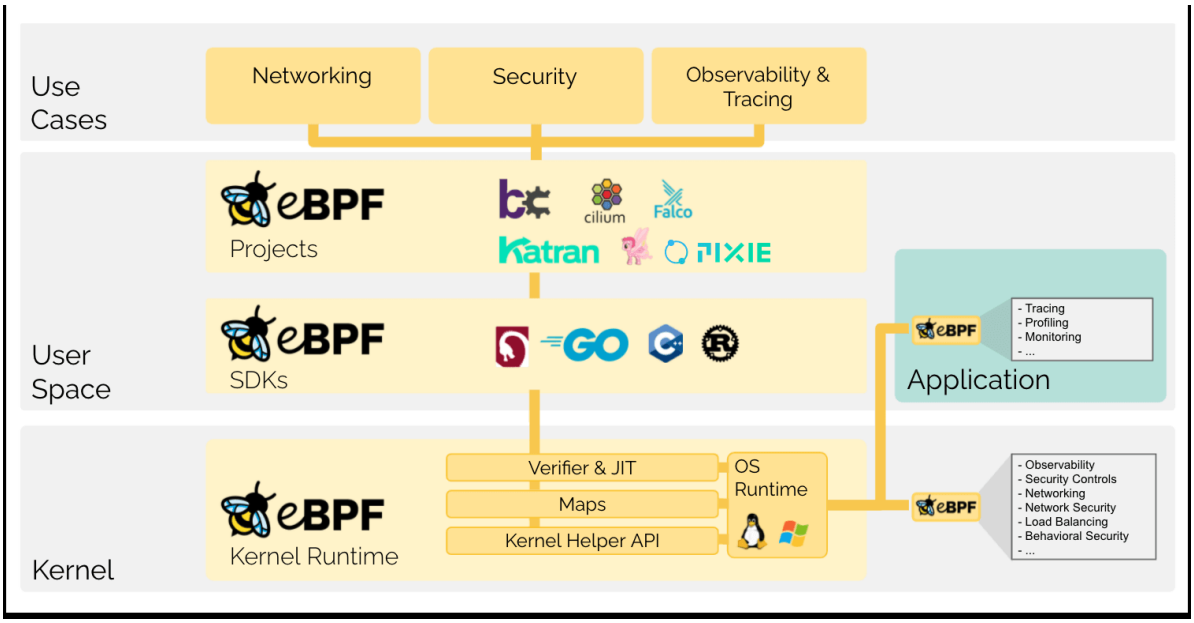
What is eBPF?

eBPF 是一项革命性的技术，起源于 Linux 内核，可以在特权上下文（如操作系统内核）中运行沙盒程序。它用于安全有效地扩展内核的功能，而无需更改内核源代码或加载内核模块。
从历史上看，由于内核具有监督和控制整个系统的特权能力，操作系统一直是实现可观察性、安全性和网络功能的理想场所。同时，操作系统内核由于其核心作用和对稳定性和安全性的高要求而难以发展。因此，与在操作系统之外实现的功能相比，操作系统级别的创新率传统上较低。



eBPF从根本上改变了这一准则。通过允许在操作系统中运行沙盒程序，应用程序开发人员可以运行 eBPF 程序，以便在运行时向操作系统添加其他功能。然后，操作系统保证安全性和执行效率，就像在实时 （JIT） 编译器和验证引擎的帮助下进行本机编译一样。这导致了一波基于 eBPF 的项目，涵盖了广泛的用例，包括下一代网络、可观测性和安全功能。

如今，eBPF 被广泛用于驱动各种用例：在现代数据中心和云原生环境中提供高性能网络和负载平衡，以低开销提取细粒度的安全可观测性数据，帮助应用程序开发人员跟踪应用程序，为性能故障排除、预防性应用程序和容器运行时安全实施提供见解等等。可能性是无穷无尽的，eBPF正在解锁的创新才刚刚开始。

What Is eBPF.io

eBPF.io 是每个人就eBPF主题进行学习和协作的地方。eBPF是一个开放的社区，每个人都可以参与和分享。无论您是想阅读eBPF的初步介绍，查找进一步的阅读材料，还是迈出成为主要eBPF项目贡献者的第一步，eBPF.io 都会在此过程中为您提供帮助。

What do eBPF and BPF stand for?

BPF最初代表Berkeley Packet Filter，但现在eBPF（扩展BPF）可以做的不仅仅是数据包过滤，首字母缩略词不再有意义。eBPF现在被认为是一个独立的术语，不代表任何东西。在 Linux 源代码中，术语 BPF 仍然存在，在工具和文档中，术语 BPF 和 eBPF 通常可以互换使用。原始BPF有时被称为cBPF（经典BPF），以区别于eBPF。

What is the bee named?

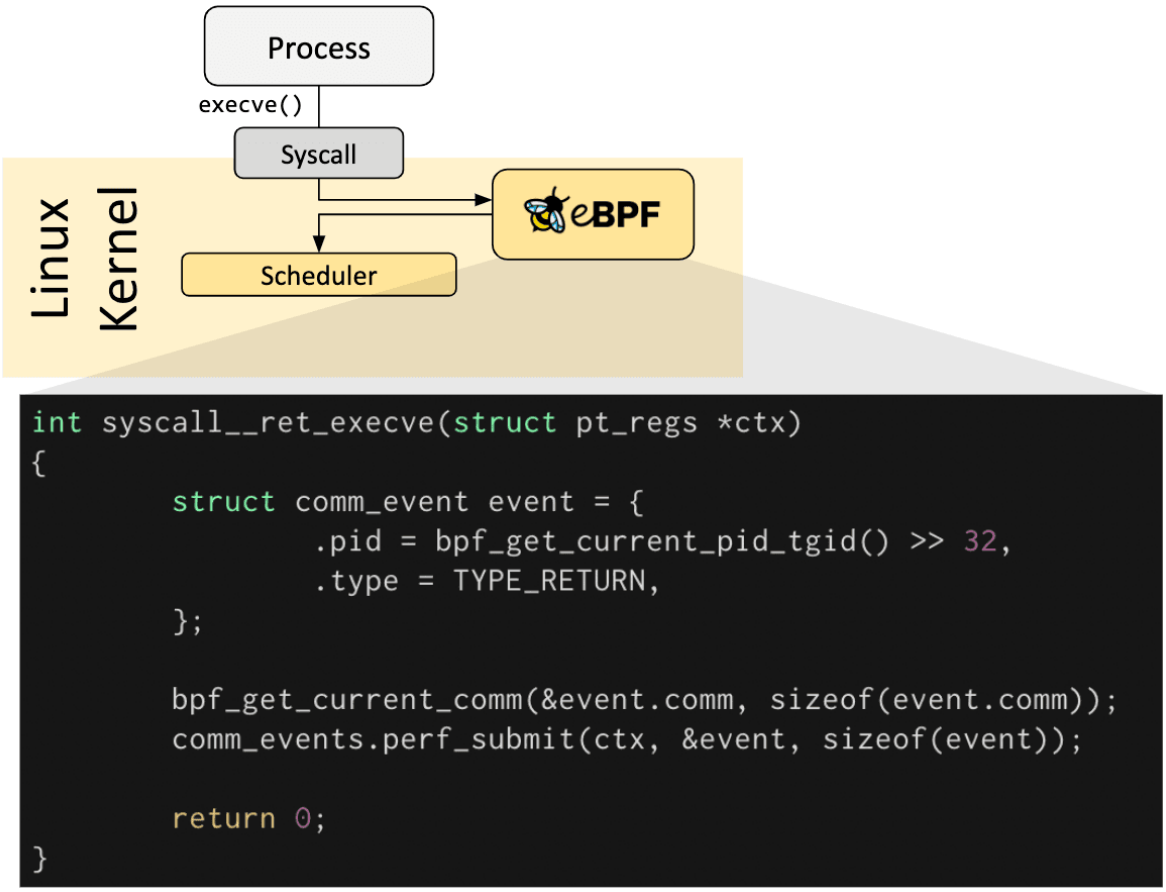
bee是eBPF的官方标志，最初由Vadim Shchekoldin创建。在第一届eBPF峰会上，进行了投票，bee被命名为eBee。（有关徽标的可接受使用的详细信息，请参阅 Linux 基金会品牌指南。

Introduction to eBPF

以下章节是对eBPF的快速介绍。如果您想了解有关 eBPF 的更多信息，请参阅 eBPF 和 XDP 参考指南。无论您是希望构建 eBPF 程序的开发人员，还是对利用使用 eBPF 的解决方案感兴趣，了解基本概念和体系结构都很有用。

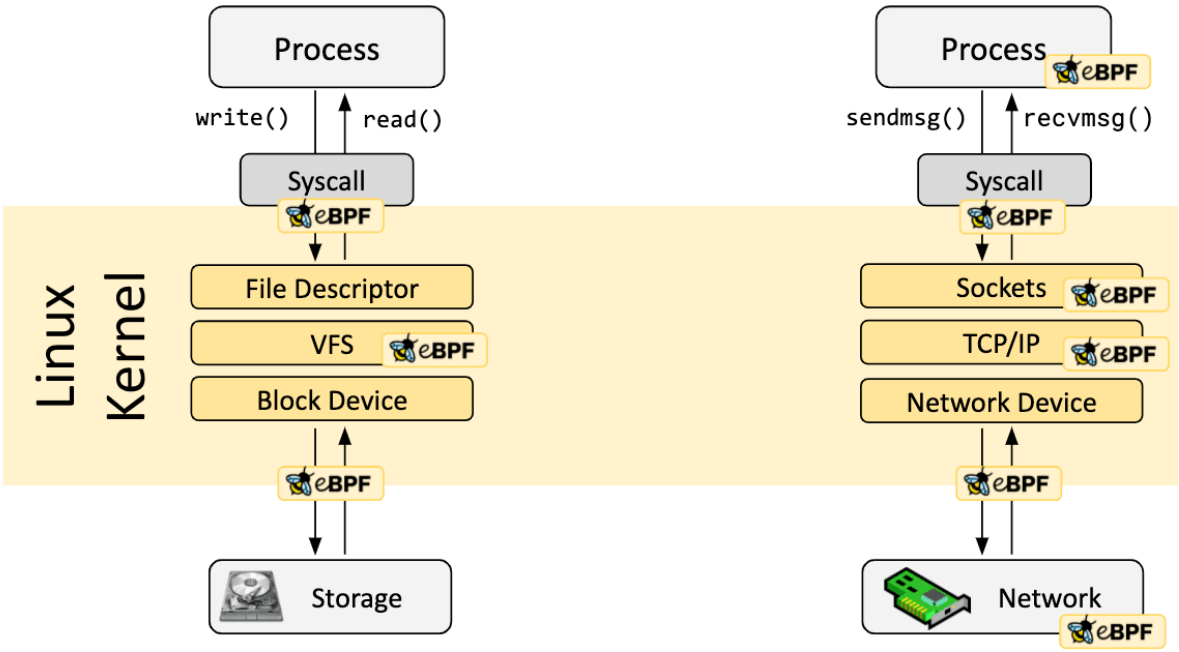
Hook Overview

eBPF 程序是事件驱动的，在内核或应用程序通过某个钩点时运行。预定义的钩子包括系统调用、函数进入/退出、内核跟踪点、网络事件和其他几个钩子。



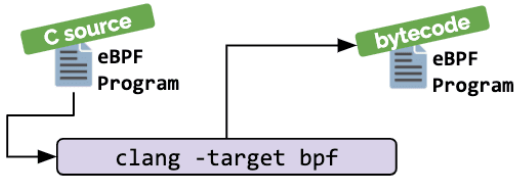

```
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

如果不存在满足特定需求的预定义钩子，则可以创建内核探测（kprobe）或用户探测（uprobe）来附加内核或用户应用程序中几乎任何位置的eBPF程序。



How are eBPF programs written?

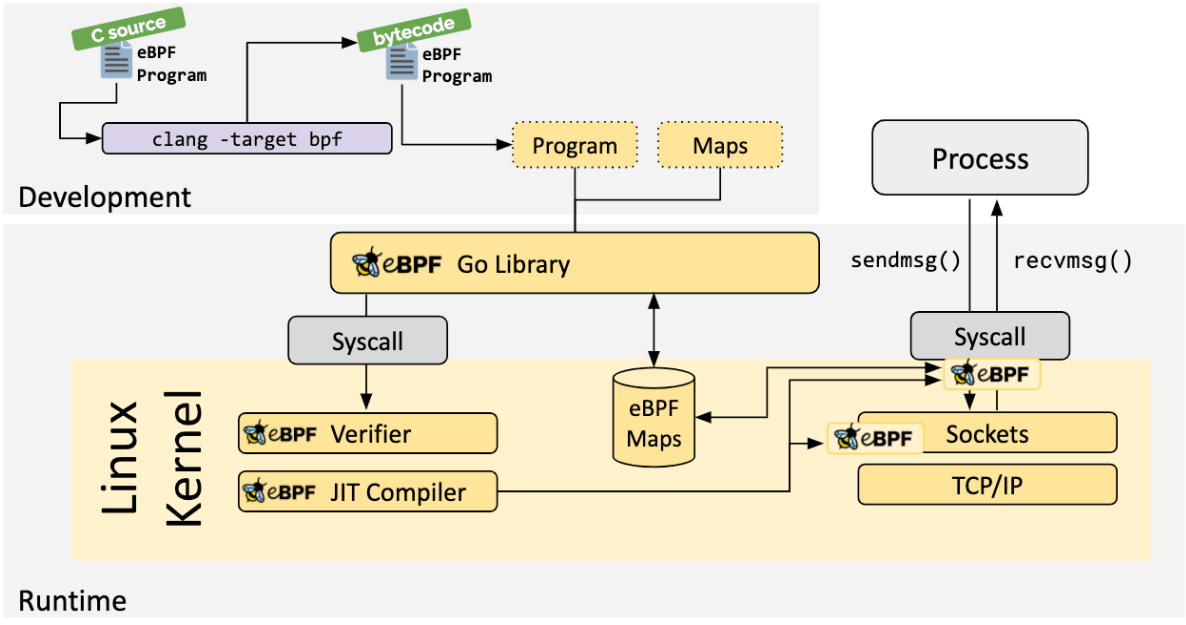
在很多场景中，eBPF不是直接使用，而是通过Cilium，bcc或bpftrace等项目间接使用，这些项目在eBPF之上提供抽象，不需要直接编写程序，而是提供指定基于意图的定义的能力，然后用eBPF实现。



如果不存在更高级别的抽象，则需要直接编写程序。Linux 内核期望 eBPF 程序以字节码的形式加载。虽然可以直接编写字节码，但更常见的开发实践是利用像LLVM这样的编译器套件将伪C代码编译为eBPF字节码。

Loader & Verification Architecture

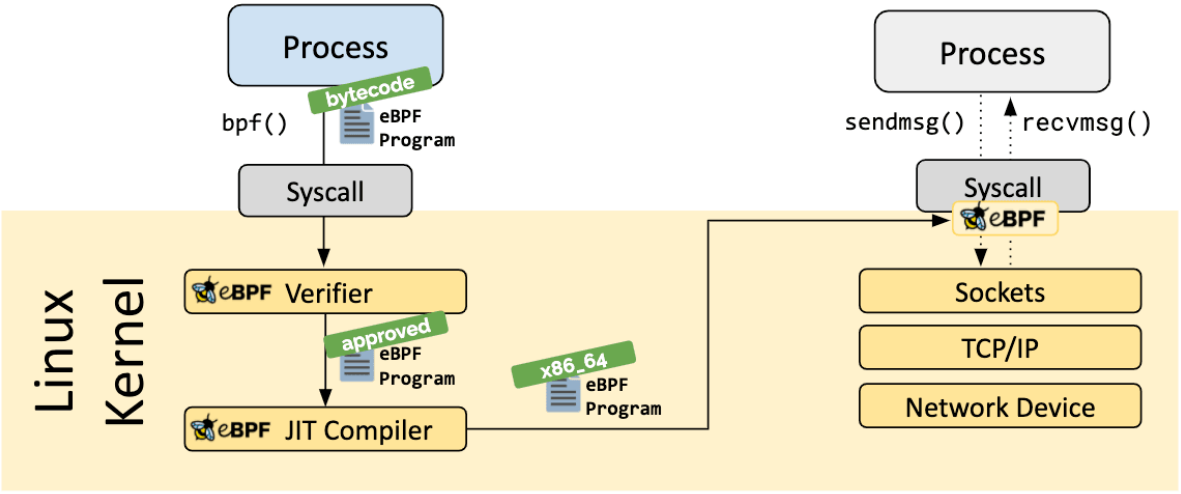
当确定了所需的钩子后，可以使用 bpf 系统调用将 eBPF 程序加载到 Linux 内核中。这通常使用可用的 eBPF 库之一来完成。下一节将介绍可用的开发工具链。



当程序加载到 Linux 内核中时，它会在附加到请求的钩子之前经过两个步骤：

Verification

验证步骤可确保 eBPF 程序可以安全运行。它验证程序是否满足多个条件，例如：



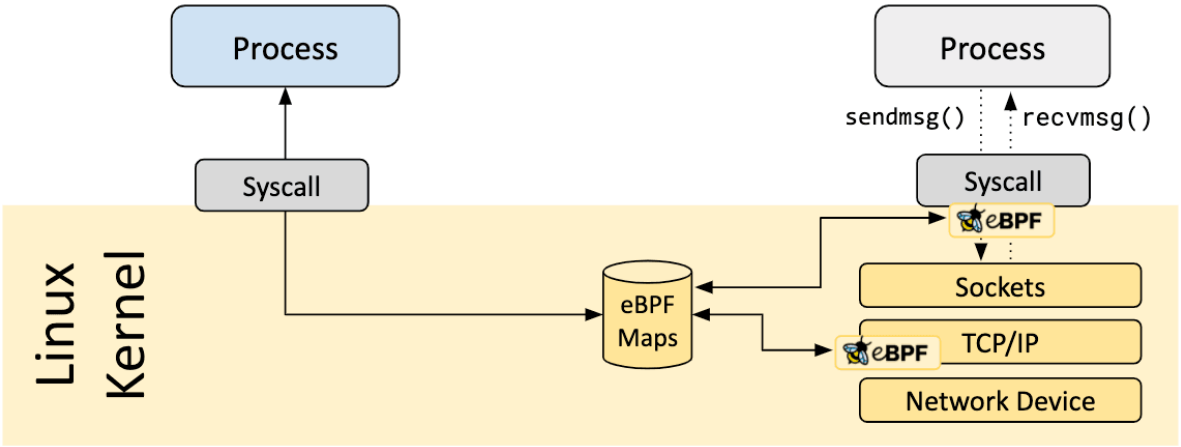
- 加载 eBPF 程序的进程具有所需的功能（权限）。除非启用了非特权 eBPF，否则只有特权进程才能加载 eBPF 程序。
- 该程序不会崩溃或以其他方式损坏系统。
- 程序总是运行到完成（即程序不会永远处于循环中，从而阻止进一步的处理）。

JIT Compilation

即时 （JIT） 编译步骤将程序的通用字节码转换为特定于机器的指令集，以优化程序的执行速度。这使得 eBPF 程序的运行效率与本机编译的内核代码或作为内核模块加载的代码一样高效。

Maps

eBPF 程序的一个重要方面是共享收集的信息和存储状态的能力。为此，eBPF程序可以利用eBPF maps的概念在广泛的数据结构中存储和检索数据。eBPF maps可以从 eBPF 程序访问，也可以通过系统调用从用户空间中的应用程序访问。
A vital aspect of eBPF programs is the ability to share collected information and to store state. For this purpose, eBPF programs can leverage the concept of eBPF maps to store and retrieve data in a wide set of data structures. eBPF maps can be accessed from eBPF programs as well as from applications in user space via a system call.



以下是支持的eBPF maps类型的不完整列表，以便了解数据结构的多样性。对于各种eBPF maps类型，共享和每 CPU 变体都可用。(For various map types, both a shared and a per-CPU variation is available.)

- 哈希表、数组
- LRU（最近最少使用）
- 环形缓冲器
- 堆栈跟踪
- LPM（最长前缀匹配）

Helper Calls

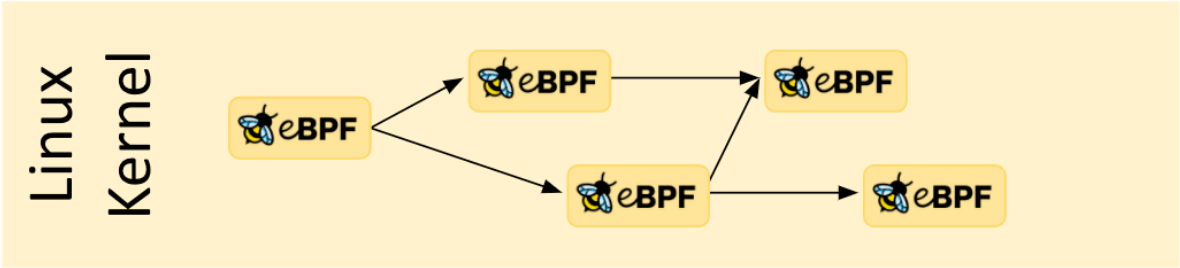
eBPF 程序不能调用任意内核函数。允许这样做会将 eBPF 程序绑定到特定的内核版本，并使程序的兼容性复杂化。相反，eBPF 程序可以调用helper functions，这是内核提供的众所周知且稳定的 API。

可用的帮助程序调用集在不断发展。可用的帮助程序调用示例：

- 生成随机数
- 获取当前时间和日期
- eBPF map访问
- Get process/cgroup context
- 操作网络数据包和转发逻辑

Tail & Function Calls

eBPF 程序可以与tail and function calls的概念组合。Function calls允许在 eBPF 程序中定义和调用函数。Tail calls可以调用和执行另一个 eBPF 程序并替换执行上下文，类似于 execve（） 系统调用对常规进程的操作方式。



eBPF Safety

权力越大，责任越大。

eBPF是一项非常强大的技术，现在运行在许多关键软件基础设施组件的核心。在 eBPF 的开发过程中，当 eBPF 被考虑纳入 Linux 内核时，eBPF 的安全性是最关键的方面。eBPF的安全性通过几层确保：

Required Privileges

除非启用了非特权 eBPF，否则所有打算将 eBPF 程序加载到 Linux 内核的进程都必须在特权模式（root） 下运行或需要该功能CAP_BPF。这意味着不受信任的程序无法加载 eBPF 程序。

如果启用了非特权 eBPF，则非特权进程可以加载某些 eBPF 程序，这些程序的功能集会减少，并且对内核的访问有限。

Verifier

如果允许进程加载 eBPF 程序，则所有程序仍会通过 eBPF 验证程序。eBPF验证器确保程序本身的安全性。这意味着，例如：

- 程序经过验证以确保它们始终运行完成，例如eBPF程序可能永远不会阻塞或永远处于循环中。eBPF 程序可能包含所谓的有界循环，但仅当验证者能够确保循环包含保证变为真的退出条件时，该程序才会被接受。
- 程序不得使用任何未初始化的变量或越界访问内存。
- 程序必须符合系统的大小要求。无法加载任意大的 eBPF 程序。
- 程序必须具有有限的复杂性。验证程序将评估所有可能的执行路径，并且必须能够在配置的复杂性上限限制内完成分析。

Hardening

成功完成验证后，eBPF 程序将根据程序是从特权进程还是非特权进程加载来运行强化过程。此步骤包括：

- Program execution protection：保存 eBPF 程序的内核内存受到保护并变为只读。如果出于任何原因，无论是内核错误还是恶意操纵，试图修改 eBPF 程序，内核将崩溃，而不是允许它继续执行损坏/操纵的程序。
- Mitigation against Spectre：在推测下，CPU 可能会错误预测分支并留下可通过侧通道提取的可观察到的副作用。举几个例子：eBPF 程序屏蔽内存访问，以便在瞬态指令下将访问重定向到受控区域，验证器还遵循仅在推测执行下可访问的程序路径，并且 JIT 编译器在尾部调用无法转换为直接调用的情况下发出 Retpolines。
- Constant blinding：代码中的所有常量都经过盲法处理，以防止 JIT 喷洒攻击。这可以防止攻击者将可执行代码作为常量注入，在存在另一个内核错误的情况下，攻击者可以跳转到 eBPF 程序的内存部分以执行代码。

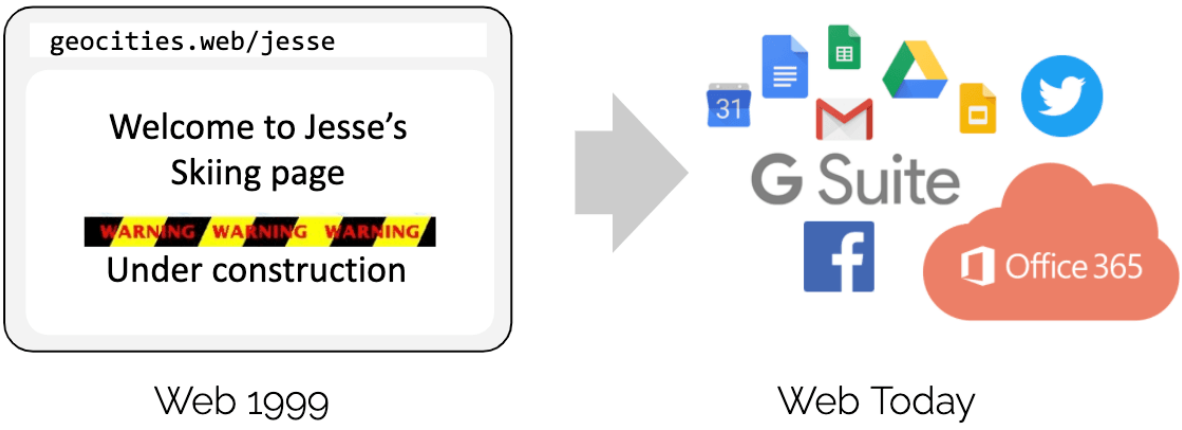
Abstracted Runtime Context

eBPF 程序不能直接访问任意内核内存。必须通过 eBPF 帮助程序访问对程序上下文之外的数据和数据结构的访问。这保证了一致的数据访问，并使任何此类访问都受制于eBPF程序的权限，例如，如果可以保证修改是安全的，则允许运行的eBPF程序修改某些数据结构的数据。eBPF 程序不能随机修改内核中的数据结构。

Why eBPF?

The Power of Programmability

让我们从一个类比开始。你还记得GeoCities吗？20年前，网页过去几乎完全是用静态标记语言（HTML）编写的。网页基本上是一个带有能够显示它的应用程序（浏览器）的文档。看看今天的网页，网页已经成为成熟的应用程序，基于Web的技术已经取代了绝大多数用需要编译的语言编写的应用程序。是什么促成了这种演变？



简短的回答是引入JavaScript的可编程性。它开启了一场巨大的革命，导致浏览器演变成几乎独立的操作系统。

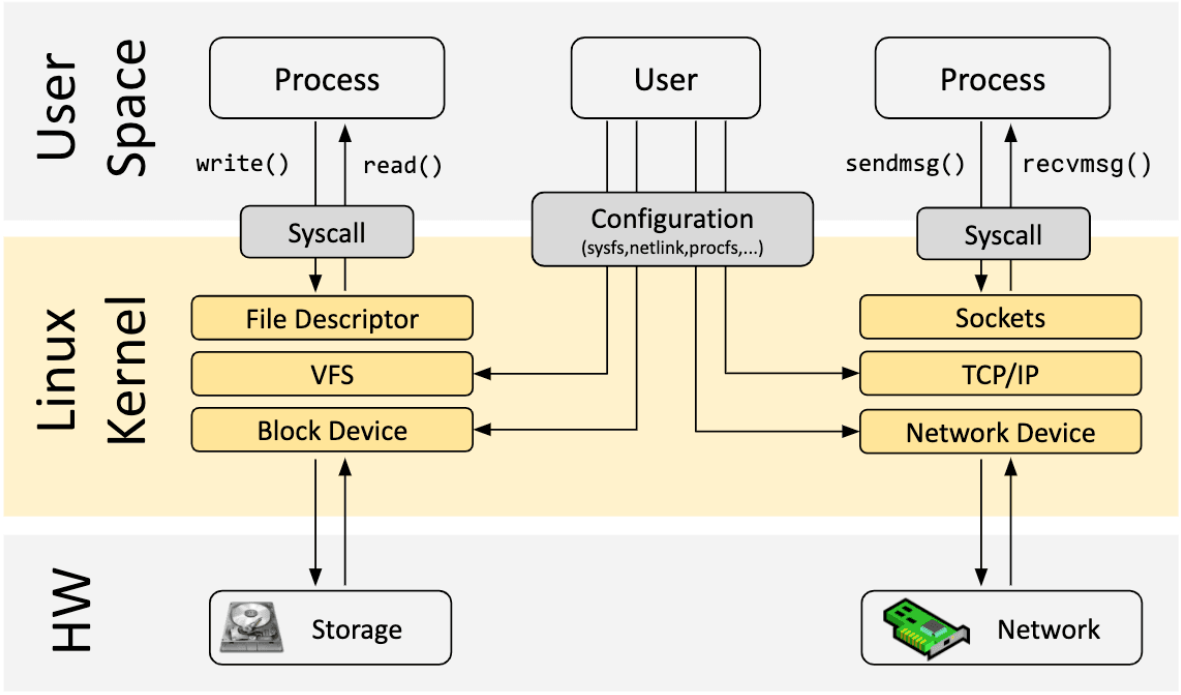
为什么会发生这种演变？程序员不再局限于运行特定浏览器版本的用户。必要构建块的可用性并没有说服标准机构需要一个新的 HTML 标记，而是将底层浏览器的创新步伐与运行在顶部的应用程序分离。这当然有点过于简单，因为HTML确实随着时间的推移而发展并为成功做出了贡献，但HTML本身的发展是不够的。

在举这个例子并将其应用于 eBPF 之前，让我们看一下在引入 JavaScript 时至关重要的几个关键方面：

- Safety：不受信任的代码在用户的浏览器中运行。这是通过沙盒化JavaScript程序和抽象对浏览器数据的访问来解决的。
- Continuous Delivery：程序逻辑的演进必须是可能的，而不需要不断发布新的浏览器版本。这是通过提供足以构建任意逻辑的正确低级构建块来解决的。
- Performance：必须以最小的开销提供可编程性。通过引入即时 （JIT） 编译器解决了这个问题。对于上述所有内容，出于同样的原因，可以在eBPF中找到确切的对应部分。

eBPF's impact on the Linux Kernel

现在让我们回到eBPF。为了理解 eBPF 对 Linux 内核的可编程性影响，它有助于对 Linux 内核的架构以及它如何与应用程序和硬件交互有一个高层次的了解。



Linux 内核的主要目的是抽象硬件或虚拟硬件，并提供一致的 API（系统调用），允许应用程序运行和共享资源。为了实现这一目标，需要维护一组广泛的子系统和层来分配这些职责。每个子系统通常允许进行某种级别的配置，以满足用户的不同需求。如果无法配置所需的行为，则需要更改内核，从历史上看，留下两个选项：

- 原生支持
    - 更改内核源代码并说服 Linux 内核社区需要更改。
    - 等几年，新的内核版本成为商品。
- 内核模块
    - 编写内核模块
    - 定期修复它，因为每个内核版本都可能破坏它
    - 由于缺乏安全边界而导致的 Linux 内核损坏的风险

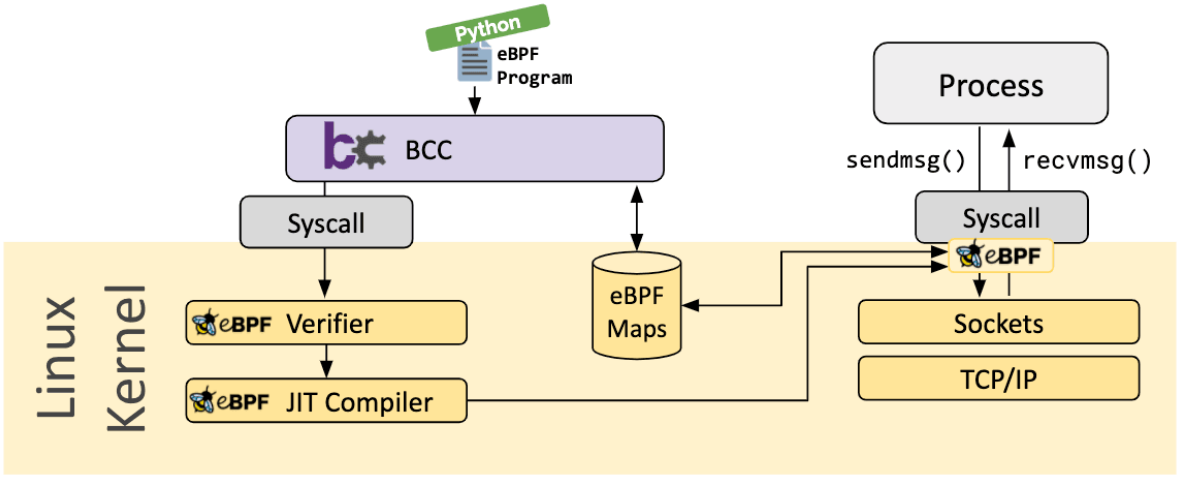
使用 eBPF，可以使用一个新选项，允许重新编程 Linux 内核的行为，而无需更改内核源代码或加载内核模块。在许多方面，这与JavaScript和其他脚本语言如何解锁系统的发展非常相似，这些系统已经变得难以改变或成本高昂。

Development Toolchains

有几个开发工具链可以帮助开发和管理eBPF程序。它们都满足了用户的不同需求：

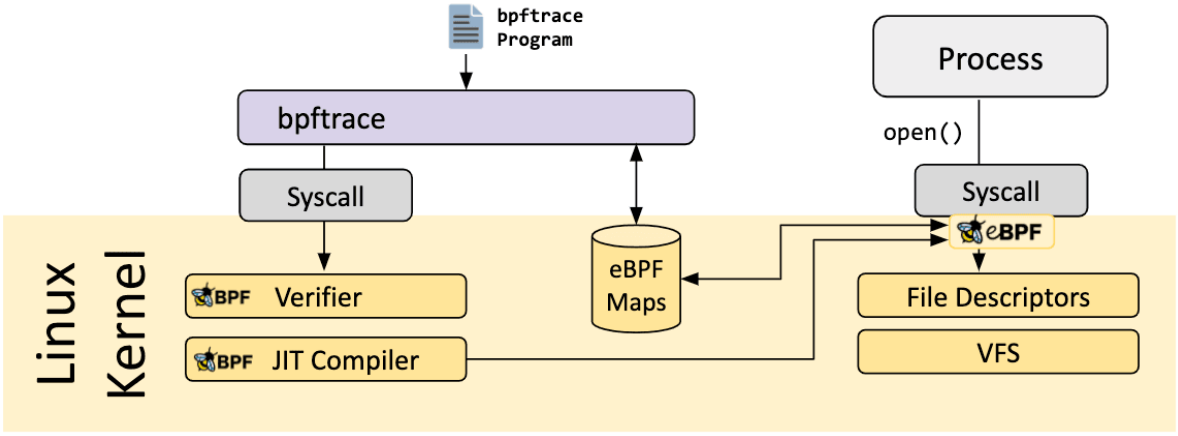
bcc

BCC是一个框架，使用户能够编写嵌入其中的eBPF程序的python程序。该框架主要针对涉及应用程序和系统分析/跟踪的用例，其中eBPF程序用于收集统计信息或生成事件，用户空间中的对应程序收集数据并以人类可读的形式显示。运行python程序将生成eBPF字节码并将其加载到内核中。



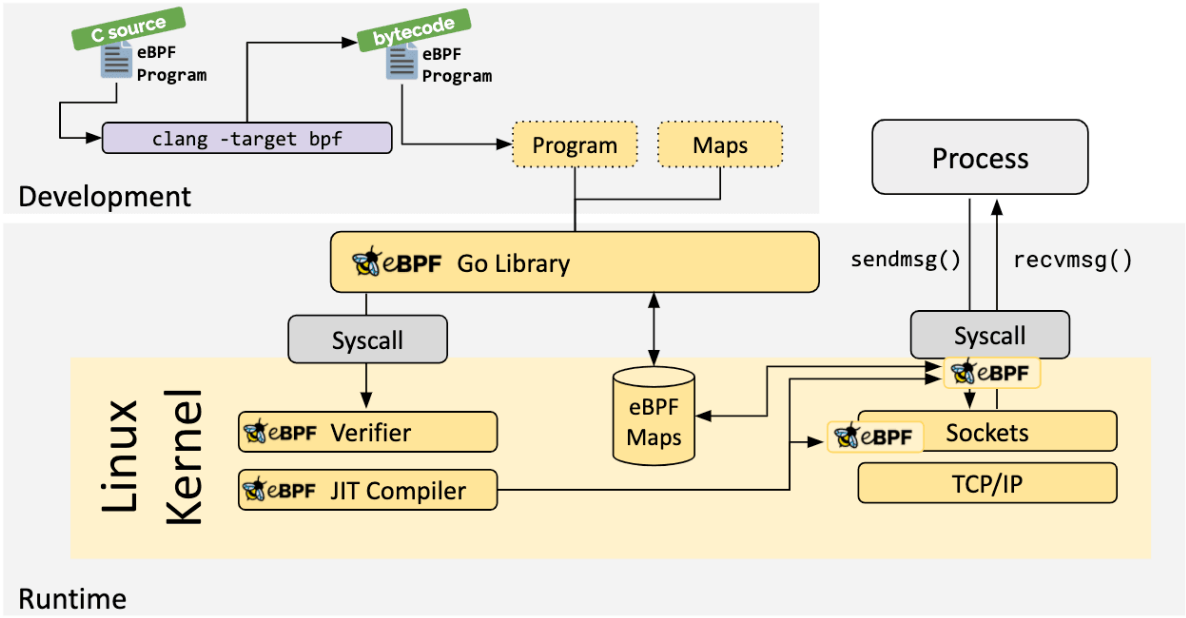
bpftrace

bpftrace 是 Linux eBPF 的高级跟踪语言，在最近的 Linux 内核 （4.x） 中可用。bpftrace 使用 LLVM 作为后端将脚本编译为 eBPF 字节码，并利用 BCC 与 Linux eBPF 子系统以及现有的 Linux 跟踪功能进行交互：内核动态跟踪 （kprobes）、用户级动态跟踪 （uprobes） 和跟踪点。bpftrace 语言的灵感来自 awk、C 和前身的跟踪器，如 DTrace 和 SystemTap。



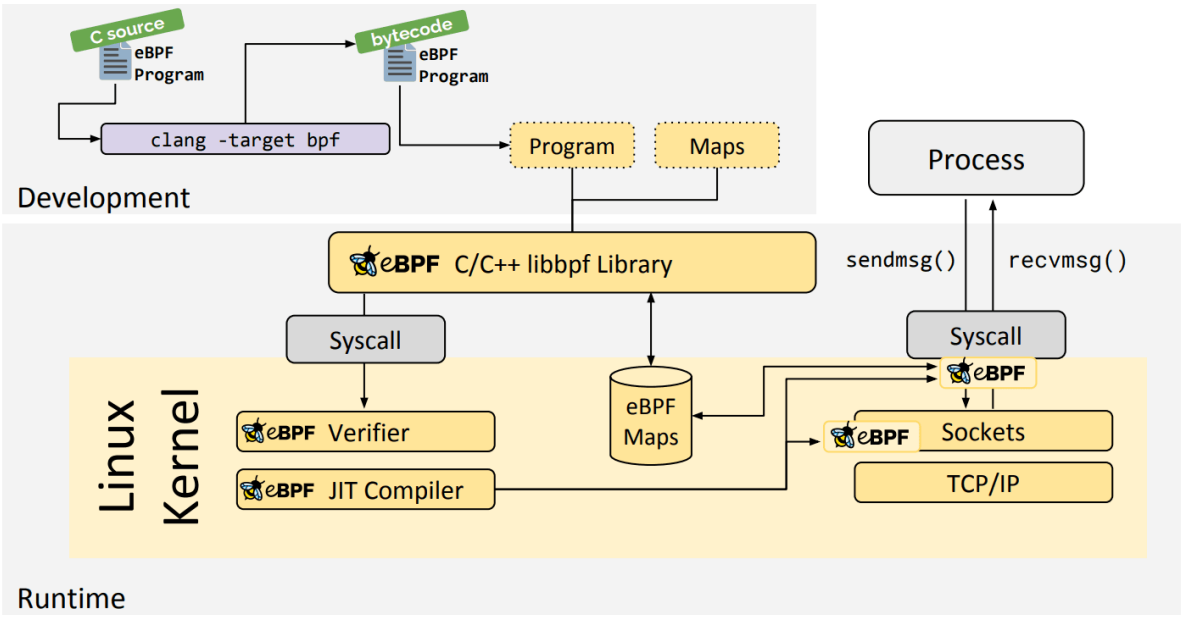
eBPF Go Library

eBPF Go库提供了一个通用的eBPF库，它将获取eBPF字节码的过程与eBPF程序的加载和管理解耦。eBPF程序通常是通过编写更高级的语言，然后使用clang / LLVM编译器编译为eBPF字节码来创建的。



libbpf C/C++ Library

libbpf 库是一个基于 C/C++ 的通用 eBPF 库，它有助于将 clang/LLVM 编译器生成的 eBPF 对象文件加载到内核中解耦，并且通常通过为应用程序提供易于使用的库 API 来抽象与 BPF 系统调用的交互。



Further Reading

If you would like to learn more about eBPF, continue reading using the following additional materials:

Documentation

- BPF & XDP Reference Guide, Cilium Documentation, Aug 2020
- BPF Documentation, BPF Documentation in the Linux Kernel
- BPF Design Q&A, FAQ for kernel-related eBPF questions

Tutorials

- Learn eBPF Tracing: Tutorial and Examples, Brendan Gregg's Blog, Jan 2019
- XDP Hands-On Tutorials, Various authors, 2019
- BCC, libbpf and BPF CO-RE Tutorials, Facebook's BPF Blog, 2020

Talks

Generic

- eBPF and Kubernetes: Little Helper Minions for Scaling Microservices (Slides), Daniel Borkmann, KubeCon EU, Aug 2020
- eBPF - Rethinking the Linux Kernel (Slides), Thomas Graf, QCon London, April 2020
- BPF as a revolutionary technology for the container landscape (Slides), Daniel Borkmann, FOSDEM, Feb 2020
- BPF at Facebook, Alexei Starovoitov, Performance Summit, Dec 2019
- BPF: A New Type of Software (Slides), Brendan Gregg, Ubuntu Masters, Oct 2019
- The ubiquity but also the necessity of eBPF as a technology, David S. Miller, Kernel Recipes, Oct 2019

Deep Dives

- BPF and Spectre: Mitigating transient execution attacks (Slides) , Daniel Borkmann, eBPF Summit, Aug 2021
- BPF Internals (Slides), Brendan Gregg, USENIX LISA, Jun 2021

Cillium

- Advanced BPF Kernel Features for the Container Age (Slides), Daniel Borkmann, FOSDEM, Feb 2021
- Kubernetes Service Load-Balancing at Scale with BPF & XDP (Slides), Daniel Borkmann & Martynas Pumputis, Linux Plumbers, Aug 2020
- Liberating Kubernetes from kube-proxy and iptables (Slides), Martynas Pumputis, KubeCon US 2019
- Understanding and Troubleshooting the eBPF Datapath in Cilium (Slides), Nathan Sweet, KubeCon US 2019
- Transparent Chaos Testing with Envoy, Cilium and BPF (Slides), Thomas Graf, KubeCon EU 2019
- Cilium - Bringing the BPF Revolution to Kubernetes Networking and Security (Slides), Thomas Graf, All Systems Go!, Berlin, Sep 2018
- How to Make Linux Microservice-Aware with eBPF (Slides), Thomas Graf, QCon San Francisco, 2018
- Accelerating Envoy with the Linux Kernel, Thomas Graf, KubeCon EU 2018
- Cilium - Network and Application Security with BPF and XDP (Slides), Thomas Graf, DockerCon Austin, Apr 2017

Hubble

- Hubble - eBPF Based Observability for Kubernetes, Sebastian Wicki, KubeCon EU, Aug 2020

Books

- Learning eBPF, Liz Rice, O'Reilly, 2023
- Security Observability with eBPF, Natália Réka Ivánkó and Jed Salazar, O'Reilly, 2022
- What is eBPF?, Liz Rice, O'Reilly, 2022
- Systems Performance: Enterprise and the Cloud, 2nd Edition, Brendan Gregg, Addison-Wesley Professional Computing Series, 2020
- BPF Performance Tools, Brendan Gregg, Addison-Wesley Professional Computing Series, Dec 2019
- Linux Observability with BPF, David Calavera, Lorenzo Fontana, O'Reilly, Nov 2019

Articles & Blogs

- BPF for security - and chaos - in Kubernetes, Sean Kerner, LWN, Jun 2019
- Linux Technology for the New Year: eBPF, Joab Jackson, Dec 2018
- A thorough introduction to eBPF, Matt Fleming, LWN, Dec 2017
- Cilium, BPF and XDP, Google Open Source Blog, Nov 2016
- Archive of various articles on BPF, LWN, since Apr 2011
- Various articles on BPF by Cloudflare, Cloudflare, since March 2018
- Various articles on BPF by Facebook, Facebook, since August 2018