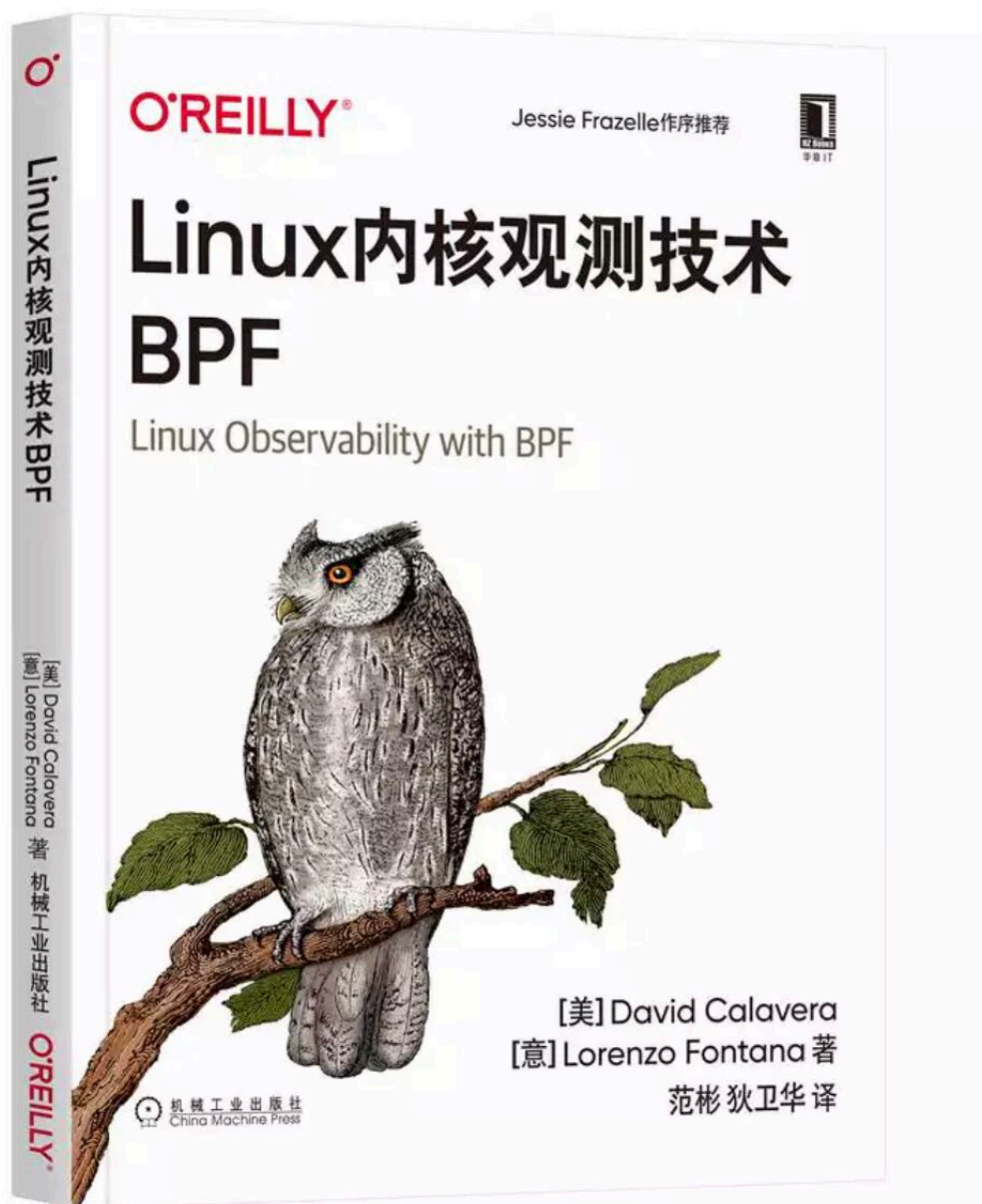


# eBPF 技术简介

---



由范老师和我一起翻译的图书《Linux内核观测技术BPF》已经在 JD 上有现货，欢迎感兴趣 BPF 技术的同学选购。链接地址 <https://item.jd.com/72110825905.html>

“eBPF 是我见过的 Linux 中最神奇的技术，没有之一，已成为 Linux 内核中顶级子模块，从 tcpdump 中用作网络包过滤的经典 cbpf，到成为通用 Linux 内核技术的 eBPF，已经完成华丽蜕变，为应用与神奇的内核打造了一座桥梁，在系统跟踪、观测、性能调优、安全和网络等领域发挥重要的角色。为 Service Mesh 打造了具备 API 感知和安全高效的容器网络方案 Cilium，其底层正是基于 eBPF 技术”

## 1. BPF

---

BPF (Berkeley Packet Filter)，中文翻译为伯克利包过滤器，是类 Unix 系统上数据链路层的一种原始接口，提供原始链路层封包的收发。1992 年，Steven McCanne 和 Van Jacobson 写了一篇名为《BSD数据包过滤：一种新的用户级包捕获架构》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。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。这样可以最大程度地减少 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，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。

tcpdump 的底层采用 BPF 作为底层包过滤技术，我们可以在命令后面增加“-d”来查看 tcpdump 过滤条件的底层汇编指令。

```
1  $ tcpdump -d 'ip and tcp port 8080'
2  (000) ldh      [12]
3  (001) jeq      #0x800          jt 2   jf 12
4  (002) ldb      [23]
5  (003) jeq      #0x6           jt 4   jf 12
6  (004) ldh      [20]
7  (005) jset     #0x1fff        jt 12  jf 6
8  (006) ldx      4*([14]&0xf)
9  (007) ldh      [x + 14]
10 (008) jeq      #0x1f90        jt 11  jf 9
11 (009) ldh      [x + 16]
12 (010) jeq      #0x1f90        jt 11  jf 12
13 (011) ret      #262144
14 (012) ret      #0
```

图 1 tcpdump 底层汇编指令

BPF 工作在内核层，BPF 的架构图如下 [来自于bpf-usenix93]：

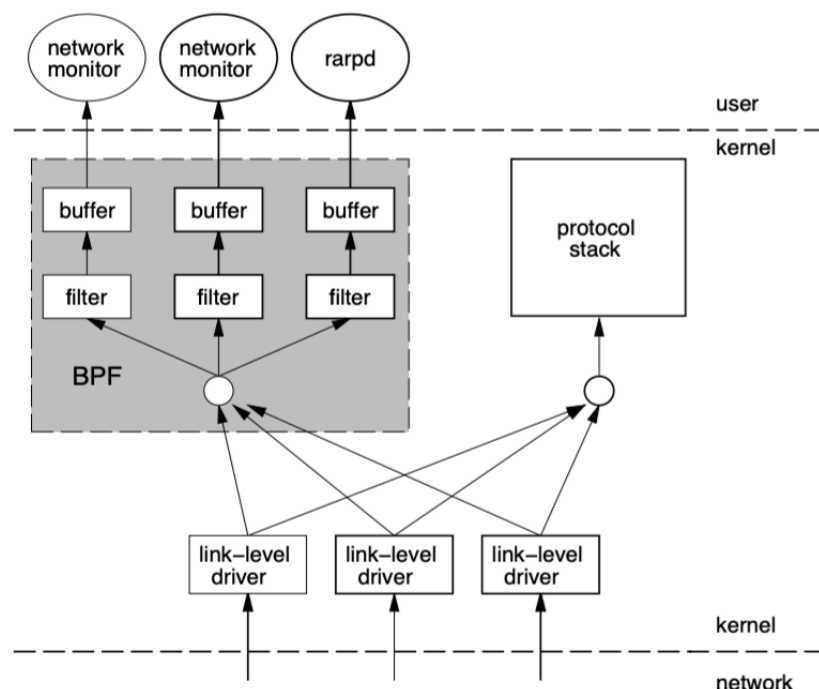


图 1 tcpdump 运行架构

## 2. eBPF

### 2.1 eBPF 介绍

2014 年初，Alexei Starovoitov 实现了 eBPF（extended Berkeley Packet Filter）。eBPF 最早出现在 3.18 内核中，此后原来的 BPF 就被称为经典 BPF，缩写 cBPF（classic BPF），cBPF 现在已经基本废弃。现在，Linux 内核只运行 eBPF，内核会将加载的 cBPF 字节码透明地转换成 eBPF 再执行。

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。

维度	cBPF	eBPF
寄存器数目	2个：A, X	10个：R0-R9, 另外 R10 是一个只读的帧指针
寄存器宽度	32位	64位
存储	16 个内存位: M[0-15]	512 字节堆栈，无限制大小的“map”存储
限制的内核调用	非常有限，仅限于 JIT 特定	有限，通过 bpf_call 指令调用
目标事件	数据包、seccomp-BPF	数据包、内核函数、用户函数、跟踪点 PMCs 等

表格 1 cBPF 与 eBPF 对比

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 程序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。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 扩展到用户空间，这也成为了 BPF 技术的转折点**。正如 Alexei 在提交补丁的注释中写到：“这个补丁展示了 eBPF 的潜力”。当前，eBPF 不再局限于网络栈，已经成为内核顶级的子系统。eBPF 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，eBPF 程序不需要重新编译内核，并且可以确保 eBPF 程序运行完成，而不会造成系统的崩溃。

于此同时，eBPF 也逐渐在系统跟踪、观测、性能调优、安全和网络等领域发挥重要的角色。Facebook、NetFlix、CloudFlare 等知名互联网公司内部广泛采用基于 eBPF 技术的各种程序用于性能分析、排查问题、负载均衡、防范 DDoS攻击，据相关信息显示在 Facebook 的机器上内置一系列 eBPF 的相关工具。

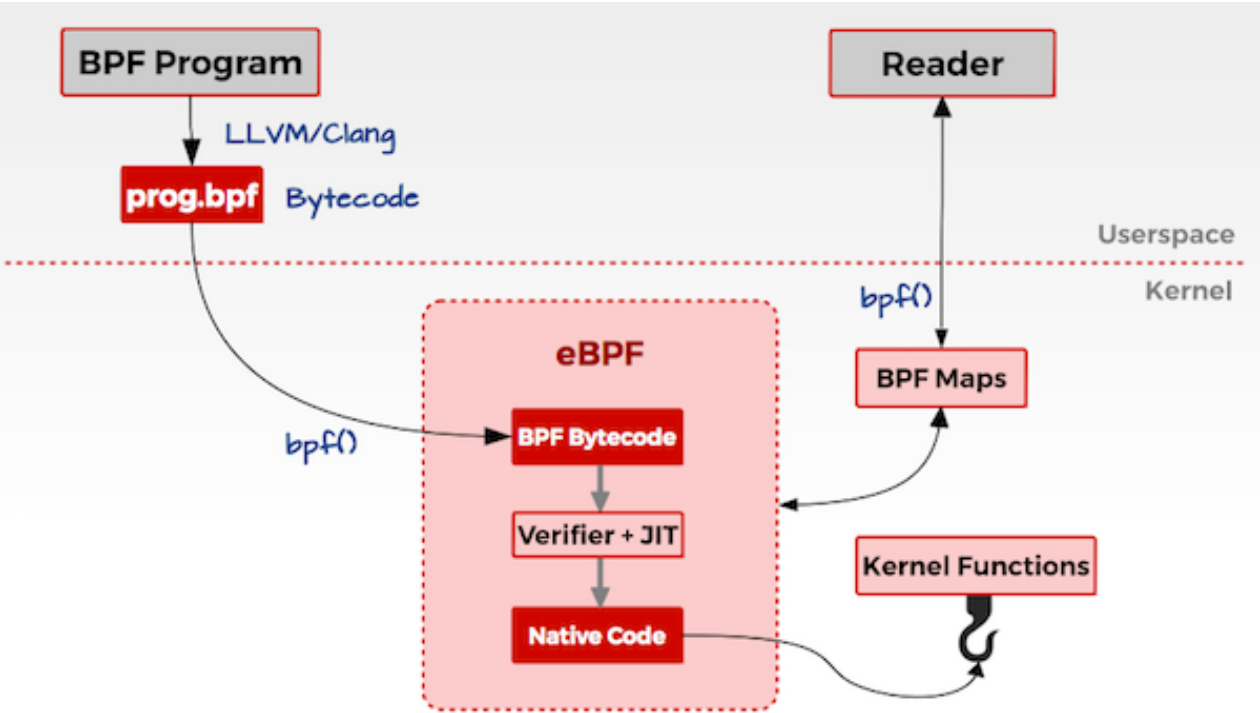


图 0 BPF 架构图

相对于系统的性能分析和观察，eBPF 技术在网络技术中的表现，更是让人眼前一亮，BPF 技术与 XDP (eXpress Data Path) 和 TC (Traffic Control) 组合可以实现功能更加强大的网络功能，更可为 SDN 软件定义网络提供基础支撑。XDP 只作用与网络包的 Ingress 层面，BPF 钩子位于网络驱动中尽可能早的位置，无需进行原始包的复制就可以实现最佳的数据包处理性能，挂载的 BPF 程序是运行过滤的理想选择，可用于丢弃恶意或非预期的流量、进行 DDOS 攻击保护等场景；而 TC Ingress 比 XDP 技术处于更高层次的位置，BPF 程序在 L3 层之前运行，可以访问到与数据包相关的大部分元数据，是本地节点处理的理想的地方，可以用于流量监控或者 L3/L4 的端点策略控制，同时配合 TC egress 则可实现对于容器环境下更高维度和级别的网络结构。

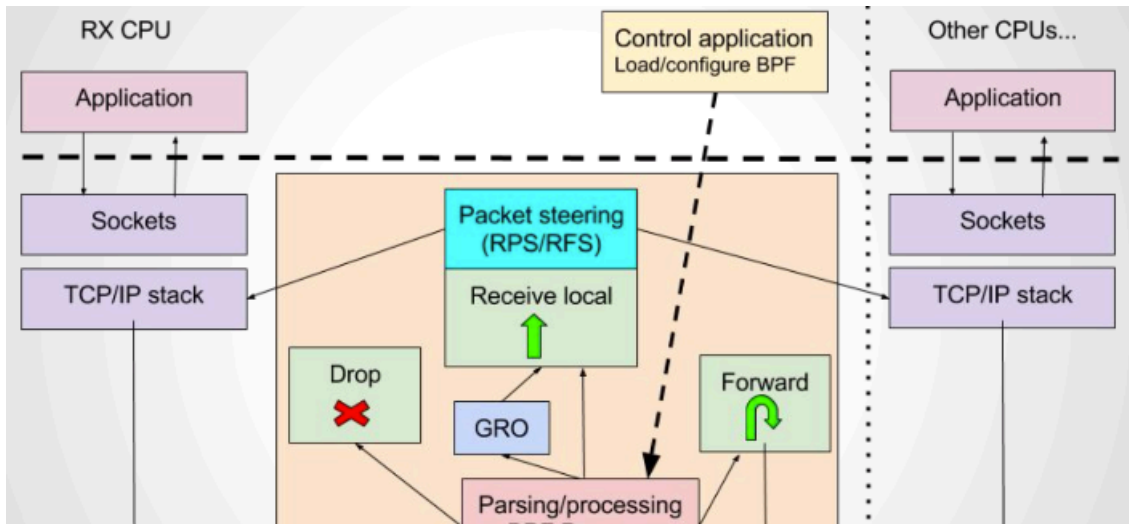


图 2 XDP 技术架构

eBPF 相关的知名的开源项目包括但不限于以下：

- Facebook 高性能 4 层负载均衡器 [Katran](#)；
- [Cilium](#) 为下一代微服务 ServiceMesh 打造了具备 API 感知和安全高效的容器网络方案；底层主要使用 XDP 和 TC 等相关技术；
- IO Visor 项目开源的 [BCC](#)、[BPFTrace](#) 和 [Kubectl-Trace](#)：[BCC](#) 提供了更高阶的抽象，可以让用户采用 Python、C++ 和 Lua 等高级语言快速开发 BPF 程序；[BPFTrace](#) 采用类似于 awk 语言快速编写 eBPF 程序；[Kubectl-Trace](#) 则提供了在 kubernetes 集群中使用 BPF 程序调试的方便操作；
- CloudFlare 公司开源的 [eBPF Exporter](#) 和 [bpf-tools](#)：[eBPF Exporter](#) 将 eBPF 技术与监控 Prometheus 紧密结合起来；[bpf-tools](#) 可用于网络问题分析和排查；

越来越多的基于 eBPF 的项目如雨后春笋一样开始蓬勃发展，而且逐步在社区中异军突起，成为一道风景线。比如 IO Visor 项目的 BCC 工具，为性能分析和观察提供了更加丰富的工具集：[图片来源](#)

图 3 Linux bcc/BPF 观测工具

同时，IO Visor 的 [bpf-docs](#) 包含了日常的文档，可以用于学习。

由于 eBPF 还在快速发展期，内核中的功能也日趋增强，一般推荐基于Linux 4.4+ (4.9 以上会更好) 内核的来使用 eBPF。部分 Linux Event 和 BPF 版本支持见下图：

## Linux Events & BPF Support

Linux Version	Events	BPF	Notes
2.6.18	Yes	No	
2.6.20	Yes	No	
2.6.22	Yes	No	
2.6.24	Yes	No	
2.6.26	Yes	No	
2.6.28	Yes	No	
2.6.30	Yes	No	
2.6.32	Yes	No	
2.6.34	Yes	No	
2.6.36	Yes	No	
2.6.38	Yes	No	
2.6.40	Yes	No	
2.6.42	Yes	No	
2.6.44	Yes	No	
2.6.46	Yes	No	
2.6.48	Yes	No	
2.6.50	Yes	No	
2.6.52	Yes	No	
2.6.54	Yes	No	
2.6.56	Yes	No	
2.6.58	Yes	No	
2.6.60	Yes	No	
2.6.62	Yes	No	
2.6.64	Yes	No	
2.6.66	Yes	No	
2.6.68	Yes	No	
2.6.70	Yes	No	
2.6.72	Yes	No	
2.6.74	Yes	No	
2.6.76	Yes	No	
2.6.78	Yes	No	
2.6.80	Yes	No	
2.6.82	Yes	No	
2.6.84	Yes	No	
2.6.86	Yes	No	
2.6.88	Yes	No	
2.6.90	Yes	No	
2.6.92	Yes	No	
2.6.94	Yes	No	
2.6.96	Yes	No	
2.6.98	Yes	No	
2.6.100	Yes	No	
2.6.102	Yes	No	
2.6.104	Yes	No	
2.6.106	Yes	No	
2.6.108	Yes	No	
2.6.110	Yes	No	
2.6.112	Yes	No	
2.6.114	Yes	No	
2.6.116	Yes	No	
2.6.118	Yes	No	
2.6.120	Yes	No	
2.6.122	Yes	No	
2.6.124	Yes	No	
2.6.126	Yes	No	
2.6.128	Yes	No	
2.6.130	Yes	No	
2.6.132	Yes	No	
2.6.134	Yes	No	
2.6.136	Yes	No	
2.6.138	Yes	No	
2.6.140	Yes	No	
2.6.142	Yes	No	
2.6.144	Yes	No	
2.6.146	Yes	No	
2.6.148	Yes	No	
2.6.150	Yes	No	
2.6.152	Yes	No	
2.6.154	Yes	No	
2.6.156	Yes	No	
2.6.158	Yes	No	
2.6.160	Yes	No	
2.6.162	Yes	No	
2.6.164	Yes	No	
2.6.166	Yes	No	
2.6.168	Yes	No	
2.6.170	Yes	No	
2.6.172	Yes	No	
2.6.174	Yes	No	
2.6.176	Yes	No	
2.6.178	Yes	No	
2.6.180	Yes	No	
2.6.182	Yes	No	
2.6.184	Yes	No	
2.6.186	Yes	No	
2.6.188	Yes	No	
2.6.190	Yes	No	
2.6.192	Yes	No	
2.6.194	Yes	No	
2.6.196	Yes	No	
2.6.198	Yes	No	
2.6.200	Yes	No	
2.6.202	Yes	No	
2.6.204	Yes	No	
2.6.206	Yes	No	
2.6.208	Yes	No	
2.6.210	Yes	No	
2.6.212	Yes	No	
2.6.214	Yes	No	
2.6.216	Yes	No	
2.6.218	Yes	No	
2.6.220	Yes	No	
2.6.222	Yes	No	
2.6.224	Yes	No	
2.6.226	Yes	No	
2.6.228	Yes	No	
2.6.230	Yes	No	
2.6.232	Yes	No	
2.6.234	Yes	No	
2.6.236	Yes	No	
2.6.238	Yes	No	
2.6.240	Yes	No	
2.6.242	Yes	No	
2.6.244	Yes	No	
2.6.246	Yes	No	
2.6.248	Yes	No	
2.6.250	Yes	No	
2.6.252	Yes	No	
2.6.254	Yes	No	
2.6.256	Yes	No	
2.6.258	Yes	No	
2.6.260	Yes	No	
2.6.262	Yes	No	
2.6.264	Yes	No	
2.6.266	Yes	No	
2.6.268	Yes	No	
2.6.270	Yes	No	
2.6.272	Yes	No	
2.6.274	Yes	No	
2.6.276	Yes	No	
2.6.278	Yes	No	
2.6.280	Yes	No	
2.6.282	Yes	No	
2.6.284	Yes	No	
2.6.286	Yes	No	
2.6.288	Yes	No	
2.6.290	Yes	No	
2.6.292	Yes	No	
2.6.294	Yes	No	
2.6.296	Yes	No	
2.6.298	Yes	No	
2.6.300	Yes	No	
2.6.302	Yes	No	
2.6.304	Yes	No	
2.6.306	Yes	No	
2.6.308	Yes	No	
2.6.310	Yes	No	
2.6.312	Yes	No	
2.6.314	Yes	No	
2.6.316	Yes	No	
2.6.318	Yes	No	
2.6.320	Yes	No	
2.6.322	Yes	No	
2.6.324	Yes	No	
2.6.326	Yes	No	
2.6.328	Yes	No	
2.6.330	Yes	No	
2.6.332	Yes	No	
2.6.334	Yes	No	
2.6.336	Yes	No	
2.6.338	Yes	No	
2.6.340	Yes	No	
2.6.342	Yes	No	
2.6.344	Yes	No	
2.6.346	Yes	No	
2.6.348	Yes	No	
2.6.350	Yes	No	
2.6.352	Yes	No	
2.6.354	Yes	No	
2.6.356	Yes	No	
2.6.358	Yes	No	
2.6.360	Yes	No	
2.6.362	Yes	No	
2.6.364	Yes	No	
2.6.366	Yes	No	
2.6.368	Yes	No	
2.6.370	Yes	No	
2.6.372	Yes	No	
2.6.374	Yes	No	
2.6.376	Yes	No	
2.6.378	Yes	No	
2.6.380	Yes	No	
2.6.382	Yes	No	
2.6.384	Yes	No	
2.6.386	Yes	No	
2.6.388	Yes	No	
2.6.390	Yes	No	
2.6.392	Yes	No	
2.6.394	Yes	No	
2.6.396	Yes	No	
2.6.398	Yes	No	
2.6.400	Yes	No	
2.6.402	Yes	No	
2.6.404	Yes	No	
2.6.406	Yes	No	
2.6.408	Yes	No	
2.6.410	Yes	No	
2.6.412	Yes	No	
2.6.414	Yes	No	
2.6.416	Yes	No	
2.6.418	Yes	No	
2.6.420	Yes	No	
2.6.422	Yes	No	
2.6.424	Yes	No	
2.6.426	Yes	No	
2.6.428	Yes	No	
2.6.430	Yes	No	
2.6.432	Yes	No	
2.6.434	Yes	No	
2.6.436	Yes	No	
2.6.438	Yes	No	
2.6.440	Yes	No	
2.6.442	Yes	No	
2.6.444	Yes	No	
2.6.446	Yes	No	
2.6.448	Yes	No	
2.6.450	Yes	No	
2.6.452	Yes	No	
2.6.454	Yes	No	
2.6.456	Yes	No	
2.6.458	Yes	No	
2.6.460	Yes	No	
2.6.462	Yes	No	
2.6.464	Yes	No	
2.6.466	Yes	No	
2.6.468	Yes	No	
2.6.470	Yes	No	
2.6.472	Yes	No	
2.6.474	Yes	No	
2.6.476	Yes	No	
2.6.478	Yes	No	
2.6.480	Yes	No	
2.6.482	Yes	No	
2.6.484	Yes	No	
2.6.486	Yes	No	
2.6.488	Yes	No	
2.6.490	Yes	No	
2.6.492	Yes	No	
2.6.494	Yes	No	
2.6.496	Yes	No	
2.6.498	Yes	No	
2.6.500	Yes	No	
2.6.502	Yes	No	
2.6.504	Yes	No	
2.6.506	Yes	No	
2.6.508	Yes	No	
2.6.510	Yes	No	
2.6.512	Yes	No	
2.6.514	Yes	No	
2.6.516	Yes	No	
2.6.518	Yes	No	
2.6.520	Yes	No	
2.6.522	Yes	No	
2.6.524	Yes	No	
2.6.526	Yes	No	
2.6.528	Yes	No	
2.6.530	Yes	No	
2.6.532	Yes	No	
2.6.534	Yes	No	
2.6.536	Yes	No	
2.6.538	Yes	No	
2.6.540	Yes	No	
2.6.542	Yes	No	
2.6.544	Yes	No	
2.6.546	Yes	No	
2.6.548	Yes	No	
2.6.550	Yes	No	
2.6.552	Yes	No	
2.6.554	Yes	No	
2.6.556	Yes	No	
2.6.558	Yes	No	
2.6.560	Yes	No	
2.6.562	Yes	No	
2.6.564	Yes	No	
2.6.566	Yes	No	
2.6.568	Yes	No	
2.6.570	Yes	No	
2.6.572	Yes	No	
2.6.574	Yes	No	
2.6.576	Yes	No	
2.6.578	Yes	No	
2.6.580	Yes	No	
2.6.582	Yes	No	
2.6.584	Yes	No	
2.6.586	Yes	No	
2.6.588	Yes	No	
2.6.590	Yes	No	
2.6.592	Yes	No	
2.6.594	Yes	No	
2.6.596	Yes	No	
2.6.598	Yes	No	
2.6.600	Yes	No	
2.6.602	Yes	No	
2.6.604	Yes	No	
2.6.606	Yes	No	
2.6.608	Yes	No	
2.6.610	Yes	No	
2.6.612	Yes	No	
2.6.614	Yes	No	
2.6.616	Yes	No	
2.6.618	Yes	No	
2.6.620	Yes	No	
2.6.622	Yes	No	
2.6.624	Yes	No	
2.6.626	Yes	No	
2.6.628	Yes	No	
2.6.630	Yes	No	
2.6.632	Yes	No	
2.6.634	Yes	No	
2.6.636	Yes	No	
2.6.638	Yes	No	
2.6.640	Yes	No	
2.6.642	Yes	No	
2.6.644	Yes	No	
2.6.646	Yes	No	
2.6.648	Yes	No	
2.6.650	Yes	No	
2.6.652	Yes	No	
2.6.654	Yes	No	
2.6.656	Yes	No	
2.6.658	Yes	No	
2.6.660	Yes	No	
2.6.662	Yes	No	
2.6.664	Yes	No	
2.6.666	Yes	No	
2.6.668	Yes	No	
2.6.670	Yes	No	
2.6.672	Yes	No	
2.6.674	Yes	No	
2.6.676	Yes	No	
2.6.678	Yes	No	
2.6.680	Yes	No	
2.6.682	Yes	No	
2.6.684	Yes	No	
2.6.686	Yes	No	
2.6.688	Yes	No	
2.6.690	Yes	No	
2.6.692	Yes	No	
2.6.694	Yes	No	
2.6.696	Yes	No	
2.6.698	Yes	No	
2.6.700	Yes	No	
2.6.702	Yes	No	
2.6.704	Yes	No	
2.6.706	Yes	No	
2.6.708	Yes	No	
2.6.710	Yes	No	
2.6.712	Yes	No	
2.6.714	Yes	No	
2.6.716	Yes	No	
2.6.718	Yes	No	
2.6.720	Yes	No	
2.6.722	Yes	No	
2.6.724	Yes	No	
2.6.726	Yes	No	
2.6.728	Yes	No	
2.6.730	Yes	No	
2.6.732	Yes	No	
2.6.734	Yes	No	
2.6.736	Yes	No	
2.6.738	Yes	No	
2.6.740	Yes	No	
2.6.742	Yes	No	
2.6.744	Yes	No	
2.6.746	Yes	No	
2.6.748	Yes	No	
2.6.750	Yes	No	
2.6.752	Yes	No	
2.6.754	Yes	No	
2.6.756	Yes	No	
2.6.758	Yes	No	
2.6.760	Yes	No	
2.6.762	Yes	No	
2.6.764	Yes	No	
2.6.766	Yes	No	
2.6.768	Yes	No	
2.6.770	Yes	No	
2.6.772	Yes	No	

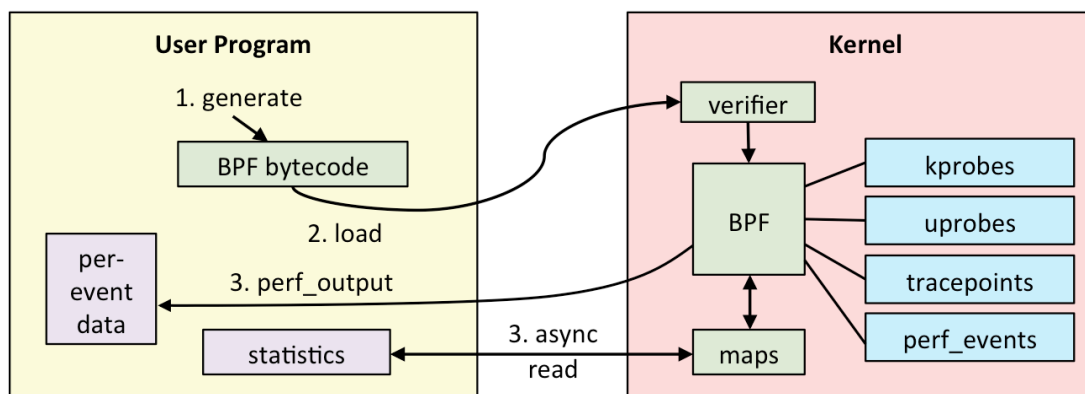


图 5 eBPF 观测架构

eBPF 分为用户空间程序和内核程序两部分：

- 用户空间程序负责加载 BPF 字节码至内核，如需要也会负责读取内核回传的统计信息或者事件详情；
- 内核中的 BPF 字节码负责在内核中执行特定事件，如需要也会将执行的结果通过 maps 或者 perf-event 事件发送至用户空间；

其中用户空间程序与内核 BPF 字节码程序可以使用 map 结构实现双向通信，这为内核中运行的 BPF 字节码程序提供了更加灵活的控制。

用户空间程序与内核中的 BPF 字节码交互的流程主要如下：

1. 我们可以使用 LLVM 或者 GCC 工具将编写的 BPF 代码程序编译成 BPF 字节码；
2. 然后使用加载程序 Loader 将字节码加载至内核；内核使用验证器（verifier）组件保证执行字节码的安全性，以避免对内核造成灾难，在确认字节码安全后将其加载对应的内核模块执行；BPF 观测技术相关的程序类型可能是 kprobes/uprobes/tracepoint/perf\_events 中的一个或多个，其中：
  - **kprobes**：实现内核中动态跟踪。kprobes 可以跟踪到 Linux 内核中的函数入口或返回点，但是不是稳定 ABI 接口，可能会因为内核版本变化导致，导致跟踪失效。
  - **uprobes**：用户级别的动态跟踪。与 kprobes 类似，只是跟踪的函数为用户程序中的函数。
  - **tracepoints**：内核中静态跟踪。tracepoints 是内核开发人员维护的跟踪点，能够提供稳定的 ABI 接口，但是由于是研发人员维护，数量和场景可能受限。
  - **perf\_events**：定时采样和 PMC。
3. 内核中运行的 BPF 字节码程序可以使用两种方式将测量数据回传至用户空间
  - **maps** 方式可用于将内核中实现的统计摘要信息（比如测量延迟、堆栈信息）等回传至用户空间；
  - **perf-event** 用于将内核采集的事件实时发送至用户空间，用户空间程序实时读取分析；

如无特殊说明，本文中所说的 BPF 都是泛指 BPF 技术。

## 2.3 eBPF 的限制

eBPF 技术虽然强大，但是为了保证内核的处理安全和及时响应，内核中的 eBPF 技术也给予了诸多限制，当然随着技术的发展和演进，限制也在逐步放宽或者提供了对应的解决方案。



- eBPF 程序不能调用任意的内核参数，只限于内核模块中列出的 BPF Helper 函数，函数支持列表也随着内核的演进在不断增加；（添加个数说明）
- eBPF 程序中循环次数限制，这主要是用来防止在 kprobes 中插入任意的循环，导致锁住整个系统；解决办法包括展开循环，并为需要循环的常见用途添加辅助函数。Linux 5.3 在 BPF 中包含了对有界循环的支持，它有一个可验证的运行时间上限；
- eBPF 堆栈大小被限制在 MAX\_BPF\_STACK，截止到内核 Linux 5.8 版本，被设置为 512；参见 [include/linux/filter.h](#)，这个限制特别是在栈上存储多个字符串缓冲区时：一个 char[256] 缓冲区会消耗这个栈的一半。目前没有计划增加这个限制，解决方法是改用 bpf 映射存储，它实际上是无限的。

```
1  /* BPF program can access up to 512 bytes of stack space. */
2  #define MAX_BPF_STACK 512
```

- eBPF 字节码大小最初被限制为 4096 条指令，截止到内核 Linux 5.8 版本，当前已将放宽至 100 万指令（BPF\_COMPLEXITY\_LIMIT\_INSNS），参见：[include/linux/bpf.h](#)，对于无权限的 BPF 程序，仍然保留 4096 条限制（BPF\_MAXINSNS）；新版本的 eBPF 也支持了多个 eBPF 程序级联调用，虽然传递信息存在某些限制，但是可以通过组合实现更加强大的功能。

```
1  #define BPF_COMPLEXITY_LIMIT_INSNS      1000000 /* yes. 1M insns */
```

## 2.4 eBPF 与内核模块对比

在 Linux 观测方面，eBPF 总是会拿来与 kernel 模块方式进行对比，eBPF 在安全性、入门门槛上比内核模块都有优势，这两点在观测场景下对于用户来讲尤其重要。

维度	Linux 内核模块	eBPF
kprobes/tracepoints	支持	支持
安全性	可能引入安全漏洞或导致内核 Panic	通过验证器进行检查，可以保障内核安全
编译性	需要编译内核	不需要编译内核，引入头文件即可
运行	基于相同内核运行	基于稳定 ABI 的 BPF 程序可以编译一次，各处运行
与应用程序交互	打印日志或文件	通过基于 perf_event map 结构
数据结构丰富性	一般	丰富
入门门槛	高	低

表格 2 eBPF 与 Linux 内核模块方式对比

## 3. 应用案例



大名鼎鼎的性能分析大师 Brendan Gregg 等编写了诸多的 BCC 或 BPFTrace 的工具集可以拿来直接使用，完全可以满足我们日常问题分析和排查。

BCC 在 CentOS 7 系统中可以通过 yum 快速安装

```
1 # yum install bcc -y
2 Resolving Dependencies
3 --> Running transaction check
4 ---> Package bcc.x86_64 0:0.8.0-1.el7 will be updated
5 --> Processing Dependency: bcc(x86-64) = 0.8.0-1.el7 for package: python-
    bcc-0.8.0-1.el7.x86_64
6 ---> Package bcc.x86_64 0:0.10.0-1.el7 will be an update
7 --> Processing Dependency: bcc-tools = 0.10.0-1.el7 for package: bcc-
    0.10.0-1.el7.x86_64
8 --> Running transaction check
9 ---> Package bcc-tools.x86_64 0:0.8.0-1.el7 will be updated
10 ---> Package bcc-tools.x86_64 0:0.10.0-1.el7 will be an update
11 ---> Package python-bcc.x86_64 0:0.8.0-1.el7 will be updated
12 ---> Package python-bcc.x86_64 0:0.10.0-1.el7 will be an update
13 --> Finished Dependency Resolution
14 ...
```

其他系统的安装方式参见：[INSTALL.md](#)

BCC 中每一个工具都有一个对应的使用样例，比如 [execsnoop.py](#) 和 [execsnoop\\_example.txt](#)，在使用样例中有详细的使用说明，而且 BCC 中的工具使用的帮助文档格式基本类似，上手非常方便。

BCC 的程序一般情况下都需要 root 用户来运行。

## 3.1 Linux 性能分析 60 秒（BPF版本）

英文原文 [Linux Performance Analysis in 60,000 Milliseconds](#), [视频地址](#)

```
1 uptime
2 dmesg | tail
3 vmstat 1
4 mpstat -P ALL 1
5 pidstat 1
6 iostat -xz 1
7 free -m
8 sar -n DEV 1
9 sar -n TCP,ETCP 1
10 top
```

60s 系列 BPF 版本如下：

1. <b>execsnoop</b>	filetop filelife fileslower vfscount vfststat cachestat cachetop	opensnoop statsnoop syncsnoop	c* java* node* php* python* ruby*	mysqld_qslower bashreadline ucalls uflow uac uobinew	gethostlatency memleak sslsniff	Other: capable syscount killsnoop
---------------------	---	-------------------------------------	---	---	---------------------------------------	--

图 6 60s 排查之 BPF 版本

对于在系统中运行的 "闪电侠" 程序，运行周期非常短，但是可能会带来系统的抖动延时，我们采用 `top` 命令查看一般情况下难以发现，我们可以使用 BCC 提供的工具 `execsnoop` 来进行排查：

```

1  # Trace file opens with process and filename: opensnoop
2  #/usr/share/bcc/tools/execsnoop
3  PCOMM          PID      PPID      RET  ARGS
4  sleep          3334    21029     0   /usr/bin/sleep 3
5  sleep          3339    21029     0   /usr/bin/sleep 3
6  conntrack      3341    1112     0   /usr/sbin/conntrack --stats
7  conntrack      3342    1112     0   /usr/sbin/conntrack --count
8  sleep          3344    21029     0   /usr/bin/sleep 3
9  iptables-save  3347    9211     0   /sbin/iptables-save -t filter
10 iptables-save  3348    9211     0   /sbin/iptables-save -t nat

```

### 3.2 slab dentry 过大导致的网络抖动排查

#### 现象

网络 ping 的延时间歇性有规律出现抖动

#### 问题排查

采用 `execsnoop` 分析发现，某个运行命令 `cat /proc/slabinfo` 的运行时间间隔与抖动的频率完全吻合，顺着这个的线索定位，我们发现云厂商提供的 Java 版本的云监控会定期调用 `cat /proc/slabinfo` 来获取内核缓存的信息；

通过命令 `slabtop` 发现系统中的 `dentry` 项的内存占用非常大，系统内存 128G，`dentry` 占用 70G 以上，所以问题很快就定位到是系统在打开文件方面可能有相关问题；

#### 根因分析

我们使用对于打开文件跟踪的 BCC 工具 `opensnoop` 很快就定位到是某个程序频繁创建和删除临时文件，最终定位为某个 PHP 程序设置的调用方式存在问题，导致每次请求会创建和删除临时文件；代码中将 http 调用中的 `contentType` 设置成了 `Http::CONTENT_TYPE_UPLOAD`，导致每次请求都会生成临时文件，修改成 `application/x-www-form-urlencoded` 问题解决。

问题的原理可参考 [记一次对网络抖动经典案例的分析](#) 和 [systemtap脚本分析系统中dentry SLAB占用过高问题](#)

## 3.3 生成火焰图

火焰图是帮助我们对系统耗时进行可视化的图表，能够对程序中那些代码经常被执行给出一个清晰的展现。Brendan Gregg 是火焰图的创建者，他在 [GitHub](#) 上维护了一组脚本可以轻松生成需要的可视化格式数据。使用 BCC 中的工具 `profile` 可很方面地收集道 CPU 路径的数据，基于数据采用工具可以轻松生成火焰图，查找到程序的性能瓶颈。

使用 `profile` 搜集火焰图的程序没有任何限制和改造

`profile` 工具可以让我们轻松对于系统或者程序的 CPU 性能路径进行可视化分析：

```
1 /usr/share/bcc/tools/profile -h
2 usage: profile [-h] [-p PID | -L TID] [-U | -K] [-F FREQUENCY | -c COUNT]
   [-d]
3           [-a] [-I] [-f] [--stack-storage-size STACK_STORAGE_SIZE]
4           [-C CPU]
5           [duration]
6
7 Profile CPU stack traces at a timed interval
8
9 positional arguments:
10   duration              duration of trace, in seconds
11
12 optional arguments:
13   -h, --help            show this help message and exit
14   -p PID, --pid PID     profile process with this PID only
15   -L TID, --tid TID     profile thread with this TID only
16   -U, --user-stacks-only
17                       show stacks from user space only (no kernel space
18                       stacks)
19   -K, --kernel-stacks-only
20                       show stacks from kernel space only (no user space
21                       stacks)
22   -F FREQUENCY, --frequency FREQUENCY
23                       sample frequency, Hertz
24   -c COUNT, --count COUNT
25                       sample period, number of events
26   -d, --delimited       insert delimiter between kernel/user stacks
27   -a, --annotations     add _[k] annotations to kernel frames
28   -I, --include-idle    include CPU idle stacks
29   -f, --folded          output folded format, one line per stack (for
   flame
```

```

30             graphs)
31     --stack-storage-size STACK_STORAGE_SIZE
32             the number of unique stack traces that can be
stored
33             and displayed (default 16384)
34     -C CPU, --cpu CPU      cpu number to run profile on
35
36 examples:
37     ./profile              # profile stack traces at 49 Hertz until Ctrl-C
38     ./profile -F 99        # profile stack traces at 99 Hertz
39     ./profile -c 1000000    # profile stack traces every 1 in a million
events
40     ./profile 5            # profile at 49 Hertz for 5 seconds only
41     ./profile -f 5         # output in folded format for flame graphs
42     ./profile -p 185       # only profile process with PID 185
43     ./profile -L 185       # only profile thread with TID 185
44     ./profile -U           # only show user space stacks (no kernel)
45     ./profile -K           # only show kernel space stacks (no user)

```

`profile` 配合 `FlameGraph` 可以轻松帮我们绘制出 CPU 使用的火焰图。

```

1 $ profile -af 30 > out.stacks01
2 $ git clone https://github.com/brendangregg/FlameGraph
3 $ cd FlameGraph
4 $ ./flamegraph.pl --color=java < ../out.stacks01 > out.svg

```

图 7 火焰图

### 3.3 排查网络调用来源

在生产场景下，会有些特定场景需要抓取连接到外网特定地址的程序，这时候我们可以采用 BCC 工具集中的 `tcplife` 来定位。

```
1 /usr/share/bcc/tools/tcplife -h
2 usage: tcplife [-h] [-T] [-t] [-w] [-s] [-p PID] [-L LOCALPORT]
3             [-D REMOTEPORT]
4
5 Trace the lifespan of TCP sessions and summarize
6
7 optional arguments:
8   -h, --help            show this help message and exit
9   -T, --time            include time column on output (HH:MM:SS)
10  -t, --timestamp       include timestamp on output (seconds)
11  -w, --wide            wide column output (fits IPv6 addresses)
12  -s, --csv             comma separated values output
13  -p PID, --pid PID     trace this PID only
14  -L LOCALPORT, --localport LOCALPORT
15                        comma-separated list of local ports to trace.
16  -D REMOTEPORT, --remoteport REMOTEPORT
17                        comma-separated list of remote ports to trace.
18
19 examples:
20   ./tcplife            # trace all TCP connect()s
21   ./tcplife -t         # include time column (HH:MM:SS)
22   ./tcplife -w         # wider columns (fit IPv6)
23   ./tcplife -stT       # csv output, with times & timestamps
24   ./tcplife -p 181     # only trace PID 181
25   ./tcplife -L 80      # only trace local port 80
26   ./tcplife -L 80,81   # only trace local ports 80 and 81
27   ./tcplife -D 80      # only trace remote port 80
```

通过在机器上使用 `tcplife` 来获取的网络连接信息，我们可以看到包括了 PID、COMM、本地 IP 地址、本地端口、远程 IP 地址和远程端口，通过这些信息非常方便排查到连接到特定 IP 地址的程序，尤其是连接的过程非常短暂，通过 `netstat` 等其他工具不容易排查的场景。

```

1 # /usr/share/bcc/tools/tcplife
2 PID      COMM          IP LADDR          LPORT RADDR
   RPORT  TX_KB  RX_KB  MS
3 1776  blackbox_export  4  169.254.20.10  35830 169.254.20.10
   53      0      0 0.36
4 27150 node-cache        4  169.254.20.10  53     169.254.20.10
   35830   0      0 0.36
5 12511 coredns           4  127.0.0.1      58492 127.0.0.1
   8080    0      0 0.32
6 ...

```

如果我们想知道更加详细的 TCP 状态情况，那么 `tcptracer` 可展示更加详细的 TCP 状态，其中 C 代表 Connect X 表示关闭，A 代表 Accept。

```

1 # /usr/share/bcc/tools/tcptracer
2 Tracing TCP established connections. Ctrl-C to end.
3 T  PID      COMM          IP SADDR          DADDR          SPORT
   DPORT
4 C  21066  ilogtail        4  10.81.128.12   100.100.49.128  40906  80
5 X  21066  ilogtail        4  10.81.128.12   100.100.49.128  40906  80
6 C  21066  ilogtail        4  10.81.128.12   100.100.49.128  40908  80
7 X  21066  ilogtail        4  10.81.128.12   100.100.49.128  40908  80

```

`tcpstates` 还能够展示出来 TCP 状态机的流转情况：

```

1 # /usr/share/bcc/tools/tcpstates
2 SKADDR      C-PID C-COMM      LADDR          LPORT RADDR
   RPORT OLDSTATE  -> NEWSTATE  MS
3 ffff9fd7e8192000 22384 curl        100.66.100.185  0      52.33.159.26  80
   CLOSE      -> SYN_SENT  0.000
4 ffff9fd7e8192000 0      swapper/5   100.66.100.185  63446  52.33.159.26  80
   SYN_SENT   -> ESTABLISHED 1.373
5 ffff9fd7e8192000 22384 curl        100.66.100.185  63446  52.33.159.26  80
   ESTABLISHED -> FIN_WAIT1  176.042

```

同样，我们也可以实时获取到 TCP 连接超时或者重连的网络连接；也可以通过抓取 UDP 包相关的连接信息，用于定位诸如 DNS 请求超时或者 DNS 请求的发起进程。

## 4. 编写 BPF 程序

对于大多数开发者而言，更多的是基于 BPF 技术之上编写解决我们日常遇到的各种问题，当前 BCC 和 BPFTrace 两个项目在观测和性能分析上已经有了诸多灵活且功能强大的工具箱，完全可以满足我们日常使用。

- [BCC](#) 提供了更高阶的抽象，可以让用户采用 Python、C++ 和 Lua 等高级语言快速开发 BPF 程序；
- [BPFTrace](#) 采用类似于 awk 语言快速编写 eBPF 程序；

更早期的工具则是使用 C 语言来编写 BPF 程序，使用 LLVM clang 编译成 BPF 代码，这对于普通使用者上手有不少门槛当前仅限于对于 eBPF 技术更加深入的学习场景。

## 4.1 BCC 版本 HelloWorld

图 8 BCC 整体架构

使用 BCC 前端绑定语言 Python 编写的 Hello World 版本：

```
1  #!/usr/bin/python3
2
3  from bcc import BPF
4
5  # This may not work for 4.17 on x64, you need replace kprobe__sys_clone
   with kprobe__x64_sys_clone
6  prog = """
7      int kprobe__sys_clone(void *ctx) {
8          bpf_trace_printk("Hello, World!\\n");
9          return 0;
10     }
11     """
12
13     b = BPF(text=prog, debug=0x04)
14     b.trace_print()
```

运行程序前需要安装过 bcc 相关工具包，当运行正常的时候我们发现每当 `sys_clone` 系统调用时，运行的控制台上就会打印 “Hello, World!”，在打印文字前面还包含了调用程序的进程名称，进程 ID 等信息；

如果运行报错，可能是缺少头文件，一般安装 kernel-devel 包即可。



```

1 # python ./hello.py
2         kubelet-8349  [006] d... 33637334.829981: : Hello, World!
3         kubelet-8349  [006] d... 33637334.838594: : Hello, World!
4         kubelet-8349  [006] d... 33637334.843788: : Hello, World!

```

## 4.3 BPFTrace

BPFTrace 是基于 BPF 和 BCC 的开源项目，与 BCC 不同的是其提供了更高层次的抽象，可以使用类似 AWK 脚本语言来编写基于 BPF 的跟踪或者性能排查工具，更加易于入门和编写，该工具的主要灵感来自于 Solaris 的 D 语言。BPFTrace 更方便与编写单行的程序。BPFTrace 与 BCC 一样也是 IO Visor 组织下的项目，仓库参见 [bpftrace](#)。更加深入的学习资料参见：[Reference Guide](#) 和 [One-Liner Tutorial](#)。

BPFTrace 使用 LLVM 将脚本编译成 BPF 二进制码，后续使用 BCC 与 Linux 内核进行交互。从功能层面上讲，BPFTrace 的定制性和灵活性不如 BCC，但是比 BCC 工具更加易于理解和使用，降低了 BPF 技术的使用门槛。

使用样例：

```

1 # 统计进程调用 sys_enter 的次数
2 #bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
3 Attaching 1 probe...
4 ^C
5
6 @[bpftrace]: 6
7 @[systemd]: 24
8 @[snmp-pass]: 96
9 @[sshd]: 125
10
11 # 统计内核中函数堆栈的次数
12 # bpftrace -e 'profile:hz:99 { @[kstack] = count(); }'
13 Attaching 1 probe...
14 ^C
15
16 [...]
17 @[
18 filemap_map_pages+181
19 __handle_mm_fault+2905
20 handle_mm_fault+250
21 __do_page_fault+599
22 async_page_fault+69
23 ]: 12
24 [...]
25 @[
26 cpuidle_enter_state+164
27 do_idle+390
28 cpu_startup_entry+111

```

```

29 start_secondary+423
30 secondary_startup_64+165
31 ]: 22122

```

## 4.3 C 语言原生方式

采用 LLVM Clang 的方式编译会涉及到内核编译环境搭建，而且还需要自己编译 Makefile 等操作，属于高级用户使用：

bpf\_program.c

```

1  #include <linux/bpf.h>
2  #define SEC(NAME) __attribute__((section(NAME), used))
3
4  static int (*bpf_trace_printk)(const char *fmt, int fmt_size,
5                                ...) = (void *)BPF_FUNC_trace_printk;
6
7  SEC("tracepoint/syscalls/sys_enter_execve")
8  int bpf_prog(void *ctx) {
9      char msg[] = "Hello, BPF World!";
10     bpf_trace_printk(msg, sizeof(msg));
11     return 0;
12 }
13
14 char _license[] SEC("license") = "GPL";

```

loader.c

```

1  #include "bpf_load.h"
2  #include <stdio.h>
3
4  int main(int argc, char **argv) {
5      if (load_bpf_file("bpf_program.o") != 0) {
6          printf("The kernel didn't load the BPF program\n");
7          return -1;
8      }
9
10     read_trace_pipe();
11
12     return 0;
13 }

```

Makefile 文件（部分）

```

1  build: ${BPFCODE.c} ${BPFLOADER}
2      $(CLANG) -O2 -target bpf -c $(BPFCODE:=.c) $(CCINCLUDE) -o ${BPFCODE:=.o}

```

其中 clang 编译中的选型 `-target bpf` 表明我们将代码编译成 bpf 的字节码。

完整的程序参见：[hello\\_world](#)；更多的样例代码可以参见对应内核中 `kernel-src/samples/bpf/` 下的样例代码。

## 5. 参考资料

---

1. [The BSD Packet Filter: A New Architecture for User-level Packet Capture](#)
2. [\[译\] Cilium: BPF 和 XDP 参考指南 \(2019\) Cillum BPF and XDP Reference Guide](#)
3. [Cloudflare架构以及BPF如何占据世界](#)
4. [關於 BPF 和 eBPF 的筆記](#)
5. [Dive into BPF: a list of reading material 中文](#)
6. [eBPF 简史](#)
7. <https://www.youtube.com/watch?v=znBGt7oHJyQ>
8. [BPF Documentation HOWTO interact with BPF subsystem](#)
9. [Linux 内核 BPF 文档](#)
10. [Linux Extended BPF \(eBPF\) Tracing Tools](#) Brendan Gregg
11. [性能提升40%: 腾讯 TKE 用 eBPF绕过 conntrack 优化K8s Service](#)
12. [SDN handbook](#)