртывания и тестирования систем, отправляя всю необходимую информацию в журнал для дальнейшего изучения.

Также, для использования на промышленных системах, реализация утилиты должна обладать следующими свойствами:

- Безопасность — при использовании утилиты вероятность нарушения работоспособности системы должна быть сведена к минимуму.

- Запуск на рабочей системе — утилита не должна требовать изменения конфигурации ядра системы и его пересборки.

- Работа при реальной нагрузке — отслеживание реально существующего сетевого трафика, а не создание синтетического. Не допускается ограничение или изоляция трафика.

- Низкие накладные расходы — влияние на производительность рабочей системы должно быть сведено к минимуму.

- Независимость от конкретной сборки ядра — возможность адаптации утилиты для запуска на разных конфигурациях систем и разных версиях ядра Linux.

3. Обзор

Для понимания того, как информация о пути сетевого пакета может помочь при отладке проблем в комплексных сетевых инфраструктурах, рассмотрим систему, упрощённая конфигурация которой представлена на рис. 1.

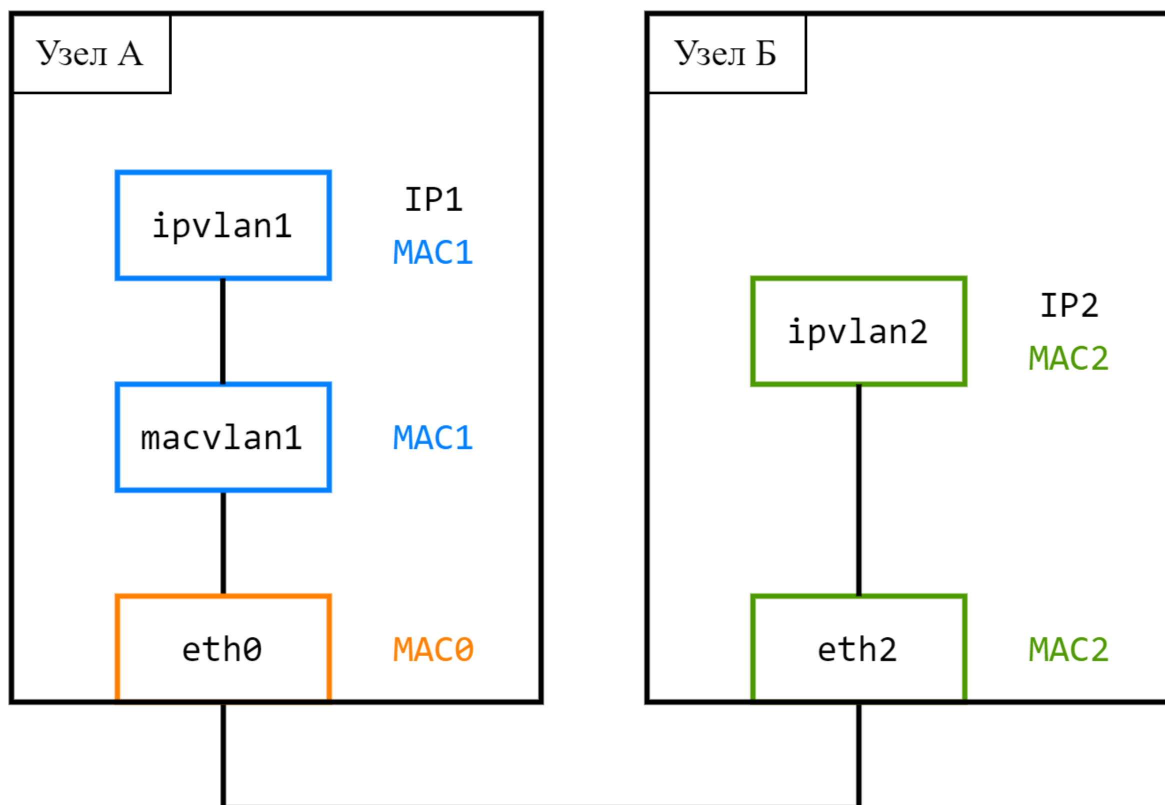

Рис. 1 Конфигурации системы из двух узлов

Использование утилиты `ping` подтверждает наличие связи между интерфейсами `ipvlan1` и `ipvlan2`. Неожиданно связь обрывается, ICMP запросы продолжают отправляться, но ответов `ipvlan1` не получает. Через какое-то время связь восстанавливается, ответы снова приходят. Такое поведение наблюдается на протяжении всей работы утилиты `ping`. Все интерфейсы работают исправно, маршрутизация настроена верно, а сама ошибка не воспроизводится какими-то определёнными действиями. При этом программа tcpdump показывает, что ICMP ответы всё это время приходят на узел А, но по какой-то причине не доходят до `ipvlan1`. После проверки содержимого сетевых пакетов выясняется, что при потере связи ICMP ответы отправляются на адрес `MAC0` интерфейса `eth0`, хотя должны отправляться на адрес `MAC1` интерфейса `ipvlan1`. Дальнейшее исследование пакетов во всей локальной сети показывает,

что такое поведение вызывает работа механизма IPv4 Address Conflict Detection[1]. Он предотвращает появления узлов с одинаковыми IP адресами в локальной сети с помощью специальных ARP запросов. Однако, когда такой запрос приходит на узел А, из-за неверной настройки он отвечает, что адрес `IP1` принадлежит устройству с адресом `MAC0`. Этот ответ через широковещательный канал приходит в узел Б, и он начинает отправлять ICMP ответы с интерфейса `ipvlan2` на адрес `MAC0`, из-за чего связь в утилите `ping` прерывается. Через какое-то время, узел Б генерирует обычный ARP запрос на адрес `IP1` и получает в ответ правильный адрес `MAC1`, вследствие чего связь восстанавливается.

Ключевым этапом на пути к решению описанной проблемы является понимание того, что ICMP ответы всегда приходят на Узел А, но в какие-то моменты обрабатываются не так, как задумано конфигурацией системы. При сравнении путей ICMP ответов на Узле А до и во время пропадания связи, можно было бы заметить, что во втором случае процесс обработки не доходит до интерфейса `ipvlan1` и пакет сбрасывается ещё на интерфейсе `eth0`. Ядро поддерживает представление об иерархии сетевых интерфейсов и при получении пакетов для адреса `MAC1` на `eth0` передаёт их на обработку на `macvlan1`. Если этого не происходит и пакет обрабатывается интерфейсом `eth0`, это означает, что MAC-адрес получателя — `MAC0`. Таким образом, становится понятно, что проблема возникает из-за изменения MAC-адреса в ICMP ответах, приходящих с узла Б. Эта информация сужает область поиска источника проблемы и ускоряет процесс отладки системы.

3.1 Подходы к отслеживанию пути сетевого пакета

В этой главе рассматриваются подходы, с помощью которых можно отследить путь сетевого пакета в ядре Linux, и проводится их оценка на основе возможности реализации выставленных к утилите требований при использовании подхода в качестве основного механизма утилиты.

3.1.1 Модификация кода ядра

Самый прямолинейный способ получить какую-либо информацию из ядра Linux — модифицировать его исходный код. Для вывода пути пакета необходимо добавить в начало сетевых функций [6] фильтрацию обрабатываемых ими

[1] https://tools.ietf.org/html/rfc5227

пакетов и, в случае удовлетворения фильтру, вывод имени функции с помощью метода `printk()`. Тогда путь пакета можно будет увидеть в системном журнале.

Это непрактичный подход. Сложность модификации кода ядра и последующей его отладки повышает вероятность возникновения критических ошибок, способных нарушить работоспособность системы, вследствие чего нельзя достичь необходимого уровня безопасности. Модификация и пересборка ядра делают невозможным запуск на рабочей системе. Есть возможность осуществить работу при реальной нагрузке, однако придётся модифицировать вставки кода при смене фильтра на сетевой трафик. Достижение низких накладных расходов осложнено необходимостью фильтровать пакет в каждой функции ядра. Адаптация под изменяемые имена и код функций мешает реализации независимости от конкретной сборки ядра.

3.1.2 Зондирование ядра

Использование зондирования ядра (kernel probe) позволяет запускать пользовательский код до или после исполнения функций ядра Linux [7]. Лишь малая часть функций не поддерживает зондирование, поэтому с его помощью можно исследовать практически всё ядро. Для того, чтобы отследить путь сетевого пакета с помощью этого подхода, необходимо написать модули ядра для всех функций, участвующих в обработке пакета. В каждом модуле обозначается имя соответствующей функции и описывается метод, который будет фильтровать обрабатываемые пакеты.

По сравнению с модификацией кода ядра, при использовании зондирования безопасность возрастает благодаря вынесению функциональности в отдельные модули, однако вероятность нарушить работу ядра всё ещё остаётся слишком высокой. За счёт динамической загрузки модулей ядра появляется возможность запуска на рабочей системе, что всё же осложняется постоянной пересборкой этих модулей при изменении фильтра. Необходимость пересборки модулей также осложняет и работу при реальной нагрузке. Накладные расходы больше за счёт проведения дополнительных операций при передаче управления в описанные методы. Реализация независимости от конкретной сборки ядра упрощается за счёт того, что для разных версий нужно адаптировать только имена функций.

3.1.3 Точки трассировки

Точки трассировки (tracepoints) — это статически определенные места в коде ядра Linux, в которых можно запускать пользовательский код [8]. Их использование для отслеживания пути пакета схоже с использованием зондирования: для всех наблюдаемых точек трассировки создаются модули ядра с реализацией фильтрации сетевых пакетов. Важным отличием точек трассировки является их принадлежность к стабильному API Linux, вследствие чего, если такая точка объявлена, её нельзя убрать или переместить в следующих версиях ядра. Поэтому разработчики ядра стараются держать список точек трассировки минимальным и не добавлять новые без крайней необходимости.

В основном, использование точек трассировки обладает теми же особенностями, что и зондирование ядра. Отличия лишь в том, что понижаются накладные расходы за счёт меньшего количества этих точек и упрощается реализация независимости от конкретной сборки ядра, так как имена этих точек неизменны. Однако информация о пути пакета при использовании этого подхода становится менее подробной.

3.1.4 ftrace

Механизм трассировки ftrace (Function Tracer) позволяет получить различную отладочную информацию о вызовах функций ядра Linux [9]. Принцип его работы заключается в выполнении кода для трассировки в вызовах функции `mcount()`, которые ставятся в начало исполнения всех функций ядра Linux, т.к. при его сборке выставляются флаги «-pg» компилятора `gcc`. При обычной работе все вызовы `mcount()` являются операциями NOP (no operation), вследствие чего они не влияют на производительность системы.

Для отслеживания пути определённого сетевого пакета с помощью ftrace необходимо выбрать трассировщик `function_graph`, показывающий все вызовы из определённых фильтром функций. В фильтре указывается имя функции, участвующей в ранних этапах обработки пакета. Для входящих пакетов это `netif_receive_skb_list_internal`. Далее необходимо ограничить весь трафик в системе, кроме отслеживаемого. После запуска трассировки, путь пакета можно будет наблюдать в файле `/sys/kernel/debug/tracing/trace_pipe`.

ftrace можно использовать на рабочей системе без модификации ядра или дополнительных модулей. У него стабильный интерфейс, не зависящий от

конкретной сборки ядра. Самой же противоречивой особенностью ftrace в рамках данной работы является невозможность загрузить пользовательский код в ядро при трассировке. С одной стороны, благодаря этому значительно повышается безопасность и понижаются накладные расходы. Но с другой — пропадает возможность задать фильтр для наблюдаемого трафика, из-за чего реализация работы при реальной нагрузке становится невозможной.

3.1.5 Extended Berkeley Packet Filter

Расширенный фильтр пакетов Беркли (extended Berkeley Packet Filter, eBPF) — это виртуальная машина внутри ядра Linux с собственным набором инструкций, на которой можно запускать пользовательский код (eBPF программы) в различных местах ядра [10]. Отследить путь сетевого пакета с помощью eBPF можно используя два типа программ: для подсистемы управления трафиком (Linux Traffic Control) и для зондирования функций. Программы первого типа могут отфильтровать отслеживаемый пакет, второго — проследить его обработку функциями ядра.

Статическая проверка и изолированный запуск на виртуальной машине каждой eBPF программы гарантируют безопасность при исполнении. Механизм загрузки программ в ядро без необходимости модификации или создания модулей позволяет реализовать запуск на работающей системе. Изменение программ для фильтрации пакетов позволит осуществить работу при реальной нагрузке. Оптимизированный для JIT компиляции набор инструкций eBPF машины и избавление от фильтрации пакета в каждой зондируемой функции позволят достичь низких накладных расходов. Абстрагирование программного интерфейса eBPF от реализации ядра способствует независимости от конкретной сборки.

3.1.6 Вывод

В табл. 1 представлена оценка всех описанных подходов в зависимости от возможности реализации выставленных к утилите требований:

- «+» — реализация требования возможна.
- «+/–» — реализация требования возможна, но осложнена.
- «–» — реализация требования невозможна либо значительно осложнена по сравнению с остальными подходами.

	Модификация кода ядра	Зондирование ядра	Точки трассировки	ftrace	eBPF
Безопасность	–	+/–	+/–	+	+
Запуск на работающей системе	–	+/–	+/–	+	+
Работа при реальной нагрузке	+/–	+/–	+/–	–	+
Низкие накладные расходы	+/–	+/–	+	+	+
Независимость от сборки ядра	–	+/–	+	+	+

Таблица 1. Сравнение подходов к отслеживанию пути сетевого пакета.

Технология eBPF способствует реализации всех требований, выставленных к утилите, вследствие чего она и была выбрана как основной механизм для отслеживания пути сетевых пакетов.

3.2 Обзор технологии eBPF

eBPF является расширением технологии BPF (BSD Packet Filter, позднее — Berkeley Packet Filter), разработанной в 1992 году [11]. BPF основана на виртуальной машине с двумя 32-битными регистрами и позволяет путём интерпретации запускать фильтры сетевых пакетов, созданные в пространстве пользователя. Её отличительной особенностью являлась высокая производительность, сильно опережавшая существовавшие на тот момент альтернативы. На этой технологии основывается библиотека libpcap и, как следствие, все программы её использующие, такие как tcpdump и Wireshark.

Со временем для технологии BPF появилась возможность JIT компиляции программ и несколько дополнительных применений: интеграция в механизм seccomp, модуль xt_bpf для iptables и классификатор cls_act для подсистемы управления трафиком [12].

eBPF была разработана в 2014 году, в связи с устареванием BPF и массовым переходом на 64-битные платформы[2]. Основными улучшениями являются:

[2] https://lore.kernel.org/patchwork/patch/452162/

- Увеличение количества регистров виртуальной машины до десяти и их расширение до 64-х бит. Это позволило соотносить их один к одному с регистрами процессора, что упростило написание и трансляцию программ.

- Модификация существующих инструкций и добавление новых. Набор инструкций стал приближённым к машинным, что повысило производительность при JIT компиляции.

- Поддержка ассоциативных массивов (eBPF maps) с множеством возможных типов значений. Они позволяют программам обмениваться информацией между друг другом и с пространством пользователя. Эта особенность позволяет добиться большой гибкости самих программ и удобно получать необходимые данные из ядра.

- Расширение набора поддерживаемых точек крепления: классификаторы подсистемы управления трафиком, точки зондирования ядра, точки трассировки, события perf, сокеты.

Возможность безопасно и производительно исполнять пользовательский код практически в любом место ядра Linux поспособствовала созданию на основе eBPF множества современных инструментов для профилирования и отладки, предоставляющих ранее недоступные возможности. Примерами таких инструментов служат проект BCC (BPF Compiler Collection) и утилита bpftrace [13].

Постоянное развитие eBPF разработчиками ядра и добавление новых возможностей показывает заинтересованность сообщества в этой технологии. В будущем она может стать основным инструментом по расширению функциональности Linux без модификации исходных кодов ядра. Об этом свидетельствует появление способа добавления новых возможностей в TCP стек Linux из пространства пользователя с помощью eBPF[3].

[3] https://inl.info.ucl.ac.be/publications/making-linux-tcp-stack-more-extensible-ebpf.html

3.3 Жизненный цикл eBPF программ

Жизненный цикл eBPF программ в общем виде изображён на рис. 2:

1. Файл `prog.c`, содержащий определения eBPF программ и eBPF массивов, компилируется в объектный файл `prog.o` формата ELF.

2. Загрузчик анализирует `prog.o`, формирует структуры `bpf_attr`, определяющие eBPF объекты, и загружает их в ядро.

3. Верификатор проверяет программу и создает файловый дескриптор `prog_fd`, через который программа вызывается в соответствующей точке крепления.

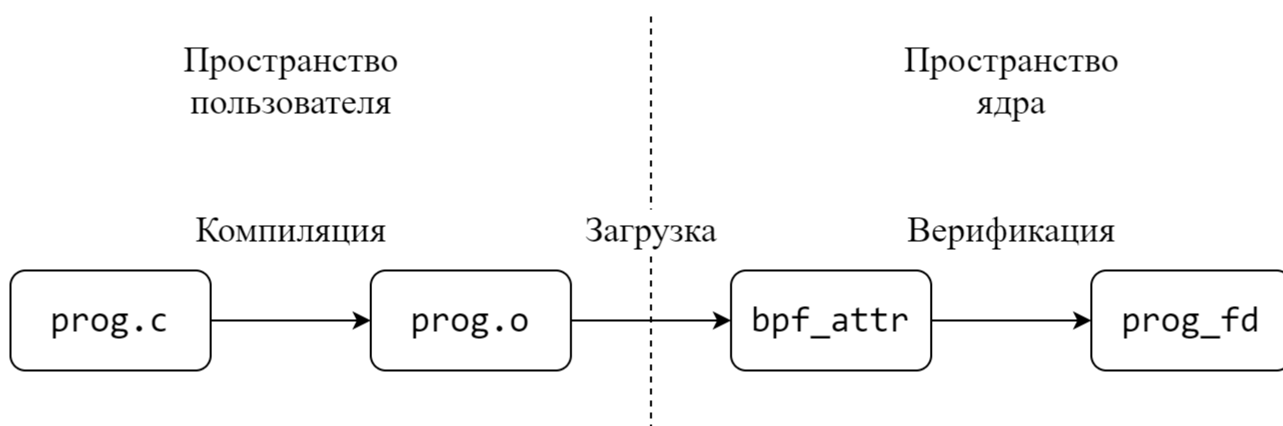

Рис. 2. Жизненный цикл eBPF программы

3.3.1 Компиляция и загрузка

Управление eBPF объектами (программами и массивами), в том числе и их загрузка в ядро, осуществляется через системный вызов `bpf()` [14]. Он принимает eBPF объекты в формате структур `bpf_attr`. Для eBPF программ такая структура будет содержать все инструкции и дополнительную информацию, вроде точки крепления и используемых eBPF массивов. Чтобы не заполнять эти структуры вручную, а определять их на высоком уровне с помощью языка C, была реализована поддержка eBPF в программной инфраструктуре LLVM[4] и наборе компиляторов GCC[5]. Полученный после

[4] https://reviews.llvm.org/D6494
[5] https://gcc.gnu.org/legacy-ml/gcc-patches/2019-08/msg01987.html

компиляции объектный файл формата ELF анализирует специальный загрузчик, составляет структуры `bpf_attr` и загружает их в ядро.

Для создания таких загрузчиков разработчиками Linux развивается библиотека `libbpf` [15], предоставляющая интерфейс над системным вызовом `bpf()`. В частности, на ней основана утилита `bpftool` [16], с помощью которой осуществляется менеджмент eBPF объектов из командной строки. Например, просмотр инструкций загруженных eBPF программ, создание и модификация eBPF массивов.

3.3.2 Ограниченный язык программирования C

Особенности написания eBPF программ на языке C обусловлены ограниченным набором инструкций и дизайном виртуальной машины. Из них можно отметить следующие:

1. Все eBPF объекты должны быть определены в отдельных ELF секциях в исходном файле.

2. Все функции, вызываемые в программе, должны быть встраиваемыми (inline), кроме eBPF помощников (eBPF helpers) [17] — специальных методов, определённых в ядре для использовния в eBPF программах.

3. Возможно использование только циклов, ограниченных постоянными значениями (bounded loops).

4. Нельзя использовать глобальные переменные, а также строки и массивы с постоянными значениями.

Подробный список особенностей и ограничений при написании eBPF программ на языке C приведён в документации проекта Cilium «BPF and XDP Reference Guide» [18].

3.3.3 Верификатор

Верификатор проверяет каждую eBPF программу, загружаемую в ядро [12]. Первым этапом верификации является построение из инструкций ориентированного ациклического графа для проверки наличия циклов и недостижимых инструкций. При этом также проводится проверка инструкций по контекстно-свободной грамматике. На втором этапе верификатор симулирует запуск программы по всем возможным путям, наблюдая при этом изменения

стека и регистров. Прохождение такой проверки гарантирует, что запуск программы не сможет нарушить работу ядра.

4. Реализация утилиты

В рамках данной работы для отслеживания пути сетевых пакетов с помощью eBPF используются программы двух типов точек крепления: классификаторы подсистемы управления трафиком (eBPF классификаторы) и точки зондирования ядра (eBPF зонды).

Также для обмена информацией между программами и сбора данных о пути пакета используются два eBPF массива:

- `skb_map` — указатель на структуру `sk_buff`, соответствующую отслеживаемому пакету.
- `path_map` — отметки времени, прошедшего с момента запуска системы до попадания пакета в определённые функции ядра.

4.1 Алгоритм работы

Алгоритм работы утилиты представлен в виде диаграммы последовательности на рис. 3:

1. Утилита получает выражение-фильтр, определяющее наблюдаемый сетевой трафик, генерирует на его основе eBPF классификатор и загружает его в ядро вместе с массивами `skb_map` и `path_map`.

2. Компилируются eBPF зонды для всех наблюдаемых точек зондирования (сетевых функций) и загружаются в ядро.

3. Классификатор проверяет все проходящие через него пакеты и при нахождении того, что удовлетворяет фильтру, записывает указатель на соответствующую структуру `sk_buff` в массив `skb_map`.

4. Зонды просматривают указатели структур `sk_buff`, обрабатываемых зондируемыми функциями, и, если находится совпадение с указателем, лежащим в `skb_map`, зонд записывает текущую отметку времени в `path_map`. Также, если текущий зонд помечен как один из потенциально последних в пути обработки пакета, он заполняет значение массива `skb_map` единицами, что служит для утилиты сигналом к завершению отслеживания пути пакета.

5. Утилита периодически проверяет значение в массиве `skb_map`. Когда оно заполняется единицами, утилита собирает информацию из массива `path_map`, сопоставляет значения с именами функций, сортирует по отметкам времени и направляет получившийся путь пакета в стандартное устройство вывода.

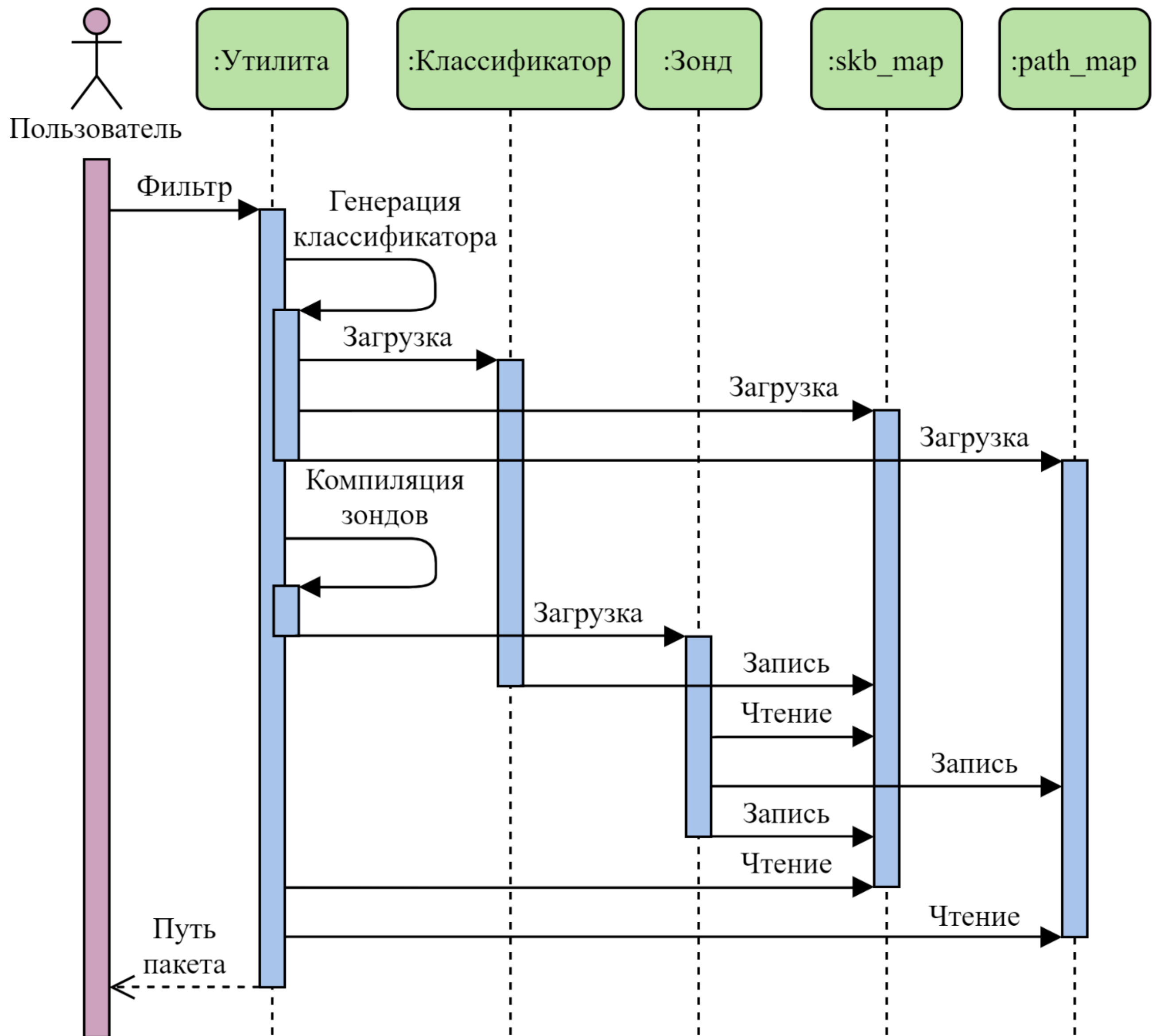

Рис. 3 Алгоритм работы утилиты

4.2 eBPF программы

Связка классификатор+зонды позволяет фильтровать сетевой пакет на самых ранних этапах его обработки сетевым стеком и отслеживать его перемещение по ядру простым сравнением указателей. Такой подход избавил от необходимости проверять пакет на каждой зондируемой функции, что понизило накладные расходы и упростило написание программ.

4.2.1 eBPF классификатор

Задача классификатора — проверка входящих пакетов на соответствие фильтру. Чтобы не проводить лишние операции, проверка производится только когда пакет ещё не найден — значение массива `skb_map` равно нулю. Упрощённый код классификатора представлен в лист. 1.

```
// Определение ELF секции, в которой будет располагаться программа
__section("main")
int skb_filter(struct __sk_buff *skb) {
// Вызов eBPF помощника для получения указателя на значение массива
// skb_map, соответствующее ключу 0
    uint32_t skb_key = 0;
    void   **skb_val = map_lookup_elem(&skb_map, &skb_key);
    if (skb_val  == NULL) return TC_ACT_OK;
    if (*skb_val != 0)    return TC_ACT_OK;
// Определение указателей на данные пакета
    void *data         = (void *)(long)skb->data;
    void *data_end     = (void *)(long)skb->data_end;
    struct ethhdr *eth = data;
    struct iphdr  *iph = data + sizeof(*eth);
// Проверка на выход за границы содержимого пакета
    if (data+sizeof(*eth)+sizeof(*iph) > data_end) return TC_ACT_OK;
// Проверка на IPv4 протокол
    if (eth->h_proto != htons(ETH_P_IP)) return TC_ACT_OK;
// Проверка протокола транспортного уровня
    if (iph->protocol != FILTER_IP_PROTO) return TC_ACT_OK;
// Проверка IP-адреса отправителя
    if (iph->saddr != htonl(FILTER_SRC_IP)) return TC_ACT_OK;
// Запись указателя на пакет в skb_map
    *skb_val = (void *)skb;
```

Листинг 1. Упрощённый код eBPF классификатора

Важная особенность написания eBPF программ — это строгая проверка всех указателей. В программе, представленной выше, это указатели `skb_val`, `data` и `data_end`. Верификатор не может отследить доступ к значениям по указателям, поэтому программы, в которых такие проверки не проводятся, отклоняются при загрузке в ядро [18].

Из-за контекста своей рабочей области, eBPF классификаторы должны сами выполнять необходимые действия с пакетом (например, сброс или перенаправление на другой интерфейс) и возвращать соответствующие значения `TC_ACT` [19], чтобы осведомить об этих действиях ядро. Здесь проводится только фильтрация пакета, над ним не производится никаких дополнительных действий, вследствие чего всегда возвращается значение `TC_ACT_OK`, что сигнализирует ядру о продолжении обработки пакета.

В текущей своей реализации прототип утилиты может отслеживать пути входящих ICMP, TCP и UDP пакетов. Это достигается путём изменения значений макросов `FILTER_IP_PROTO` и `FILTER_SRC_IP` при компиляции. Эти макросы задают протокол транспортного уровня и IP-адрес отправителя соответственно.

4.2.2 eBPF зонды

Действия зондов заключаются в сравнении указателя на обрабатываемый зондируемой функцией сетевой пакет со значением в `skb_map` и записи отметки времени по соответствующему ключу в `path_map` при совпадении. Также, если зонд помечен как последний, он заполняет `skb_map` единицами. Помимо сигнала о завершении наблюдения пути пакета, такое действие позволяет избежать ложные срабатывания в случае обработки системой нового пакета с тем же адресом. Упрощённая версия кода, используемого всеми зондами представлена в лист. 2.

```
__section(KP_SEC)
int skb_check(struct pt_regs *ctx) {
    uint32_t skb_key = 0;
    void   **skb_val = map_lookup_elem(&skb_map, &skb_key);
    if (skb_val == NULL) return 0;
// Первый аргумент зондируемой функции -- указатель на sk_buff
    void *skb = (void *) PT_REGS_PARM1(ctx);
// Сравнение указателей
    if (skb != *skb_val) return 0;
// Запись отметки времени, полученной через eBPF помощник
    uint32_t path_key   = KP_NUM;
    uint64_t path_value = ktime_get_ns();
    r = map_update_elem(&path_map, &path_key, &path_value, BPF_ANY);
    if (r < 0) return 0;
// Вывод информации о прохождении пакетом данной функции
    trace_printk(KP_NAME ": %llu ns\n", path_value);
// Запись единиц в skb_map, если зонд помечен как последний
#if KP_FIN
    *skb_val = SKB_FIN;
#endif
```

Листинг 2. Упрощённый код eBPF зонда.

Для идентификации конкретного зонда используется макросы, значения которых определяется во время компиляции:

- `KP_NUM` хранит номер зондируемой функции, который используется в качестве ключа при записи отметок времени в массив path_map.
- `KP_SEC` содержит строку формата «`kprobe/<имя_зондируемой_функции>`», необходимую для определения названия ELF секции для программы. На основе этой секции загрузчик определяет функцию ядра, к которой необходимо прикрепить зонд.
- `KP_FIN` равен единице, если зонд помечен как последний, или нулю в обратном случае.

Значения для этих макросов берутся из файла `kp_funcs.list`, в котором перечислены все функции, для которых будут созданы зонды. Каждая строка содержит имя функции и цифру 1 или 0, являющуюся флагом, последняя эта

функция или нет. Номер функции определяется как порядковый номер строки в этом файле. Под последней подразумевается функция ядра, которой заканчивается обработка пакета.

Таким образом, из одного файла с описанием eBPF программы и с помощью списка функций `kp_funcs.list` компилируются объектные файлы для всех наблюдаемых точек зондирования.

4.3 eBPF массивы

eBPF массивы это структуры, хранящие данные вида «(ключ, значение)». Они могут выполнять роль хеш-таблицы, хранить в себе файловые дескрипторы для вызова eBPF программ или же просто выступать в роли массива целочисленных значений. eBPF массивы необходимы для сохранения состояния eBPF программ и обмена данными между ними. Если же их прикрепить в файловую систему `bpffs`, появится возможность просматривать и модифицировать их значения из пространства пользователя [20].

Общий способ определения eBPF массива представлен в лист. 3 на примере определения `skb_map`.

```
__section("maps")
struct bpf_elf_map skb_map = {
// Тип -- массив целочисленных значений
    .type       = BPF_MAP_TYPE_ARRAY,
// Размер ключа. Для TYPE_ARRAY это всегда 4
    .size_key   = sizeof(uint32_t),
// Размер значения
    .size_value = sizeof(void *),
// Количество элементов
    .max_elem   = 1,
// Флаг прикрепления в файловую систуму bpffs
    .pinning    = PIN_GLOBAL_NS

};
```

Листинг 3. Определение массива skb_map.

Массив `path_map` определён аналогичным образом. Единственное отличие — количество элементов, которое должно быть не меньше, чем количество зондируемых функций.

4.4 Загрузка и прикрепление eBPF программ

Упомянутая ранее утилита bpftool не имеет возможности прикреплять программы к точкам зондирования, в то время как библиотека libbpf такую функциональность предоставляет. Поэтому, в рамках данной работы, на основе libbpf был создан отдельный загрузчик для зондов. Помимо анализа программ и прикрепления их к соответствующим функциям, он также подменяет определённые в них eBPF массивы на уже существующие в системе, созданные после загрузки eBPF классификатора. В противном случае для каждого зонда создавались бы свои изолированные массивы и программы не могли бы обмениваться данными друг с другом.

Классификаторы также не загружаются в систему с помощью bpftool. Вместо этого используется утилита tc из пакета iproute2 [21]. Для сетевого интерфейса, на котором ожидается наблюдаемый трафик, создаётся дисциплина очередей (queueing discipline) cls_act. Затем объектный файл с eBPF программой в качестве фильтра прикрепляется к этой дисциплине на входящий (ingress) или исходящий (egress) трафик. Утилита tc не использует библиотеку libbpf и реализует собственные способы взаимодействия через системный вызов bpf().

4.5 Разница контекстов точек крепления

Используемые в реализации программы работают в разных контекстах ядра, вследствие чего получают разные аргументы при запуске: eBPF классификаторы — структуру __sk_buff, eBPF зонды — структуру pt_regs.

Структура __sk_buff[6] является зеркалом структуры sk_buff, содержащей всю информацию о сетевом пакете, обрабатываемым ядром [6, с. 483-492]. Обращения к полям __sk_buff преобразуются в обращения к полям sk_buff во время верификации программы. Важно заметить, что, в то время как структура sk_buff определена только внутри ядра и может изменяться в разных версиях, определение зеркала __sk_buff является частью стабильного API. Поэтому использование такого зеркала позволяет без изменения запускать программы на разных версиях ядра Linux, сохраняет изоляцию и не добавляет лишние накладные расходы на создание копии структуры.

[6] https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h

Структура `pt_regs`[7] содержит состояние регистров процессора при вызове функции. Через неё eBPF зонд может получить доступ ко всем аргументам, переданным в зондируемую функцию. Доступ к полям структуры `pt_regs` осуществляется через специальные макросы. При компиляции eBPF зондов в эти макросы подставляются значения, зависящие от архитектуры процессора, на котором запущена система.

Таким образом, только классификаторы имеют доступ к содержимому сетевых пакетов, вследствие чего и используются для фильтрации. Зонды же получают указатель на обрабатываемый пакет из структуры `pt_regs` и сравнивают с указателем на отслеживаемый пакет.

4.6 Особенности реализации

Файл со списком зондируемых функций `kp_funcs.list` не только помогает при компиляции зондов, но и обеспечивает независимость утилиты от конкретной сборки ядра. Имена функций в разных версиях Linux меняются и этот файл позволяет проще адаптироваться к этим изменениям. Также пользователь при необходимости может добавить в `kp_funcs.list` свои функции, где он хочет пронаблюдать прохождение сетевого пакета.

На системах с ядром Linux в конфигурационном файле `/etc/security/limits.conf` определены пределы для различных системных значений, таких как максимальные размер файла, размер стека или количество процессов [22]. Для утилиты важным значением является `memlock` — максимальный размер фиксированного в памяти адресного пространства. Этим пределом ограничивается количество страниц в оперативной памяти, которые операционная система не будет складывать в файл подкачки. Это необходимо для загрузки большого количества eBPF зондов, поэтому этот предел необходимо повысить на тестируемой системе для корректной работы утилиты.

4.7 Демонстрация работы прототипа

Взаимодействие с прототипом утилиты происходит через сценарий `bpfpath.sh`, написанный на языке Bash Script. Первым аргументом сценария является протокол транспортного уровня (`icmp`, `tcp` или `udp`), вторым — IP адрес отправителя пакета. После вызова сценария идёт компиляции и загрузка в

[7] https://github.com/torvalds/linux/blob/master/arch/x86/include/uapi/asm/ptrace.h

ядро всех eBPF программ. При прохождении пакета, удовлетворяющего фильтру, его путь будет отображён в файле `/sys/kernel/debug/tracing/trace_pipe`.

Работа прототипа демонстрируется на примере отслеживания путей ICMP ответов на узле A системы, конфигурация которой была описана в обзоре. В лист. 4 представлен путь ICMP ответа при наличии связи. В лист. 5 — при её отсутствии.

```
[004] ..s1  8562.107530: 0: __netif_receive_skb_core: 8562028538131 ns
[004] ..s4  8562.107551: 0: macvlan_forward_source:   8562028561143 ns
[004] ..s4  8562.107555: 0: ip_rcv_core:              8562028565136 ns
[004] ..s4  8562.107557: 0: nf_hook_slow:             8562028566873 ns
[004] ..s4  8562.107559: 0: nf_ip_checksum:           8562028568891 ns
[004] ..s4  8562.107560: 0: nf_ct_get_tuple:          8562028569857 ns
[004] ..s4  8562.107562: 0: ip_route_input_noref:     8562028572165 ns
[004] ..s4  8562.107563: 0: ip_route_input_slow:      8562028573256 ns
[004] ..s4  8562.107567: 0: fib_validate_source:      8562028576612 ns
[004] ..s4  8562.107571: 0: ip_local_deliver:         8562028581026 ns
[004] ..s4  8562.107572: 0: nf_hook_slow:             8562028581720 ns
[004] ..s4  8562.107573: 0: ipt_do_table:             8562028582731 ns
[004] ..s4  8562.107575: 0: nf_confirm:               8562028584775 ns
[004] ..s4  8562.107576: 0: raw_local_deliver:        8562028586250 ns
[004] ..s4  8562.107577: 0: icmp_rcv:                 8562028587511 ns
[004] ..s4  8562.107578: 0: ping_rcv:                 8562028588550 ns
[004] .Ns4  8562.107585: 0: consume_skb:              8562028594927 ns
```

Листинг 4. Путь ICMP пакета при наличии связи

```
[004] ..s1  8597.573506: 0: __netif_receive_skb_core: 8597494666740 ns
[004] ..s4  8597.573525: 0: macvlan_forward_source:   8597494689310 ns
[004] ..s4  8597.573530: 0: ip_rcv_core:              8597494694221 ns
[004] ..s4  8597.573531: 0: kfree_skb:                8597494696007 ns
```

Листинг 5. Путь ICMP пакета при отсутствии связи

Наличие пути пакета во втором случае показывает, что пакеты в систему всё-таки приходят, но сбрасываются на этапе обработки IP заголовка. Такое поведение является следствием того, что пакет обрабатывается не тем интерфейсом, которому предназначался. Из этого можно сделать вывод, что происходит изменение MAC-адреса получателя в ICMP ответах.

Заключение

В рамках данной работы был реализован прототип утилиты, способной отслеживать пути сетевых пакетов в ядре Linux.

Были достигнуты следующие результаты:

- Сформулированы требования к реализации утилиты на основе анализа существующих инструментов для сетевой отладки. Соответствие этим требованиям позволит использовать утилиту в реальных условиях на промышленных системах.
- Технология eBPF выбрана в качестве наиболее подходящей в рамках выставленных к утилите требований. Сделан обзор технологии и предоставлен механизм её применения для отслеживания путей сетевых пакетов.
- Разработан алгоритм работы утилиты и реализован прототип, способный отследить пути входящих ICMP, TCP и UDP пакетов. Описаны особенности и ограничения прототипа, проведена демонстрация работы.

Исходные коды прототипа размещены в GitHub репозитории[8].

Как было показано на примере, информация о пути сетевого пакета, которую предоставляет прототип, позволяет быстрее решать неочевидные проблемы. Для дальнейшего развития прототипа до полноценной утилиты сетевой отладки были выделены следующие направления:

- Расширить набор параметров для задания фильтров сетевого трафика: поддержка различных протоколов канального, сетевого и транспортного уровней и определения отслеживаемых значений в заголовках этих протоколов.
- Внедрить дополнительные виды вывода пути пакета: графы вызова функций в виде текста и графической отрисовки.
- Добавить отслеживание дупликации пакета в ядре при обработке несколькими интерфейсами одновременно.

[8] https://github.com/restonich/bpfpath

Список литературы

[1] Paul Cobbaut. Linux Networking. — 2015 — URL: http://linux-training.be/linuxnet.pdf (дата обращения: 4.06.2020)

[2] Anthony Critelli. A beginner's guide to network troubleshooting in Linux // Red Hat Enable Sysadmin. — 2019 — URL: https://www.redhat.com/sysadmin/beginners-guide-network-troubleshooting-linux (дата обращения: 4.06.2020)

[3] Shashank Nandishwar Hegde. Basic network troubleshooting in Linux with nmap // Red Hat Enable Sysadmin. — 2020 — URL: https://www.redhat.com/sysadmin/nmap-troubleshooting (дата обращения: 4.06.2020)

[4] Brendan Gregg. Documentation: Network Monitoring // Solaris operating system. — 2005 — URL: http://www.brendangregg.com/Solaris/network.html (дата обращения: 4.06.2020)

[5] Sean Wilkins. The Top 10 Basic Network Troubleshooting Tools Every IT Pro Should Know // Pluralsight blog. — 2011 — URL: https://www.pluralsight.com/blog/it-ops/network-troubleshooting-tools (дата обращения: 4.06.2020)

[6] Rami Rosen. Linux Kernel Networking Implementation and Theory // Apress. — ISBN 978-1-4302-6196-4 — 2014 — 648 с.

[7] Jim Keniston, Prasanna S Panchamukhi, Masami Hiramatsu. Kernel Probes (Kprobes) // Linux kernel documentation. — URL: https://www.kernel.org/doc/Documentation/kprobes.txt (дата обращения: 4.06.2020)

[8] Mathieu Desnoyers. Using the Linux Kernel Tracepoints // Linux kernel documentation. — URL: https://www.kernel.org/doc/Documentation/trace/tracepoints.txt (дата обращения: 4.06.2020)

[9] Steven Rostedt. ftrace — Function Tracer // Linux kernel documentation. — URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt (дата обращения: 4.06.2020)

[10] Matt Fleming. A thorough introduction to eBPF // LWN. — 2017 — URL: https://lwn.net/Articles/740157/ (дата обращения: 4.06.2020)

[11] Steven McCanne, Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture // Lawrence Berkeley Laboratory. — 1992 — URL: https://www.tcpdump.org/papers/bpf-usenix93.pdf (дата обращения: 4.06.2020)

[12] Jay Schulist, Daniel Borkmann, Alexei Starovoitov. Linux Socket Filtering aka Berkeley Packet Filter (BPF) // Linux kernel documentation. — URL: https://www.kernel.org/doc/Documentation/networking/filter.txt (дата обращения: 4.06.2020)

[13] Brendan Gregg. BPF Performance Tools // Addison-Wesley Professional Computing Series. — ISBN-13 978-0136554820 — 2019 — 880 с.

[14] bpf — perform a command on an extended BPF map or program // Linux Programmer's Manual. — 2019 — URL: https://man7.org/linux/man-pages/man2/bpf.2.html (дата обращения: 4.06.2020)

[15] Automated upstream mirror for libbpf stand-alone build // GitHub репозиторий. — URL: https://github.com/libbpf/libbpf (дата обращения: 4.06.2020)

[16] Директория исходных кодов утилиты bpftool // GitHub репозиторий. — URL: https://github.com/torvalds/linux/tree/master/tools/bpf/bpftool (дата обращения: 4.06.2020)

[17] BPF-HELPERS — list of eBPF helper functions // Linux Programmer's Manual. — 2019 — URL: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html (дата обращения: 4.06.2020)

[18] BPF and XDP Reference Guide // Cilium's documentation. — URL: https://docs.cilium.io/en/latest/bpf/ (дата обращения: 4.06.2020)

[19] Daniel Borkmann. BPF — BPF programmable classifier and actions for ingress/egress queueing disciplines // Linux manpage. — 2015 — URL: https://man7.org/linux/man-pages/man8/tc-bpf.8.html (дата обращения: 4.06.2020)

[20] Jonathan Corbet. Persistent BPF objects // LWN. — 2015 — URL: https://lwn.net/Articles/664688/ (дата обращения: 4.06.2020)

[21] Bert Hubert. tc — show / manipulate traffic control settings // Linux manpage. — 2001 — URL: https://man7.org/linux/man-pages/man8/tc.8.html (дата обращения: 4.06.2020)

[22] Cristian Gafton. limits.conf — configuration file for the pam_limits module // Linux-PAM Manual. — 2016 — URL: https://man7.org/linux/man-pages/man5/limits.conf.5.html (дата обращения: 4.06.2020)