# 4 Трассировка с помощью BPF

В программной инженерии трассировка — это метод сбора данных для профилирования и отладки. Цель этих действий — собрать во время выполнения полезную информацию для последующего анализа. Основное преимущество использования BPF для трассировки в том, что вы можете получить доступ практически к любой части информации ядра Linux с помощью своих приложений. При этом BPF не дает заметного снижения производительности и не приводит к большей загрузке системы, по сравнению с другими технологиями трассировки, и от разработчиков не требуется вносить изменения в приложения только с целью сбора данных.

Ядро Linux предоставляет несколько инструментальных возможностей, которые можно использовать вместе с BPF. В этой главе мы поговорим о них. Мы покажем, как все происходит в ядре и как найти информацию, доступную для ваших программ BPF.

Цель трассировки — дать вам глубокое понимание любой системы, получив все доступные данные и представив их вам наиболее полезным способом. Мы поговорим о нескольких разных представлениях данных и о том, как вы можете применить их в разных сценариях.

Начиная с этой главы, мы собираемся задействовать мощный инструментарий для написания BPF-программ — BPF Compiler Collection (BCC). Отметьте, пожалуйста, что GCC — это основной компилятор программ для UNIX. BCC — это набор компонентов, которые делают построение BPF-программ более предсказуемым. Даже если вы владеете Clang и LLVM, то, вероятно, не захотите тратить больше времени, чем необходимо, на создание одних и тех же утилит и обеспечение того, чтобы верификатор BPF не отбрасывал

ваши программы. BCC предоставляет повторно используемые компоненты для общих структур, таких как карты событий Perf, и интеграцию с бэкендом LLVM, чтобы задействовать наилучшие варианты отладки. Кроме того, BCC включает привязки для нескольких языков программирования — в наших примерах использован Python. Эти привязки позволяют вам писать часть программ BPF пользовательского пространства на языке высокого уровня, что значительно упрощает создание программ. В последующих главах мы также применим BCC, чтобы показать реальные примеры.

Первым шагом для трассировки программ в ядре Linux является определение точек расширения, которые оно предоставляет для присоединения программ BPF. Эти точки обычно называют *зондами*.

## Зонды

Вот одно из определений в словаре английского языка для слова *«зонд»*: «Беспилотный исследовательский космический корабль, предназначенный для передачи информации об окружающей среде».

Это определение вызывает у нас (вероятно, у вас тоже) воспоминания о научно-фантастических фильмах и эпических миссиях НАСА. Говоря о трассировке с помощью зондов, можно использовать очень похожее определение: «Трассировочные зонды — это исследовательские программы, предназначенные для передачи информации о среде, в которой они работают».

Зонды собирают данные в вашей системе и предоставляют эту информацию для изучения и анализа. Традиционно их применение в Linux подразумевало написание программ, которые были бы скомпилированы в модули ядра, что могло вызвать катастрофические последствия в производственных системах. С годами они стали более безопасными для выполнения, но все еще были громоздкими в написании и тестировании. Такие инструменты, как SystemTap, ввели новые протоколы для написания зондов и упростили получение более конкретной информации из ядра Linux и всех программ, работающих в пространстве пользователя.

BPF использует трассировки для сбора, отладки и анализа информации. Безопасная природа программ BPF делает их более привлекательными, чем те инструменты, которые все еще требуют перекомпиляции ядра. Перекомпиляция ядра с включением внешних модулей может повысить вероятность

сбоев из-за неправильного поведения кода. Верификатор BPF устраняет этот риск, анализируя программу перед загрузкой в ядро. Разработчики BPF (BPF — это сетевой фильтр, разработанный командой FreeBSD, отсюда и название) использовали определения для зондов и изменили ядро так, чтобы, когда встречается одно из этих определений, выполнялись программы BPF, а не модули ядра.

Понимание того, что представляют собой различные типы зондов, которые вы можете задать, чрезвычайно важно для исследования происходящего в вашей системе. В этом разделе мы классифицируем различные определения зондов и расскажем, как их обнаруживать в вашей системе и как связывать с ними BPF-программы.

В этой главе рассмотрим четыре типа зондов.

- *Зонды ядра*. Предоставляют динамический доступ к внутренним компонентам ядра.
- *Точки трассировки*. Обеспечивают статический доступ к внутренним компонентам ядра.
- *Зонды в пользовательском пространстве*. Дают динамический доступ к программам, работающим в пользовательском пространстве.
- *Статически определенные пользователем точки трассировки*. Обеспечивают статический доступ к программам, запущенным в пространстве пользователя.

Начнем с зондов ядра.

## Зонды ядра

Зонды ядра позволяют с минимальными издержками устанавливать динамические флаги или точки останова практически в любой инструкции ядра. Дойдя до одного из этих флагов, ядро выполняет код, прикрепленный к тесту, а затем возобновляет обычную работу. Зонды ядра могут дать вам информацию обо всем, что происходит в системе, например об открытых в ней файлах и исполняемых двоичных файлах. Важно иметь в виду, что у зондов ядра нет стабильного двоичного интерфейса приложения (ABI), то есть они могут меняться между версиями ядра. Один и тот же код может перестать работать, если вы попытаетесь подключить один и тот же зонд к двум системам с разными версиями ядра.

Зонды ядра делятся на две категории — *kprobes* и *kretprobes*. Их использование зависит от того, где в цикле выполнения вы можете вставить программу BPF. В этом разделе рассказывается, как с помощью каждой из них присоединять программы BPF к зондам и извлекать информацию из ядра.

## Kprobes

Kprobes позволяет вставлять программы BPF перед выполнением любой инструкции ядра. Вам нужно знать сигнатуру функции, которую предстоит исследовать (как говорилось ранее, это нестабильный ABI, поэтому вы должны быть осторожны при настройке зондов, если собираетесь запускать одну и ту же программу на разных версиях ядра). Когда в ходе работы ядра запускается инструкция, в которой вы установили зонд, оно попадает в ваш код, запускает вашу программу BPF и возвращается к выполнению в исходную точку.

Чтобы показать вам, как использовать kprobes, мы напишем программу BPF, которая выводит имя любого двоичного файла, выполняемого в вашей системе. В этом примере применен внешний интерфейс Python для инструментов BCC, но вы можете написать его с помощью любого другого инструмента BPF:

```
from bcc import BPF

bpf_source = """
int do_sys_execve(struct pt_regs *ctx, void filename, void argv, void envp)
{                                                                          ❶
  char comm[16];
  bpf_get_current_comm(&comm, sizeof(comm));
  bpf_trace_printk("executing program: %s", comm);
  return 0;
}
"""

bpf = BPF(text = bpf_source)                                               ❷
execve_function = bpf.get_syscall_fnname("execve")                         ❸
bpf.attach_kprobe(event = execve_function, fn_name = "do_sys_execve")      ❹
bpf.trace_print()
```

❶ Наша BPF-программа начинает работать. Помощник `bpf_get_current_comm` извлечет имя текущей команды, под управлением которой работает ядро, и сохранит его в переменной `comm`. Мы определили это как массив фиксиро-

ванной длины, потому что ядро имеет ограничение 16 символов для имен команд. После получения имени команды мы печатаем его в трассировке отладки, чтобы человек, который запустил сценарий Python, смог увидеть все команды, отображаемые BPF.

❷ Загрузка программы BPF в ядро.

❸ Связывание программы с системным вызовом `execve`. Имя этого системного вызова меняется в разных версиях ядра, и BCC предоставляет функцию для его получения, причем не требуется запоминать, какую версию ядра вы используете.

❹ Код выводит журнал трассировки, поэтому вы можете видеть все команды, которые вы отслеживаете с помощью этой программы.

## Kretprobes

Kretprobes запустит вашу программу BPF после выполнения ядром определенной инструкции и вернет значение. Обычно и kprobes, и kretprobes объединяются в одну BPF-программу, чтобы иметь полное представление о поведении инструкции.

Используем пример, аналогичный приведенному в предыдущем разделе, чтобы показать вам, как работает kretprobes:

```
from bcc import BPF

bpf_source = """
int ret_sys_execve(struct pt_regs *ctx) {                                    ❶
  int return_value;
  char comm[16];
  bpf_get_current_comm(&comm, sizeof(comm));
  return_value = PT_REGS_RC(ctx);

  bpf_trace_printk("program: %s, return: %d", comm, return_value);
  return 0;
}
"""

bpf = BPF(text = bpf_source)                                                 ❷
execve_function = bpf.get_syscall_fnname("execve")
bpf.attach_kretprobe(event = execve_function, fn_name = "ret_sys_execve") ❸
bpf.trace_print()
```

❶ Определение функции, которая реализует программу BPF. Ядро выполнит ее сразу после завершения системного вызова `execve`. `PT_REGS_RC` — это макрос, который будет считывать возвращенное значение из реестра BPF для этого конкретного контекста. Мы также используем `bpf_trace_printk`, чтобы отметить команду и возвращенное ей значение в нашем журнале отладки.

❷ Инициализация программы BPF и ее загрузка в ядро.

❸ Замена присоединенной функции на `attach_kretprobe`.

**ЧТО ТАКОЕ АРГУМЕНТ КОНТЕКСТА**

Возможно, вы заметили, что у обеих BPF-программ первый аргумент в присоединенной функции один и тот же — `ctx`. Этот параметр, называемый контекстом, дает доступ к информации, которую ядро обрабатывает в настоящее время. Следовательно, контекст зависит от типа программы BPF, которую вы используете в данный момент. Процессор будет хранить информацию о текущей задаче, которую выполняет ядро. Эта структура также зависит от архитектуры вашей системы: набор регистров процессора ARM отличается от имеющегося в процессоре x64. Вы можете получить доступ к этим регистрам, не беспокоясь об архитектуре, с помощью макросов, определенных в ядре, например, `PT_REGS_RC`.

Зонды ядра — это мощный способ доступа к ядру. Но как мы упоминали ранее, они могут быть нестабильными, потому что вы привязываетесь к динамическим точкам в исходном коде ядра, которые могут измениться или исчезнуть в следующей версии. Есть другой, более безопасный способ прикрепления программ к ядру.

## Точки трассировки

Точки трассировки — это статичные маркеры в коде ядра, которые можно использовать для присоединения кода в работающем ядре. Основное их отличие от kprobes заключается в том, что они встроены в ядро разработчиками. Вот почему мы называем их статичными. ABI для точек трассировки более стабилен: ядро всегда гарантирует, что точки трассировки, имеющиеся в старой версии, будут существовать и в новых версиях. Однако, учитывая, что разработчикам необходимо добавить их в ядро, они не всегда охватывают все подсистемы, из которых состоит ядро.

Как мы упоминали в главе 2, вы можете найти все доступные точки трассировки в своей системе, просмотрев все файлы в `/sys/kernel/debug/tracing/events`. Например, можете найти все точки трассировки для самого BPF, глядя на события, определенные в `/sys/kernel/debug/tracing/events/bpf`:

```
sudo ls -la /sys/kernel/debug/tracing/events/bpf
total 0
drwxr-xr-x  14 root root 0 Feb  4 16:13 .
drwxr-xr-x 106 root root 0 Feb  4 16:14 ..
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_create
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_delete_elem
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_lookup_elem
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_next_key
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_map_update_elem
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_get_map
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_get_prog
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_pin_map
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_obj_pin_prog
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_prog_get_type
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_prog_load
drwxr-xr-x   2 root root 0 Feb  4 16:13 bpf_prog_put_rcu
-rw-r--r--   1 root root 0 Feb  4 16:13 enable
-rw-r--r--   1 root root 0 Feb  4 16:13 filter
```

Каждый подкаталог, указанный в этих выходных данных, соответствует точке трассировки, к которой мы можем присоединить программы BPF. Но там есть еще два дополнительных файла. Первый файл, `enable`, позволяет включать и отключать все точки трассировки для подсистемы BPF. Если содержимое файла равно `0`, точки трассировки отключены, если `1` — включены. Файл `filter` позволяет задать способ фильтрования событий подсистемой Trace в ядре. BPF не использует данный файл. Подробнее об этом можно узнать из документации по трассировке ядра (oreil.ly/miNRd).

Написание программ BPF с применением точек трассировки аналогично трассировке с помощью kprobes. Вот пример, который задействует программу BPF для трассировки всех приложений в вашей системе, загружающих другие программы BPF:

```
from bcc import BPF

bpf_source = """
int trace_bpf_prog_load(void ctx) {                    ❶
  char comm[16];
  bpf_get_current_comm(&comm, sizeof(comm));
```

```
  bpf_trace_printk("%s is loading a BPF program", comm);
  return 0;
}
"""

bpf = BPF(text = bpf_source)
bpf.attach_tracepoint(tp = "bpf:bpf_prog_load",
                      fn_name = "trace_bpf_prog_load")    ❷
bpf.trace_print()
```

❶ Объявление функции, которая определяет программу BPF. Этот код уже должен быть вам знаком. Есть лишь несколько синтаксических изменений в первом примере, который вы видели, когда мы говорили о kprobes.

❷ Основное отличие этой программы: вместо того чтобы прикреплять программу к kprobe, мы прикрепляем ее к точке трассировки. BCC придерживается соглашения об именовании точек трассировки: сначала указывается подсистема для трассировки — в данном случае `bpf`, затем двоеточие, за которым следует точка трассировки в подсистеме `pbf_prog_load`. Это означает, что каждый раз, когда ядро выполняет функцию `bpf_prog_load`, программа получит событие и выведет имя приложения, которое выполняет данную инструкцию `bpf_prog_load`.

Зонды ядра и точки трассировки предоставят вам полный доступ к ядру. Мы рекомендуем использовать точки трассировки всякий раз, когда это возможно, но вы не обязаны применять их только потому, что они безопаснее. Воспользуйтесь преимуществами динамических возможностей зондов ядра. В следующем разделе мы обсудим, как получить аналогичный уровень наблюдаемости в программах, работающих в пользовательском пространстве.

## Зонды пользовательского пространства

Зонды пользовательского пространства позволяют устанавливать динамические флаги в программах, работающих в пользовательском пространстве. Они являются эквивалентом зондов ядра для программ, работающих вне ядра. Когда вы определяете uprobe, ядро создает ловушку вокруг прикрепленной инструкции. Когда ваше приложение достигнет этой инструкции, ядро запускает событие, использующее зондовую функцию в качестве обратного вызова. uprobe также дают вам доступ к любой библиотеке, с которой свя-

зана ваша программа, и вы можете отслеживать такие вызовы, если знаете правильное имя инструкции.

Как и зонды ядра, зонды пользовательского пространства подразделяются на две категории: uprobes и uretprobes — в зависимости от того, где в цикле выполнения вы можете вставить свою BPF-программу. Рассмотрим несколько примеров.

## Uprobes

Вообще говоря, uprobes — это ловушки, которые ядро вставляет в набор команд программы перед выполнением конкретной инструкции. Вы должны быть осторожны, когда присоединяете uprobes к различным версиям одной и той же программы, потому что сигнатуры функций могут меняться от версии к версии. Единственный способ гарантировать, что программа BPF будет работать в разных версиях, — это убедиться в том, что подпись не изменилась. Можете использовать команду `nm` в Linux, чтобы получить список всех символов, включенных в объектный файл ELF, что является хорошим способом проверить, существует ли еще инструкция, которую вы отслеживаете, в вашей программе, например:

```
package main
import "fmt"

func main() {
        fmt.Println("Hello, BPF")
}
```

Вы можете скомпилировать программу Go, используя `go build -o hello-bpf main.go`. Можно ввести команду `nm`, чтобы получить информацию обо всех точках инструкций, которые содержит двоичный файл. `nm` — это программа, включенная в GNU Development Tools, которая выводит символы из объектных файлов. Если вы отфильтруете то, что содержит `main`, то получите список, подобный следующему:

```
nm hello-bpf | grep main
0000000004850b0 T main.init
00000000567f06 B main.initdone.
00000000485040 T main.main
000000004c84a0 R main.statictmp_0
00000000428660 T runtime.main
```

```
0000000044da30 T runtime.main.func1
00000000044da80 T runtime.main.func2
000000000054b928 B runtime.main_init_done
00000000004c8180 R runtime.mainPC
0000000000567f1a B runtime.mainStarted
```

Теперь, когда у вас есть список символов, можете отслеживать, когда они выполняются, даже между разными процессами, выполняющими один и тот же бинарный код.

Чтобы отследить, когда будет выполнена основная функция в предыдущем примере, напишем программу BPF и прикрепим ее к uprobe, который прервется до того, как какой-либо процесс вызовет эту инструкцию:

```
from bcc import BPF

bpf_source = """
int trace_go_main(struct pt_regs *ctx) {
  u64 pid = bpf_get_current_pid_tgid();                    ❶
  bpf_trace_printk("New hello-bpf process running with PID: %d", pid);
}
"""

bpf = BPF(text = bpf_source)
bpf.attach_uprobe(name = "hello-bpf",
    sym = "main.main", fn_name = "trace_go_main")    ❷
bpf.trace_print()
```

❶ Используем функцию `bpf_get_current_pid_tgid`, чтобы получить идентификатор процесса (PID), который выполняет программу `hello-bpf`.

❷ Подключаем эту программу к uprobe. Вызов должен знать, что объект, который мы хотим отследить, `hello-bpf`, является абсолютным путем к объектному файлу. Ему также нужны символ, который мы отслеживаем внутри объекта, — в данном случае `main.main` — и программа BPF, которую хотим запустить. При этом каждый раз, когда кто-то запускает `hello-bpf` в вашей системе, мы получаем новую запись трассировки.

## Uretprobes

Uretprobes — это то же самое, что и зонд kretprobes, но для программ пользовательского пространства. Они присоединяют программы BPF к инструкциям, возвращающим значения, и предоставляют вам возвращенные значения для получения доступа к регистрам из вашего кода BPF.

Комбинирование uprobes и uretprobes позволяет вам писать более сложные программы BPF. Они могут дать более целостное представление о приложениях, работающих в вашей системе. Если вы вставите код трассировки до запуска функции и сразу после ее завершения, то сможете собрать больше данных и оценить поведение приложения. Обычный вариант — измерение времени выполнения функции без отслеживания каждой строки кода в своем приложении.

Мы повторно используем программу Go, описанную в разделе «Uprobes», чтобы измерить, сколько времени потребуется для выполнения основной функции. Этот пример BPF длиннее, чем предыдущие, поэтому мы разделили его на несколько блоков:

```
bpf_source = """
BPF_HASH(cache, u64, u64);                          ❶

int trace_start_time(struct pt_regs *ctx) {
  u64 pid = bpf_get_current_pid_tgid();
  u64 start_time_ns = bpf_ktime_get_ns();           ❷
  cache.update(&pid, &start_time_ns);               ❸
  return 0;
}
"""
```

❶ Создание хеш-карты BPF. Эта таблица позволяет функциям uprobe и uretprobe обмениваться данными. В этом случае мы используем PID приложения в качестве ключа таблицы и сохраняем время запуска функции в качестве значения. Две наиболее интересные операции в функции `uprobe` выполняются так, как описано далее.

❷ Получаем текущее время в системе в наносекундах, как задано ядром.

❸ Создаем запись в нашем кэше с PID программы и текущим временем. Можно предположить, что это время запуска функции приложения.

Теперь объявим функцию uretprobe. Реализуйте функцию, которую нужно вызвать, когда ваша инструкция отработает. Эта функция uretprobe похожа на другие, которые вы видели в разделе «Kretprobes»:

```
bpf_source += """
static int print_duration(struct pt_regs *ctx) {
  u64 pid = bpf_get_current_pid_tgid();                    ❶
  u64 start_time_ns = cache.lookup(&pid);
  if (start_time_ns == 0) {
```

```
    return 0;
  }
  u64 duration_ns = bpf_ktime_get_ns() - start_time_ns;
  bpf_trace_printk("Function call duration: %d", duration_ns);    ❷
  return 0;                                                       ❸
}
"""
```

❶ Получаем PID для нашего приложения — это нужно для того, чтобы определить момент старта. Используем функцию `lookup`, чтобы извлечь это время из карты, где мы его сохранили до запуска функции.

❷ Рассчитываем продолжительность выполнения функции, определив разницу во времени.

❸ Отмечаем задержку в нашем журнале трассировки, чтобы потом отобразить ее в терминале.

Теперь остальная часть программы должна прикрепить две функции BPF к правильным зондам:

```
bpf = BPF(text = bpf_source)
bpf.attach_uprobe(name = "hello-bpf", sym = "main.main",
           fn_name = "trace_start_time")
bpf.attach_uretprobe(name = "hello-bpf", sym = "main.main",
           fn_name = "print_duration")
bpf.trace_print()
```

Мы добавили строку в оригинальный пример uprobe там, где к uretprobe присоединяется функция печати для приложения.

В этом разделе вы увидели, как отслеживать операции, которые протекают в пространстве пользователя, с помощью BPF. Комбинируя функции BPF, выполняющиеся в разные моменты жизненного цикла приложения, можно получить гораздо более богатую информацию. Однако, как упоминалось в начале этого раздела, хотя зонды пользовательского пространства являются мощным средством, они также нестабильны. Функции BPF могут перестать работать только потому, что кто-то изменит имя функции приложения. А сейчас рассмотрим более устойчивый способ трассировки программ пользовательского пространства.

## Статические точки трассировки пользовательского пространства

Статически определенные пользователем точки трассировки (USDT) — это статические точки трассировки для приложений в пользовательском пространстве. Их совокупное использование — это удобный способ для инструментов приложений, потому что накладные расходы на точку входа в возможности трассировки, которые предлагает BPF, невелики. Вы можете использовать их и как соглашение для отслеживания приложений в рабочей среде, независимо от языка программирования, на котором они написаны.

USDT были применены в DTrace, инструменте, изначально разработанном в Sun Microsystems для динамического инструментария систем Unix. До недавнего времени DTrace не был доступен в Linux из-за проблем с лицензированием, однако разработчики ядра Linux исходили из опыта создания DTrace при реализации USDT.

Подобно статическим точкам трассировки ядра, USDT требуют, чтобы разработчики дополнили свой код инструкциями, которые ядро будет использовать в качестве ловушек для выполнения программ BPF. Версия USDTs Hello World состоит всего из нескольких строк кода:

```
#include <sys/sdt.h>
int main() {
  DTRACE_PROBE("hello-usdt", "probe-main");
}
```

В этом примере мы применим макрос, который Linux предоставляет для определения нашего первого USDT. Вы уже можете видеть, откуда ядро получает информацию. `DTRACE_PROBE` регистрирует точку трассировки, которую ядро будет использовать для внедрения нашего обратного вызова функции BPF. Первым аргументом в этом макросе является программа, сообщающая о трассировке, вторым — название трассы, о которой мы хотим узнать.

Многие приложения, которые вы, возможно, установили в своей системе, используют этот тип проверки для доступа к данным трассировки во время выполнения каким-либо предсказуемым образом. Например, популярная база данных MySQL предоставляет все виды информации, задействуя статически определенные точки трассировки. Вы можете собирать информацию

из запросов, выполняемых на сервере, а также из многих других пользовательских операций. Node.js — среда выполнения JavaScript, построенная на основе движка Chrome V8, — также предоставляет точки трассировки, которые можно использовать для извлечения информации о времени выполнения.

Прежде чем показать вам, как присоединять программы BPF к определенной пользовательской точке трассировки, нужно поговорить о том, как их обнаружить. Поскольку точки трассировки определены в двоичном формате внутри исполняемых файлов, нам нужен способ найти список зондов, определенных программой, не копаясь в исходном коде. Одним из способов извлечь эту информацию является непосредственное чтение двоичного файла в формате ELF. Мы можем перекомпилировать предыдущий пример Hello World USDT с помощью GCC:

```
gcc -o hello_usdt hello_usdt.c
```

Эта команда сгенерирует двоичный файл с именем `hello_usdt`, с помощью которого можно применить несколько инструментов для нахождения точек трассировки. Linux предоставляет утилиту `readelf` для отображения информации о файлах ELF. Испробуйте ее на только что скомпилированном примере:

```
readelf -n ./hello_usdt
```

Вы можете увидеть USDT, который мы определили на основе вывода команды:

```
Displaying notes found in: .note.stapsdt
  Owner             Data size      Description
  stapsdt           0x00000033     NT_STAPSDT (SystemTap probe descriptors)
  Provider: "hello-usdt"
  Name: "probe-main"
```

`readelf` способна дать намного больше информации о двоичном файле. В нашем небольшом примере она показывает лишь несколько строк информации, но ее вывод слишком подробен для анализа более сложных двоичных файлов.

Лучшим вариантом для обнаружения точек трассировки, определенных в двоичном файле, является использование инструмента BCC `tplist`, который может отображать как точки трассировки ядра, так и USDT. Преимущество этого инструмента в простоте его вывода: он показывает только опреде-

ления точек трассировки без дополнительной информации об исполняемом файле. Используется примерно так же, как `readelf`:

```
tplist -l ./hello_usdt
```

Здесь перечислены все точки трассировки, которые вы задаете. В нашем примере отображается только одна строка с определением `probe-main`:

```
./hello_usdt "hello-usdt":"probe-main"
```

После того как вы узнаете поддерживаемые точки трассировки в своем двоичном файле, можете присоединить к ним BPF-программы аналогично тому, как было показано в предыдущих примерах:

```
from bcc import BPF, USDT

bpf_source = """
#include <uapi/linux/ptrace.h>
int trace_binary_exec(struct pt_regs *ctx) {
  u64 pid = bpf_get_current_pid_tgid();
  bpf_trace_printk("New hello_usdt process running with PID: %d", pid);
}
"""

usdt = USDT(path = "./hello_usdt")                                          ❶
usdt.enable_probe(probe = "probe-main", fn_name = "trace_binary_exec")      ❷
bpf = BPF(text = bpf_source, usdt = usdt)                                   ❸
bpf.trace_print()
```

В этом примере есть серьезное изменение, которое требует объяснения.

❶ Создать объект USDT. Мы не делали этого в предыдущих примерах. USDT не являются частью BPF, то есть вы можете использовать их, не взаимодействуя с виртуальной машиной BPF. Поскольку они не зависят друг от друга, их применение не зависит от кода BPF.

❷ Присоединить функцию BPF для отслеживания выполнения программы к зонду в нашем приложении.

❸ Инициализировать среду BPF с помощью определения точки трассировки, которое мы только что создали. Это сообщит BCC, что ему необходимо сгенерировать код для соединения нашей программы BPF с определением зонда в создаваемом двоичном файле. Когда они оба присоединены, мы можем получить трассировки, сгенерированные BPF-программой, чтобы отследить выполнение основной программы.

## USDT и другие языки программирования

Можно использовать USDT также для отслеживания приложений, написанных на других языках программирования, помимо C. Вы найдете на GitHub привязки для Python, Ruby, Go, Node.js и многих других языков. Привязки для Ruby являются одними из наших любимых из-за их простоты и совместимости с такими фреймворками, как Rails. Дейл Хэмел, который в настоящее время работает в Shopify, написал в своем блоге отличный отчет о применении USDT (https://oreil.ly/7pgNO). Он также поддерживает библиотеку *ruby-static-tracing* (https://oreil.ly/ge6cu), которая делает отслеживание приложений Ruby и Rails еще более простым.

Библиотека статической трассировки Хэмела позволяет задействовать возможности трассировки на уровне класса, не требуя добавления логики трассировки для каждого метода в этом классе. В сложных сценариях это обеспечивает удобные методы самостоятельной регистрации выделенных конечных точек трассировки.

Чтобы использовать `ruby-static-tracing` в своих приложениях, сначала укажите, когда будут включены точки трассировки. Можете включить их по умолчанию при запуске приложения, но если хотите избежать непрерывного сбора данных, активируйте их с помощью сигнала системного вызова. Дэйл Хэмел рекомендует взять в качестве этого сигнала `PROF`:

```
require 'ruby-static-tracing'

StaticTracing.configure do |config|
  config.mode = StaticTracing::Configuration::Modes::SIGNAL
  config.signal = StaticTracing::Configuration::Modes::SIGNALS::SIGPROF
end
```

В такой конфигурации вы можете добавить команду `kill` для включения статических точек трассировки вашего приложения по желанию. В следующем примере мы предполагаем, что на машине работает только процесс Ruby, и можем получить его идентификатор процесса с помощью `pgrep`:

```
kill -SIGPROF 'pgrep -nx ruby'
```

Помимо настройки активизации точек трассировки, вы можете использовать некоторые встроенные механизмы трассировки, которые предоставляет `ruby-static-tracing`. На момент написания книги библиотека содержала точки трассировки для измерения задержки и сбора трассировок стека.

Нам очень нравится, что с помощью этого встроенного модуля утомительная задача измерения задержки функции становится почти тривиальной. Вначале нужно добавить в первоначальную конфигурацию трассировщик задержки:

```
require 'ruby-static-tracing'
require 'ruby-static-tracing/tracer/concerns/latency_tracer'

StaticTracing.configure do |config|
  config.add_tracer(StaticTracing::Tracer::Latency)
end
```

После этого каждый класс, включающий модуль задержки, генерирует статические точки трассировки для каждого определенного открытого метода. Когда трассировка включена, вы можете запрашивать эти точки для сбора данных о времени. В следующем примере `ruby-static-tracing` генерирует статическую точку трассировки с именем `usdt:/proc/X/fd/:user_model:find`, выполняя соглашение об использовании имени класса в качестве пространства имен для точки трассировки и имени метода — в качестве имени точки трассировки:

```
class UserModel
  def find(id)
  end

  include StaticTracing::Tracer::Concerns::Latency
end
```

Теперь можно задействовать BCC для извлечения информации о задержке для каждого вызова метода `find`. Для этого мы используем встроенные функции BCC `bpf_usdt_readarg` и `bpf_usdt_readarg_p`. Они читают аргументы, устанавливаемые каждый раз, когда выполняется код нашего приложения. `ruby-static-tracing` всегда задает имя метода в качестве первого аргумента для точки трассировки, тогда как в качестве второго аргумента устанавливается вычисленное значение. Следующий фрагмент реализует программу BPF, которая получает информацию о точке трассировки и выводит ее в журнале трассировки:

```
bpf_source = """
#include <uapi/linux/ptrace.h>
int trace_latency(struct pt_regs *ctx) {
  char method[64];
  u64 latency;
```

```
  bpf_usdt_readarg_p(1, ctx, &method, sizeof(method));
  bpf_usdt_readarg(2, ctx, &latency);

  bpf_trace_printk("method %s took %d ms", method, latency);
}
"""
```

Нам также нужно загрузить предыдущую BPF-программу в ядро. Поскольку мы отслеживаем конкретное приложение, которое уже запущено на компьютере, то можем прикрепить программу к определенному идентификатору процесса:

```
parser = argparse.ArgumentParser()
parser.add_argument("-p", "--pid", type = int, help = "Process ID")    ❶
args = parser.parse_args()

usdt = USDT(pid = int(args.pid))
usdt.enable_probe(probe = "latency", fn_name = "trace_latency")        ❷
bpf = BPF(text = bpf_source, usdt = usdt)
bpf.trace_print()
```

❶ Указываем PID.

❷ Включаем зонд, загружаем программу в ядро и печатаем журнал трассировки. (Этот раздел очень похож на тот, который вы видели ранее.)

Мы показали, как анализировать приложения, которые статически определяют точки трассировки. Многие известные библиотеки и языки программирования включают в себя эти зонды, что помогает отлаживать запущенные приложения, особенно чтобы отслеживать их работу в производственных средах. Однако это лишь верхушка айсберга — получив данные, вы должны разобраться в них. Это именно то, чем мы займемся в дальнейшем.

## Визуализация данных трассировки

Пока мы приводили примеры, которые выводят данные в нашем отладочном отчете. Это не очень полезно в производственной среде. Разобраться в этих данных стоит, однако никто не любит копаться в длинных сложных журналах. Если мы хотим отслеживать изменения в том, почему происходят задержки и загрузка ЦП, лучше просмотреть графики за определенный период, чем разбирать числа из файла.