# Часть 1 : Что такое eBPF и почему это важно?

eBPF — это революционная технология ядра, которая позволяет разработчикам писать собственный код, который можно динамически загружать в ядро, изменяя поведение ядра. (Не волнуйтесь, если вы не уверены в том что такое ядро — мы скоро к этому вернемся в этой главе.)

Это обеспечивает новое поколение высокопроизводительных сетевых инструментов, средств наблюдения и безопасности. И, как вы увидите, если вы хотите оснастить своё приложение этими инструментами на основе eBPF, вам не нужно каким-либо образом модифицировать или перенастраивать код самого приложения, благодаря удобному расположению самого eBPF внутри ядра Linux.

Вот лишь несколько вещей, которые вы можете делать с помощью eBPF:

- Отслеживание производительности практически любого аспекта системы.
- Высокопроизводительная сеть с прозрачной видимостью.
- Обнаружение и (необязательно) предотвращение злонамеренной активности.

Давайте совершим небольшое путешествие по истории eBPF, начав с пакетного фильтра Беркли.

## Корни eBPF: пакетный фильтр Беркли

То что мы сегодня называем «eBPF» восходит к пакетному фильтру BSD, впервые описанному в 1993 году в статье[1], написанной Стивеном Макканном и Ван Якобсоном из Национальной лаборатории Лоуренса в Беркли. В этой статье рассматривается псевдомашина, которая может запускать фильтры — программы, написанные для определения того, принимать или отклонять сетевой пакет. Эти программы были написаны с помощью набора инструкций BPF, набора 32-разрядных инструкций общего назначения, который очень напоминает язык ассемблера. Вот пример, взятый непосредственно из этой статьи:

```
ldh    [12]
jeq    #ETHERTYPE IP, L1, L2
L1:    ret    #TRUE
L2:    ret    #0
```

Этот крошечный фрагмент кода отфильтровывает пакеты, которые не являются пакетами интернет-протокола (IP). Входными данными для этого фильтра является пакет Ethernet, и первая инструкция (ldh) загружает 2-байтовое значение, начиная с 12-го байта в этом пакете. В следующей инструкции (jeq) это значение сравнивается со значением, представляющим IP-пакет. Если он совпадает, выполнение переходит к инструкции с меткой L1, и пакет принимается, возвращая ненулевое значение (обозначенное здесь как #TRUE). Если он не совпадает, пакет не является IP-пакетом и отклоняется, возвращая 0.

Вы можете представить себе (или даже обратиться к статье чтобы найти примеры) более сложные фильтрующие программы, которые принимают решения на основе других аспектов

---

1«Фильтр пакетов BSD: новая архитектура для захвата пакетов на уровне пользователя», Стивен Макканн и Ван. Джейкобсон.

пакета. Важно отметить, что автор фильтра может писать свои собственные программы, которые будут выполняться в ядре, и в этом суть того что позволяет eBPF. BPF стал обозначать «Berkeley Packet Filter», и впервые он был представлен в Linux в 1997 году, в версии ядра 2.1.75[2], где он использовался в утилите `tcpdump` как эффективный способ захвата пакетов, подлежащих трассировке.

Перенесемся в 2012 год, когда `seccomp-bpf` был представлен в версии 3.5 ядра. Это позволило использовать программы BPF для принятия решений о том, разрешать или запрещать приложениям пользовательского пространства выполнять системные вызовы. Мы рассмотрим это более подробно в главе 10. Это был первый шаг в эволюции BPF от узкой области фильтрации пакетов к платформе общего назначения, которой она является сегодня. С этого момента слова «пакетный фильтр» в названии стали иметь меньше смысла!

## От BPF к eBPF

BPF превратился в то что мы называем «расширенным BPF» или «eBPF», начиная с версии ядра 3.18 в 2014 году. Это включало несколько значительных изменений:

- Набор инструкций BPF был полностью переработан чтобы быть более эффективным на 64-битных машинах, а интерпретатор был полностью переписан.
- Были введены карты eBPF, которые представляют собой структуры данных, к которым могут обращаться программы BPF и приложения пользовательского пространства, что позволяет обмениваться информацией между ними. Вы узнаете о картах в главе 2.
- Был добавлен системный вызов `bpf()` чтобы программы пользовательского пространства могли взаимодействовать с программами eBPF в ядре. Вы прочтете об этом системном вызове в главе 4.
- Добавлено несколько функций помощников BPF. Вы увидите несколько примеров в главе 2, а некоторые подробности — в главе 6.
- Был добавлен верификатор eBPF для обеспечения безопасности запуска программ eBPF. Это обсуждается в главе 6.

Это заложило основу для eBPF, но развитие не замедлилось! С тех пор eBPF значительно эволюционировал.

## Эволюция eBPF в продакшен системах

С 2005 года в ядре Linux существует функция, называемая `kprobes()` (пробы ядра), позволяющая устанавливать ловушки практически для любой инструкции в коде ядра. Разработчики могли писать модули ядра, которые прикрепляли функции к `kprobes` для целей отладки или измерения производительности[3]. Возможность присоединения программ eBPF к `kprobes` была добавлена в 2015 году, и это стало отправной точкой для революции в том, как выполняется трассировка в системах Linux. В то же время в сетевой стек ядра начали добавляться различные хуки, позволяющие программам eBPF заботиться о большем количестве аспектов сетевой функциональности. Подробнее об этом мы поговорим в главе 8. К 2016 году инструменты на основе eBPF использовались в производственных системах.

2 Эти и другие подробности взяты из презентации Алексея Старовойтова NetDev в 2015 году «BPF — виртуальный машина».

3 В документации ядра есть хорошее описание работы `kprobes`.

Работа Брендана Грегга по отслеживанию в Netflix стала широко известна в инфраструктурных и операционных кругах, как и его заявление о том что eBPF «придает Linux сверхспособности». В том же году было объявлено о проекте Cilium, который стал первым сетевым проектом, использующим eBPF для замены всего пути данных в контейнерных средах.

В следующем году Facebook (теперь Meta) сделал Katran проектом с открытым исходным кодом. Katran, балансировщик нагрузки уровня 4, удовлетворил потребности Facebook в масштабируемом и быстром решении. Каждый отдельный пакет на Facebook.com с 2017 года проходил через eBPF/XDP4[4]. Лично для меня этот год вызвал у меня воодушевление по поводу возможностей, предоставляемых этой технологией, после того, как я увидела доклад Томаса Графа о eBPF и проект Cilium на DockerCon в Остине, штат Техас.

В 2018 году eBPF стала отдельной подсистемой в ядре Linux. Боркманн из Isovalent и Алексей Старовойтов из Meta были в качестве его сопровождающих (ещё позже к ним присоединился Андрей Накрыйко, тоже из Меты). В том же году состоялось введение созданного формата типа BPF (BTF), что делает программы eBPF намного более информативными и портативными. Мы рассмотрим это в главе 5.

В 2020 году был представлен LSM BPF, позволяющий использовать программы eBPF. подключенные к интерфейсу ядра Linux Security Module (LSM). Это указывало на то, что был определен третий основной вариант использования eBPF: стало ясно что eBPF — отличная платформа и для инструментов безопасности, в дополнение к сети и наблюдению.

За прошедшие годы возможности eBPF существенно выросли благодаря работе более 300 разработчиков ядра и многих разработчиков связных инструментов пользовательского пространства (таких как bpftool, с которым мы познакомимся в главе 3), компиляторов и библиотек языков программирования.

Когда-то программы были ограничены лимитом в 4096 инструкции, но этот лимит вырос до 1 миллиона верифицированных инструкций инструкций[5] и лимит фактически стал неактуальным благодаря поддержке хвостовых вызовов и функциональных вызовов (которые вы увидите в главах 2 и 3).

> Для более глубокого ознакомления с историей eBPF, к кому лучше обратиться чем его сопровождающие, которые работали над ним с самого начала?
>
> Алексей Старовойтов выступил с увлекательной презентацией об истории BPF (https://www.youtube.com/watch?v=DAvZH13725I) которая уходит своими корнями в программно определяемые сети (SDN). В этом выступлении он обсуждает стратегии, используемые для получения раннего eBPF, патчи, принятые в ядро, и показывает что официальный день рождения eBPF — 26 сентября 2014 г., что ознаменовало принятие первого набора исправлений, охватывающих верификатор, системный вызов BPF и карты.
>
> Даниэль Боркманн также обсудил историю BPF и ее эволюция для поддержки сетей и функций отслеживания. Я очень рекомендую его выступление «eBPF и Kubernetes: маленькие помощники-миньоны для масштабирования

4 Этот примечательный факт взят из выступления Даниэля Боркманна (Daniel Borkmann) на KubeCon 2020, https://www.youtube.com/watch?v=99jUcLt3rSk — под названием «eBPF and Kubernetes: Little Helper Minions for Scaling Microservices».

5 Для получения более подробной информации о лимите инструкций и «лимите сложности» см. https://ebpf.io/blog/ebpf-updates-2021-02/#did-you-know-program-size-limit

микросервисов» (https://www.youtube.com/watch?v=DAvZH13725I), который полон интересных самородков информации.

## Сложности именования

Приложения eBPF выходят далеко за рамки фильтрации пакетов, поэтому аббревиатура теперь практически бессмысленна и стала самостоятельным термином. А поскольку ядра Linux, широко используемые в наши дни, поддерживают «расширенные» части, термины eBPF и BPF в основном используются как синонимы. В исходном коде ядра и при программировании eBPF общепринятой терминологией является BPF. Например, как мы увидим в главе 4, системным вызовом для взаимодействия с eBPF является `bpf()`, имена функций-помощников начинаются с `bpf_`, а различные типы программ (e)BPF обозначаются именами, начинающимися с `BPF_PROG_TYPE`. За пределами сообщества ядра Linux название «eBPF», похоже, прижилось, например, на сайте сообщества ebpf.io (https://ebpf.io/) и в названии eBPF Foundation (https://ebpf.foundation/).

## Ядро Linux

Чтобы понять eBPF, вам нужно четко понимать разницу между ядром и пользовательским пространством в Linux. Я рассказала об этом в своем отчете «Что такое eBPF?»[6] и адаптировала часть его содержания для следующих нескольких абзацев.

Ядро Linux — это программный слой между вашими приложениями и оборудованием, на котором они работают. Приложения работают на непривилегированном уровне, называемом пользовательским пространством, которое не может напрямую обращаться к оборудованию. Вместо этого приложение отправляет запросы, используя интерфейс системного вызова (syscall), чтобы попросить ядро действовать от его имени. Этот аппаратный доступ может включать чтение и запись файлов, отправку или получение сетевого трафика, или даже просто доступ к памяти. Ядро также отвечает за координацию параллельных процессов, что позволяет запускать множество приложений одновременно. Это показано на рис. 1-1.

Как разработчики приложений, мы обычно не используем интерфейс системных вызовов напрямую, потому что языки программирования предоставляют нам высокоуровневые абстракции и стандартные библиотеки, которые являются более простыми интерфейсами для программирования. В результате многие люди пребывают в блаженном неведении о том, как много делает ядро во время работы наших программ. Если вы хотите понять насколько часто вызывается ядро, вы можете использовать утилиту `strace` чтобы показать вам все системные вызовы, которые делает приложение.

6 Выдержка из «Что такое eBPF?» Лиз Райс. Авторское право © 2022 O’Reilly Media. Используется с разрешения.

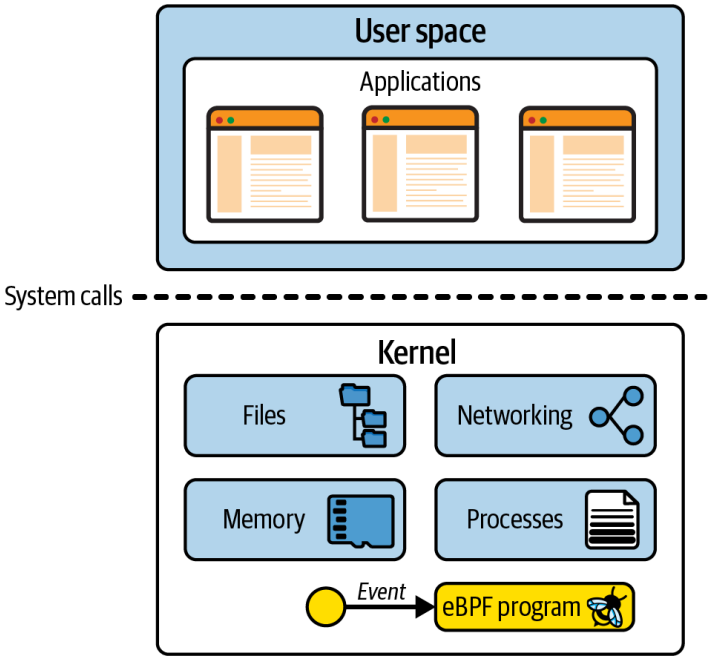


*Рисунок 1-1. Приложения в пользовательском пространстве используют интерфейс системного вызова для выполнения запросов к ядру*

Вот пример, когда использование простейшей утилиты `cat` для вывода на экран слова hello включает более 100 системных вызовов:

```
$ strace -c echo "hello"
hello
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
24.62    0.001693          56        30        12 openat
17.49    0.001203          60        20           mmap
15.92    0.001095          57        19           newfstatat
15.66    0.001077          53        20           close
10.35    0.000712         712         1           execve
3.04     0.000209          52         4           protect
2.52     0.000173          57         3           read
2.33     0.000160          53         3           brk
2.09     0.000144          48         3           munmap
1.11     0.000076          76         1           write
0.96     0.000066          66         1         1 faccessat
0.76     0.000052          52         1           getrandom
0.68     0.000047          47         1           rseq
0.65     0.000045          45         1           set_robust_list
0.63     0.000043          43         1           prlimit64
0.61     0.000042          42         1           set_tid_address
0.58     0.000040          40         1           futex
------ ----------- ----------- --------- --------- ----------------
100.00   0.006877          61       111        13 total
```

Поскольку приложения так сильно зависят от ядра, это означает что мы можем многое узнать о том, как ведет себя приложение, если сможем наблюдать за его взаимодействием с ядром. С помощью eBPF мы можем добавить инструменты в ядро чтобы получать такую информацию.

Например, если вы сможете перехватить системный вызов для открытия файлов, вы сможете точно увидеть к каким файлам обращается какое-либо приложение. Но как бы вы могли сделать этот перехват? Давайте рассмотрим что было бы задействовано, если бы мы захотели изменить поведение ядра, добавив новый код для создания какого-то вывода всякий раз, когда вызывается этот системный вызов.

## Добавление новой функциональности в ядро

Добавление новой функциональности в ядро Ядро Linux весьма сложное, на момент написания этой статьи в ядре содержится около 30 миллионов строк кода[7]. Для внесения изменений в любую кодовую базу требуется некоторое знакомство с существующим кодом, поэтому, если вы ещё не являетесь непосредственно разработчиком ядра, это, скорее всего, вызовет трудности.

Кроме того, если вы хотите внести свои изменения в развитие кодовой базы, вы столкнетесь с проблемой, которая не является чисто технической. Linux — это операционная система общего назначения, используемая во всех средах и обстоятельствах. Это означает, что если вы хотите чтобы ваше изменение стало частью официального выпуска Linux, это не просто вопрос написания кода который работает. Код этот должен быть принят сообществом (и, в частности, Линусом Торвальдсом, создателем и главным разработчиком Linux) как изменение, которое пойдет на всеобщее благо. Это как данность — принимается только одна треть представленных исправлений ядра[8].

Предположим, вы придумали хороший технический подход к перехвату системного вызова для открытия файлов. После нескольких месяцев обсуждений и тяжелой работы по разработке с вашей стороны давайте представим что это изменение принято в ядро. Замечательно! Но сколько времени пройдет, пока это изменение появится на всех машинах?

Каждые два или три месяца выпускается новый выпуск ядра Linux, но даже когда изменение попадает в один из этих выпусков, оно еще не скоро станет доступным в производственных средах большинства людей. Это связано с тем, что большинство из нас не просто использует ядро Linux напрямую — мы используем дистрибутивы Linux, такие как Debian, Red Hat, Alpine или Ubuntu, которые комплектуют версию ядра Linux с различными другими компонентами. Вы вполне можете обнаружить что ваш любимый дистрибутив использует версию ядра которой уже несколько лет.

Например, многие корпоративные пользователи используют Red Hat Enterprise Linux (RHEL). На момент написания этой статьи текущим выпуском является RHEL 8.5 от ноября 2021 года, в котором используется ядро Linux версии 4.18. Это ядро было выпущено в августе 2018 года.

Как показано на карикатуре на рис. 1.2, требуются буквально годы чтобы превратить новую функциональность из стадии идеи в производственную среду ядра Linux[9].

---

7 «Linux 5.12 приближается к 28,8 миллионам строк». Фороникс, март 2021 г.

8 Цзян И., Адамс Б., German DM. 2013. «Выживет ли мой патч? И как быстро?» (2013). Согласно этому исследованию, 33% исправлений принимаются, и большинство из них занимает от трех до шести месяцев.

9 К счастью, исправления безопасности для уже существующих функций становятся доступными быстрее.


*Рисунок 1-2. Добавление фич в ядро (мультик Вадима Щеколдина, Изовалент)*

## Модули ядра

Если вы не хотите годами ждать, пока ваши изменения попадут в ядро, есть еще один вариант. Ядро Linux было разработано для подключения модулей ядра, которые можно загружать и выгружать по требованию. Если вы хотите изменить или расширить поведение ядра, написание модуля, безусловно, является одним из способов сделать это. Модуль ядра может распространяться для других применений независимо от официального выпуска ядра Linux, поэтому его не нужно формально принимать в основную базу исходного кода.

Самая большая проблема здесь заключается в том, что это всё еще полноценное программирование ядра. Исторически пользователи очень осторожно относились к использованию модулей ядра по одной простой причине: если код ядра дает сбой, он немедленно останавливает машину и всё что на ней работает. Как пользователь может быть уверен что модуль ядра безопасен для запуска?

Быть «безопасным для запуска» означает не только отсутствие критических сбоев — пользователь хочет знать что модуль ядра безопасен с точки зрения своей защищённости. Включает ли он уязвимости, которыми может воспользоваться злоумышленник? Верим ли мы что авторы модуля не поместят в него вредоносный код? Поскольку ядро представляет собой привилегированный код, оно имеет доступ ко всему на машине, включая все данные, поэтому вредоносный код в ядре может стать серьезной причиной для беспокойства. Это относится в равной мере и к модулям ядра.

Безопасность ядра — одна из важных причин, почему дистрибутивы Linux так долго включают новые выпуски. Если другие люди использовали версию ядра в различных обстоятельствах в течение нескольких месяцев или лет, то это должно было обнаружить и устранить проблемы. Специалисты по сопровождению дистрибутива могут быть уверены что ядро, которое они отправляют своим пользователям/покупателям, *защищено*, то есть его можно безопасно запускать.

eBPF предлагает совсем другой подход к безопасности: верификатор eBPF, который гарантирует, что программа eBPF загружается только в том случае, если ее выполнение

безопасно — это не приведет к сбою машины, не заблокирует ее в жестком состоянии и не позволит произойти компрометации данных. Мы обсудим процесс верификации более подробно в главе 6.

## Динамическая загрузка программ eBPF

Программы eBPF можно динамически загружать в ядро и удалять из него. Как только они присоединяются к событию, они будут активизироваться этим событием независимо от того что вызвало само это событие. Например, если вы присоедините к системному вызову программу для открытия файлов, она будет запускаться всякий раз, когда какой-либо процесс попытается открыть файл. Неважно, был ли этот процесс уже запущен на тот момент когда программа была загружена. Это огромное преимущество по сравнению с обновлением ядра и последующей перезагрузкой машины чтобы использовать эти новые функции.

Это приводит к одной из самых сильных сторон инструментов наблюдения или безопасности, использующих eBPF, — он мгновенно получает представление обо всём что происходит на машине. В средах, где работают контейнеры, это включает в себя видимость всех процессов, запущенных внутри этих контейнеров, а также на хост-компьютере. Позже в этой главе я расскажу о последствиях этого для облачных развертываний.

Кроме того, как это показано на рис. 1-3, люди могут очень быстро создавать новые функции ядра с помощью eBPF, не требуя чтобы все остальные пользователи Linux принимали те же изменения.

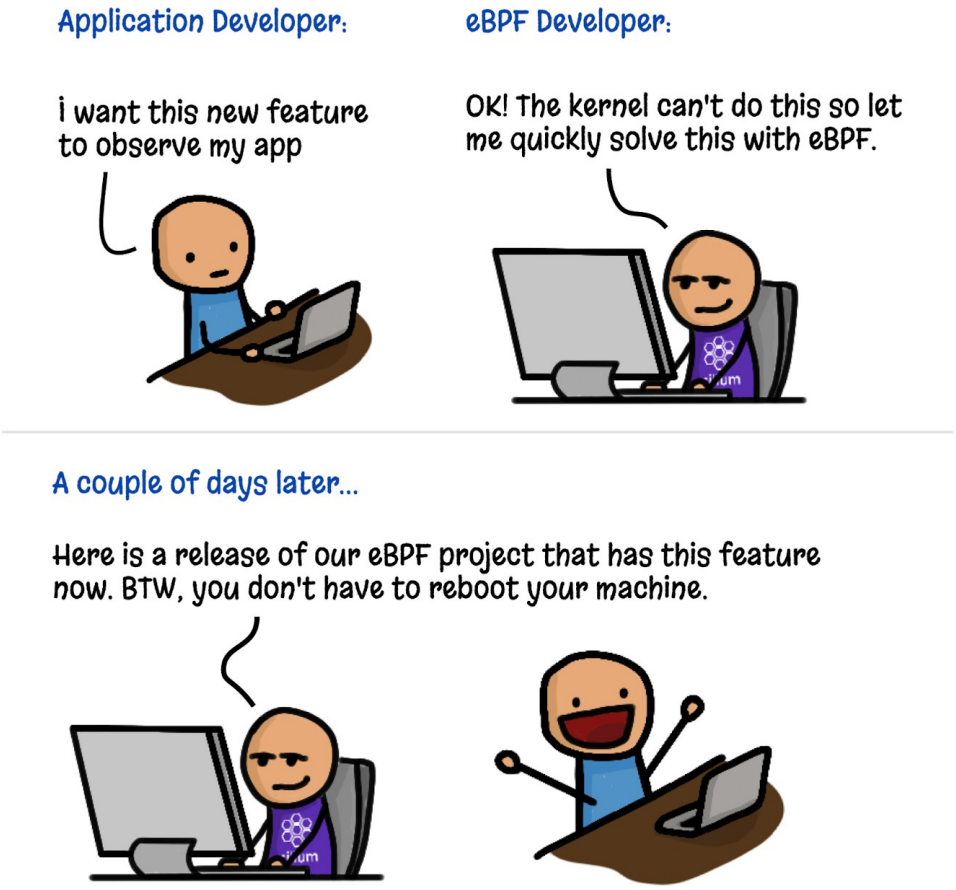


*Рисунок 1-3. Добавление функций ядра с помощью eBPF (мультик Вадима Щеколдина, Isovalent)*

## Высокая производительность программ eBPF

Программы eBPF — очень эффективный способ добавления инструментария. После загрузки и JIT-компиляции (что вы увидите в главе 3) программа выполняется как нативные машинные инструкции на этого ЦП. Кроме того, нет необходимости нести затраты на переход между ядром и пользовательским пространством (что является дорогостоящей операцией) для обработки каждого события.

Документ 2018 года[10], в котором описывается путь к данным eXpress (XDP), содержит некоторые иллюстрации того, какие улучшения производительности обеспечивает eBPF в сети. Например, реализация маршрутизации в XDP «улучшает производительность в 2,5 раза» по сравнению с обычной реализацией ядра Linux, а «XDP предлагает прирост производительности в 4,3 раза по сравнению с IPVS» для балансировки нагрузки.

Еще одним преимуществом eBPF для отслеживания производительности и наблюдения за безопасностью является то, что соответствующие события могут быть отфильтрованы в ядре, прежде чем нести затраты на их отправку в пространство пользователя. Целью первоначальной реализации BPF, прежде всего, была фильтрация только определенных сетевых пакетов. Но сегодня программы eBPF могут собирать информацию обо всех типах событий в системе, и они могут использовать сложные, настраиваемые программные фильтры для отправки в пространство пользователя только соответствующего подмножества информации.

## eBPF в облачных средах

В наши дни многие организации предпочитают не запускать какие-либо приложения, выполняя программы непосредственно на своих серверах. Вместо этого многие используют собственные облачные подходы: контейнеры, оркестраторы, такие как Kubernetes или ECS, или бессерверные подходы, такие как Lambda, облачные функции, Fargate и т. д. Все эти подходы используют автоматизацию для выбора сервера, на котором будет выполняться каждая рабочая нагрузка; в безсерверных мы даже не знаем, какой сервер выполняется каждая рабочая нагрузка.

Тем не менее, если есть задействованные серверы, то каждый из этих серверов (будь то виртуальная машина или машина с «голым железом») запускает ядро. Когда приложения запускаются в контейнере, то, если они работают на одной (виртуальной) машине, они используют одно и то же ядро. В среде Kubernetes это означает что все контейнеры во всех модулях на данном узле используют одно и то же ядро. Когда мы оснащаем это ядро программами eBPF, все контейнерные рабочие нагрузки на этом узле становятся видимыми для этих программ eBPF, как показано на рис. 1-4.

10 Høiland-Jørgensen T, Brouer JD, Borkmann D, et al. «Путь данных eXpress: быстрая программируемая обработка пакетов в ядре операционной системы». Материалы 14-й Международной конференции по новым сетевым экспериментам и технологиям (CoNEXT ’18). Ассоциация вычислительной техники; 2018: 54–66.

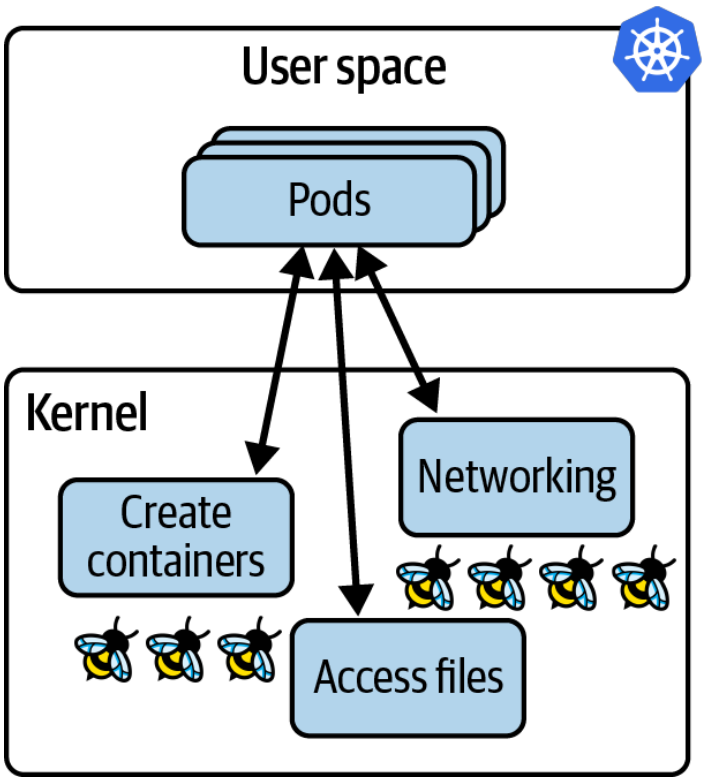


*Рисунок 1-4. Программы eBPF в ядре имеют видимость всех приложений, работающих на узле Kubernetes.*

Видимость всех процессов на узле, в сочетании с возможностью динамической загрузки программ eBPF, дает нам настоящие сверхспособности инструментов на основе eBPF в облачных вычислениях:

- Нам не нужно менять наши приложения, или даже способ их настройки, чтобы инструментировать их с помощью инструментов eBPF.
- После загрузки в ядро и привязки к событию программа eBPF может начать наблюдать за уже существующими процессами приложений.

Сравните это с моделью Sidecar[11], которая использовалась для добавления в приложения Kubernetes таких функций как ведение журналов, трассировка, безопасность и сервисная сетка. В подходе Sidecar инструментарий работает как контейнер, который «внедряется» в каждый под[12] (Pod) приложений. Этот процесс включает в себя изменение файлов конфигураций YAML, определяющеих модули приложения, добавление определения контейнера Sidecar. Этот подход, безусловно, более удобен, чем добавление инструментария в исходный код приложения (что нам приходилось делать до подхода Sidecar; например, включение библиотеки ведения журнала в наше приложение и выполнение вызовов из этой библиотеки в соответствующие позиции в коде). Тем не менее, и подход с Sidecar имеет несколько недостатков:

- Для добавления сопровождающего модуля необходимо перезапустить модуль приложения.

11 Sidecar-контейнер — это контейнер, который должен быть запущен рядом с основным контейнером внутри пода Kubernetes. Этот паттерн нужен для расширения и улучшения функциональности основного приложения без внесения в него изменений. (Прим. пер.)

12 Под (Pod) – это базовое понятие при проектировании приложений в Kubernetes. Kubernetes оперирует подами, а не контейнерами, при этом поды включают в себя контейнеры. Под может содержать в себе описания одного или нескольких контейнеров, монтируемых разделов, IP-адресов и настроек того, как контейнеры должны работать внутри пода. (Прим. пер.)

- Что-то должно модифицировать конфигурацию приложения YAML. Как правило, это автоматизированный процесс, но если что-то пойдет не так, Sidecar не будет добавлен, а значит, Pod не будет инструментирован. Например, развертывание может быть аннотировано, чтобы указать что контроллер допуска должен добавить сопроводительный YAML в спецификацию модуля для этого развертывания. Но если развертывание помечено неправильно, Sidecar не будет добавлен, и поэтому он не виден инструментарию.

- Если в поде имеется несколько контейнеров, то они могут быть в состоянии готовности в разное время, и этот их порядок может быть непредсказуемым. Время запуска всего пода может быть значительно замедлено введением Sidecar или, что еще хуже, это может вызвать возникновение гонок или другие нестабильности. Например, документация Open Service Mesh описывает, как контейнеры приложений должны быть нечувствительными ко всему отбрасываемому трафику до тех пор, пока не будет готов прокси-контейнер Envoy.

- Если сетевые функции, такие как сервис Mesh-сети, реализованы как побочный продукт, то это обязательно означает что весь трафик в контейнер приложения и из него должен проходить через сетевой стек в ядре чтобы достичь контейнера сетевого прокси, добавляя задержку к этому трафику; это показано на рис. 1-5. Мы поговорим о повышении эффективности сети с помощью eBPF в главе 9.

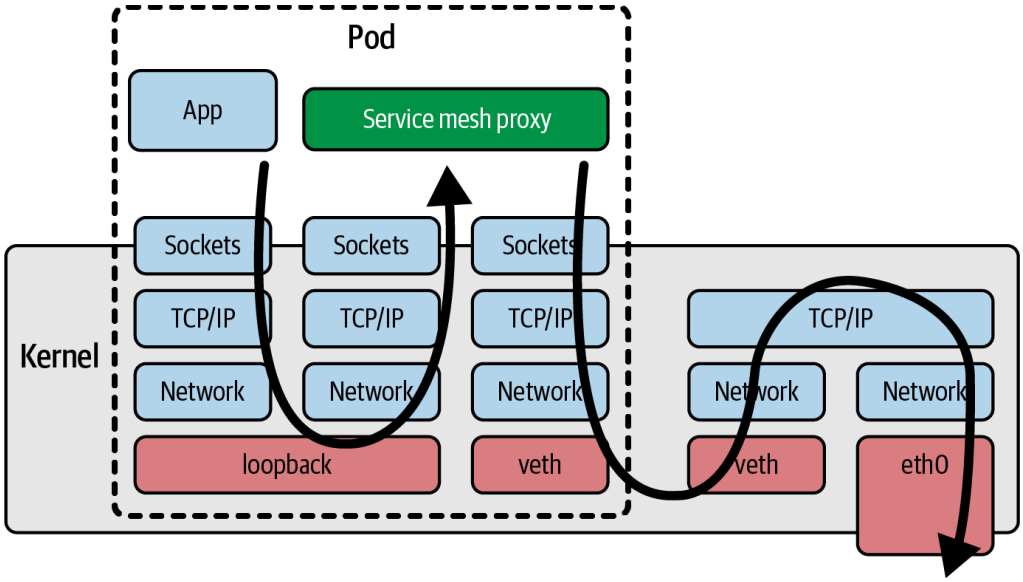


*Рисунок 1-5. Путь сетевого пакета с использованием вспомогательного контейнера прокси-сервера Service Mesh*

Все эти проблемы присущи модели с Sidecar. К счастью, теперь, когда eBPF доступен как платформа, у нас есть новая модель, позволяющая избежать этих проблем. Кроме того, поскольку инструменты на основе eBPF могут видеть всё что происходит на (виртуальной) машине, злоумышленникам сложнее их обойти. Например, если злоумышленнику удастся развернуть приложение для майнинга криптовалюты на одном из ваших хостов, он, вероятно, не окажет вам любезности, оснастив его дополнительными компонентами, которые вы используете для рабочих нагрузок своих приложений. Если вы полагаетесь на инструмент безопасности на основе Sidecar для предотвращения неожиданных сетевых подключений приложений, этот инструмент не обнаружит приложение для майнинга, подключающееся к

его пулу майнинга, если Sidecar не внедрен. Напротив, сетевая безопасность, реализованная в eBPF, может контролировать весь трафик на хост-компьютере, так что это операция по добыче криптовалюты может быть легко пресечена. Возможность отбрасывать сетевые пакеты из соображений безопасности — это то, к чему мы вернемся в главе 8.

## Итоги

Я надеюсь, что эта глава дала вам некоторое представление о том, почему eBPF как платформа настолько эффективна. Это позволяет нам изменять поведение ядра, предоставляя нам гибкость для создания специальных инструментов или настраиваемых политик. Инструменты на основе eBPF могут отслеживать любое событие в ядре и, следовательно, во всех приложениях, работающих на (виртуальной) машине, независимо от того, контейнеризованы они или нет. Программы eBPF также можно развертывать динамически, что позволяет изменять поведение на лету. До сих пор мы обсуждали eBPF на относительно концептуальном уровне. В следующей главе мы сделаем его более конкретным и рассмотрим составные части приложения на основе eBPF.