eBPF Documentation

Table of contents

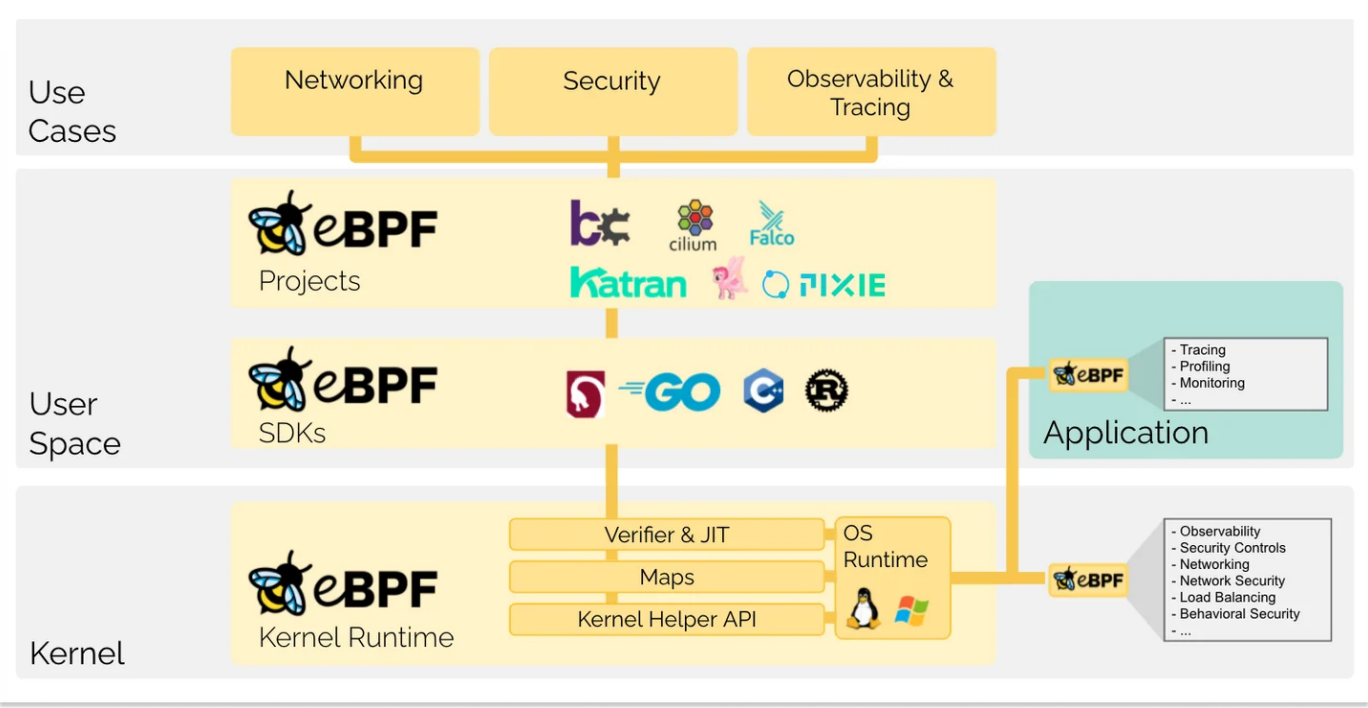
Libri

Articoli & Blog

Cosa è eBPF

eBPF è una tecnologia rivoluzionaria che ha origine nel kernel Linux e può eseguire programmi in sandbox in un contesto privilegiato del kernel senza il bisogno di cambiare il codice sorgente del kernel o caricare dei moduli del kernel.

Storicamente, il sistema operativo è sempre stato un posto ideale per implementare l'osservabilità, la sicurezza, e le funzionalità di rete grazie all'abilità privilegiata del kernel di osservare e controllare il sistema nella sua interezza. Allo stesso tempo, un kernel di un sistema operativo è difficile da far evolvere a causa del suo ruolo centrale e degli elevati requisiti di stabilità e sicurezza. Il rapporto di innovazione a livello di sistema operativo è stato così tradizionalmente più basso a confronto delle funzionalità implementate fuori dal sistema operativo.



eBPF rivoluziona in modo fondamentale questa formula. Permettendo di eseguire programmi in sandbox all'interno del sistema operativo, gli sviluppatori di applicazioni possono eseguire i programmi eBPF per aggiungere ulteriori funzionalità al sistema operativo in esecuzione. Il

sistema operativo quindi garantisce sicurezza ed efficienza di esecuzione come se fosse compilato nativamente con l'aiuto del compilatore Just-In-Time (JIT) e del motore di verifica. Questo ha portato a un'ondata di progetti basati su eBPF coprendo un largo campo di casi d'uso, incluso la rete, osservabilità, e funzionalità di sicurezza di prossima generazione.

Oggi, eBPF è usato ampiamente per gestire una larga varietà di casi d'uso: fornire networking e load-balancing ad alte prestazioni nei moderni data center e ambienti cloud native, estraendo dati dettagliati di osservabilità sulla sicurezza con un basso overhead, aiutando gli sviluppatori di applicazioni a tracciare le applicazioni, fornendo informazioni per la risoluzione di problemi di prestazioni, applicando in modo preventivo la sicurezza delle applicazioni e dei container e molto altro ancora. Le possibilità sono infinite e l'innovazione che eBPF sta sbloccando è appena iniziata.

Cos'è eBPF.io?

eBPF.io è un luogo per tutti per imparare e collaborare sull'argomento di eBPF. eBPF è una community aperta e ognuno può partecipare e condividere. Sia che vogliate leggere una prima introduzione a eBPF, trovare ulteriore materiale di lettura, o fare i primi passi per divenire contributore a progetti maggiori di eBPF, eBPF.io vi aiuterà lungo il percorso.

Cosa significano eBPF e BPF?

Originalmente BPF stava per Berkeley Packet Filter, ma ora che eBPF (extended BPF) può fare molto di più che filtrare i pacchetti di rete, l'acronimo non ha più molto senso, ed eBPF è considerato un termine a se stante. Nel codice sorgente Linux, il termine BPF persiste, e negli strumenti e nella documentazione, il termine BPF e eBPF sono generalmente intercambiabili. L'originale BPF è a volte riferito con cBPF (classic BPF) per distinguerlo da eBPF.

Come si chiama l'ape?

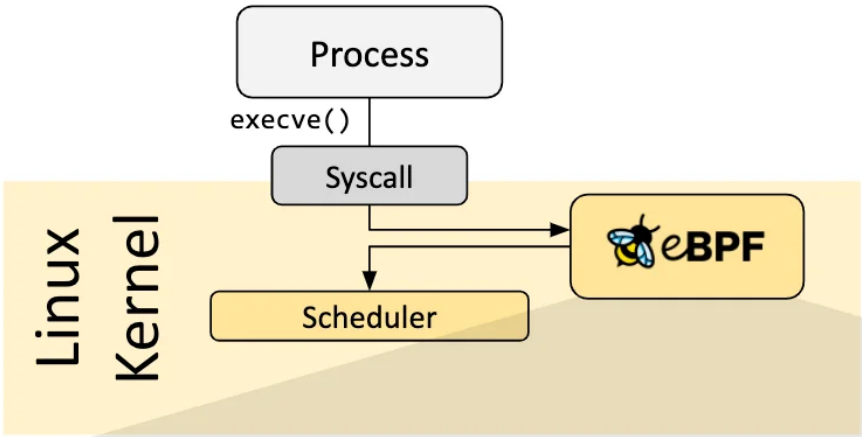
L'ape è il logo ufficiale per eBPF ed è stato originariamente creato da Vadim Shchekoldin. Al **primo eBPF Summit** c'è stata una votazione e l'ape è stata nominata eBee. (Per dettagli sull'uso accettabile del logo, consultate le **Linee guida del marchio** della Linux Foundation).

Introduzione a eBPF

I capitoli seguenti sono una veloce introduzione a eBPF. Se volete scoprire di più su eBPF, guardate la **eBPF & XDP Reference Guide**. Sia che siate uno sviluppatore che cerca di creare un programma eBPF, o siate interessati a sfruttare una soluzione che utilizza eBPF, è utile comprenderne i concetti e l'architettura di base.

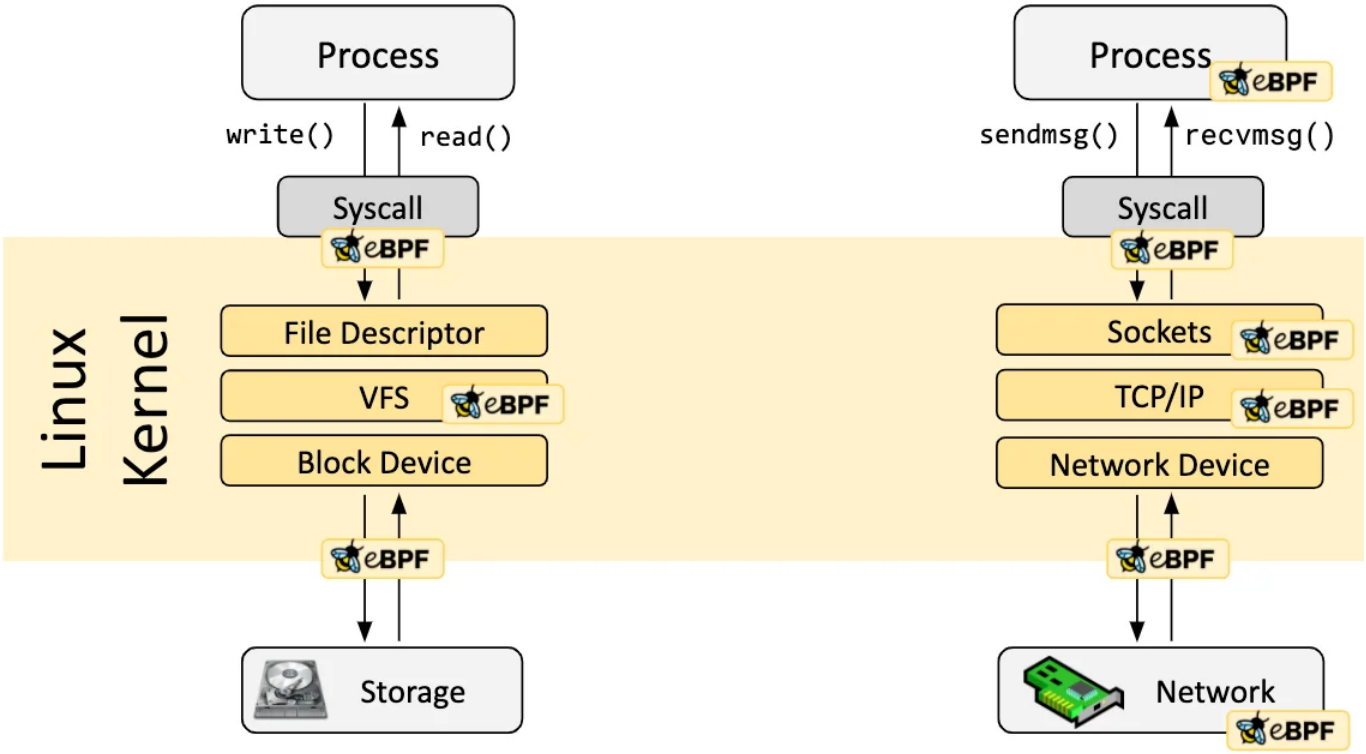
Panoramica degli Hook

I programmi eBPF sono event-driven e sono eseguiti quando il kernel o un'applicazione passa da certi hook. Hook predefiniti sono inclusi anche nelle chiamate di sistema, entrata e uscita di funzioni, kernel tracepoint, eventi di rete e molti altri.



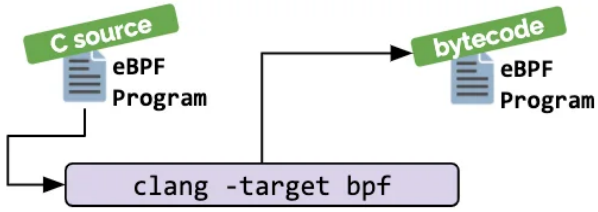

```
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

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

        return 0;
}
```

Se un hook predefinito non esiste per una necessità in particolare, è possibile creare un kernel probe (kprobe) o user probe (uprobe) per agganciare i programmi eBPF quasi ovunque nel kernel o nelle applicazioni utente.



Come sono scritti i programmi eBPF?

In molti scenari, eBPF non viene usato direttamente ma indirettamente attraverso progetti come **Cilium**, **bcc**, or **bpftrace** che forniscono una astrazione sopra eBPF e non richiedono di scrivere direttamente dei programmi ma invece offrono abilità per specificare definizioni basate su intenti che vengono poi implementate con eBPF.

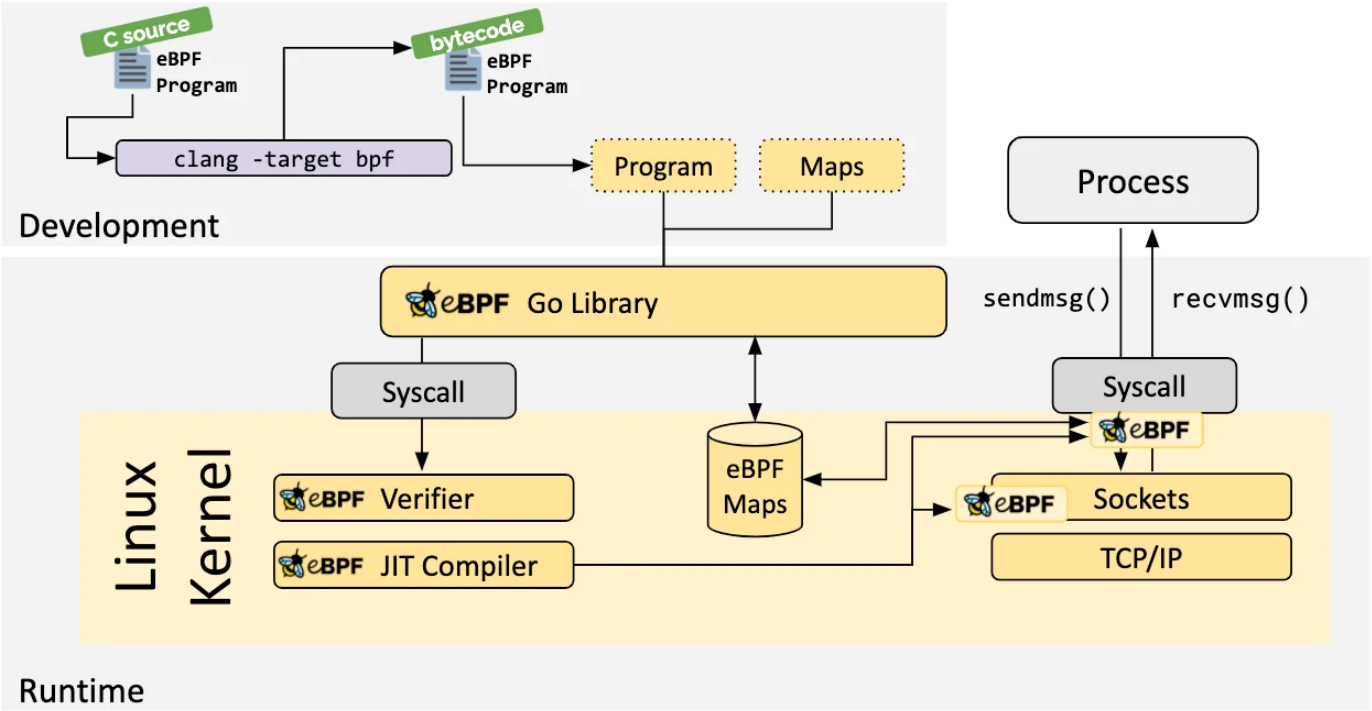


Se non esiste un livello di astrazione più alto, i programmi devono essere scritti direttamente. Il kernel Linux si aspetta che i programmi eBPF vengano caricati in forma di bytecode. Mentre è sicuramente possibile scrivere direttamente bytecode, la pratica di sviluppo più comunemente usata è di utilizzare una suite di compilatori come **LLVM** per compilare codice pseudo-C in eBPF bytecode.

Architettura del Loader e Verifica

Architettura del Loader e Verifica

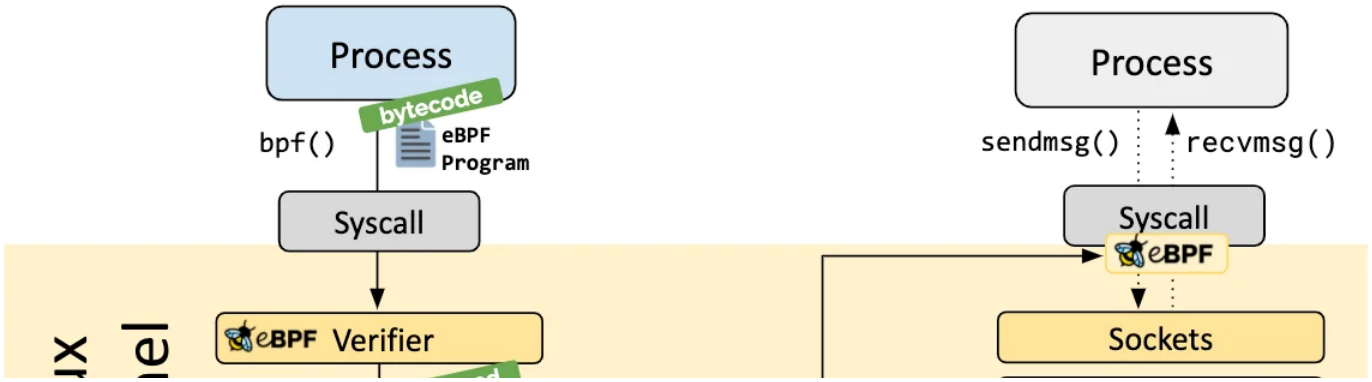
Una volta identificato l'hook desiderato, il programma eBPF può essere caricato nel kernel Linux usando la chiamata di sistema bpf. Questo è solitamente effettuato usando una libreria eBPF tra quelle esistenti. La sezione seguente fornisce una introduzione alle toolchain di sviluppo disponibili.

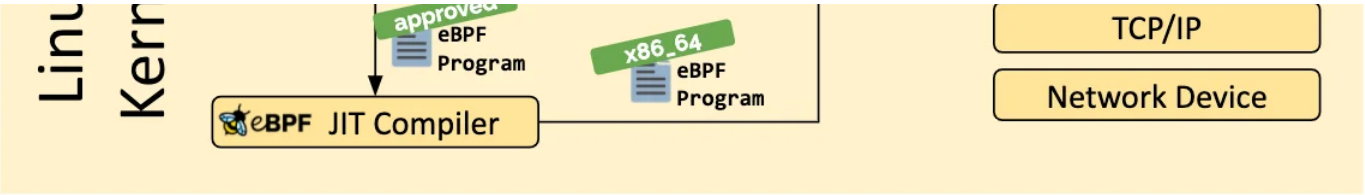


Dal momento in cui un programma è caricato nel kernel Linux, passa attraverso due fasi prima di venire agganciato all'hook richiesto:

Verifica

La fase di verifica assicura che un programma eBPF sia sicuro da eseguire. Convalida che il programma soddisfi determinate condizioni, per esempio:





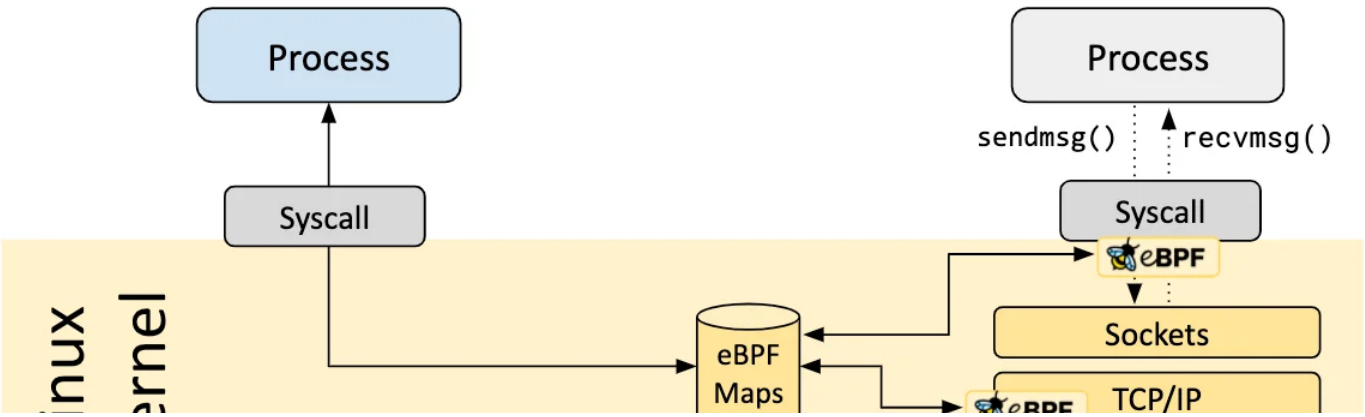
- Il processo che carica il programma eBPF possiede le capacità richieste (privilegi). A meno che “unprivileged eBPF” sia attivo, solo i processi privilegiati possono caricare i programmi eBPF.
- Il programma non si può bloccare o danneggiare il sistema in alcun modo.
- Il programma viene eseguito sempre fino alla fine (i.e. il programma non resta in un loop infinito, ritardando ulteriori elaborazioni).

Compilazione JIT

La fase della compilazione Just-in-Time (JIT) traduce il bytecode generico del programma nelle istruzioni macchina vere e proprie, al fine di ottimizzare la velocità d'esecuzione. Questo rende l’esecuzione dei programmi eBPF efficiente come quella del codice nativo del kernel o dei moduli kernel caricati.

Mappe

Un aspetto vitale dei programmi eBPF è l’abilità di condividere le informazioni raccolte e salvarne lo stato. A questo scopo, i programmi eBPF forniscono il concetto di mappe eBPF per salvare e recuperare i dati in un largo insieme di strutture dati. Le mappe eBPF possono essere accessibili dai programmi eBPF e anche dalle applicazioni nello spazio utente attraverso una chiamata di sistema.



La seguente è una lista non esaustiva dei tipi di mappe supportate per dare un'idea della diversità delle strutture dati:

- Hash tables, Arrays
- LRU (Least Recently Used)
- Ring Buffer
- Stack Trace
- LPM (Longest Prefix match)
- ...

Chiamate Helper

I programmi eBPF non possono essere invocati da funzioni arbitrarie del kernel. Se ciò fosse permesso i programmi eBPF sarebbero vincolati a versioni particolari del kernel e se ne complicherebbe quindi la compatibilità. Invece, i programmi eBPF possono fare delle chiamate a delle funzioni di helper, che sono ben note e stabili API offerte dal kernel.

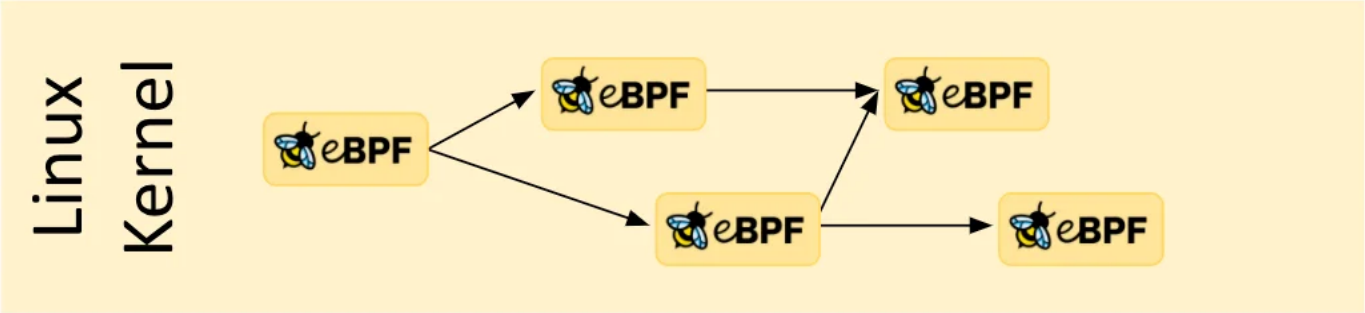
L'insieme delle chiamate helper disponibili evolve costantemente. Esempi di chiamate helper:

- Genera numeri casuali

- Restituisce l'ora e la data attuale
- Accesso a una mappa eBPF
- Restituisce il contesto del processo-cgroup
- Manipola i pacchetti di rete e la logica di inoltro

Chiamate in coda e chiamate di funzione (Tail & Function Calls)

I programmi eBPF sono componibili con il concetto di chiamata in coda (tail call) e chiamate di funzione (function call). Le chiamate di funzione permettono la definizione e la chiamata a funzioni nell'ambito del programma eBPF e rimpiazzano il contesto di esecuzione, simile a come opera la chiamata di sistema execve() per i processi regolari.



Sicurezza eBPF

Da un grande potere derivano grandi responsabilità.

eBPF è una tecnologia incredibilmente potente e ora viene eseguita nel nucleo di molti componenti critici nelle infrastrutture software. Durante lo sviluppo di eBPF, la sicurezza di eBPF è stato l'aspetto più cruciale nel momento in cui l'inclusione di eBPF è stata considerata nel kernel Linux. La sicurezza dell'EBPF è assicurata attraverso differenti livelli:

Privilegi richiesti

A meno che "unprivileged eBPF" sia disabilitato, tutti i processi che intendono caricare programmi eBPF nel kernel Linux devono venire eseguiti in modalità privilegiata (root) o richiedere le capacità CAP_BPF. Questo significa che programmi non attendibili non possono eseguire programmi eBPF.

Se “unprivileged eBPF” è abilitato, i processi non attendibili possono eseguire programmi eBPF soggetti a funzionalità di un insieme ridotto e con accesso limitato al kernel.

Verificatore

Anche se un programma può caricare un programma eBPF, tutti i programmi passano attraverso il verificatore eBPF. Il verificatore eBPF garantisce la sicurezza del programma stesso. Questo significa, per esempio:

- I programmi sono validati per essere certi che essi vengano sempre eseguiti fino al completamento, e.g. un programma eBPF non deve mai bloccarsi o entrare in un loop indefinitamente. I programmi eBPF possono contenere i cosiddetti “bounded loops” ma il programma viene accettato solo se il verificatore può assicurare che il loop contenga una condizione di uscita che sia garantita diventare vera.
- I programmi non possono usare nessuna variabile non inizializzata o accedere ad aree di memoria fuori dai limiti.
- I programmi devono rientrare nelle dimensioni richieste dal sistema. Non è possibile caricare programmi eBPF di dimensioni arbitrarie.
- Il programma deve avere una complessità finita. Il verificatore valuterà tutti i possibili percorsi di esecuzione e deve essere in grado di completare l'analisi entro i limiti del limite superiore di complessità configurato.

Il verificatore è inteso come uno strumento di sicurezza, che controlla che i programmi siano sicuri da eseguire. Non è uno strumento di sicurezza che ispeziona ciò che i programmi stanno facendo.

Hardening

Dopo aver completato la verifica con successo, il programma eBPF passa attraverso un processo di hardening in accordo al fatto se il programma viene caricato da un processo privilegiato o no. Questa fase include:

- **Protezione nell’esecuzione dei programmi**: la memoria del kernel in cui risiede il programma eBPF è protetta e impostata a sola lettura. Se per qualunque motivo, per un bug del kernel o manipolazione fraudolenta, venisse tentata la modifica del programma eBPF, il kernel si pianterebbe piuttosto che continuare l’esecuzione del programma corrotto

eBPF, il kernel si pianterebbe piuttosto che continuare l'esecuzione del programma corretto o manipolato.

- **Mitigazione contro l'attacco Spectre**: In caso di speculazione, le CPU possono predire in modo errato i rami di esecuzione e lasciare effetti collaterali osservabili che possono essere sfruttati tramite un canale laterale. Per fare qualche esempio: i programmi eBPF mascherano gli accessi alla memoria al fine di reindirizzare l'accesso sotto istruzioni transitorie ad aree monitorate, e pure il verificatore segue percorsi di programma accessibili solo in esecuzione speculativa e il compilatore JIT emette Retpolines (i.e. return trampoline) qualora le chiamate in coda (tail call) non possano essere convertite in chiamate dirette.
- **Accecamento delle costanti**: tutte le costanti presenti nel codice sorgente sono accecate al fine di prevenire attacchi JIT di tipo spraying. Questo accorgimento impedisce a un attaccante di iniettare codice eseguibile al posto di una costante e sfruttare un bug del kernel, che permetterebbe al codice attaccante di fare un jump nella sezione di memoria del programma eBPF e quindi eseguire codice maligno.

Contesto Astratto di Runtime

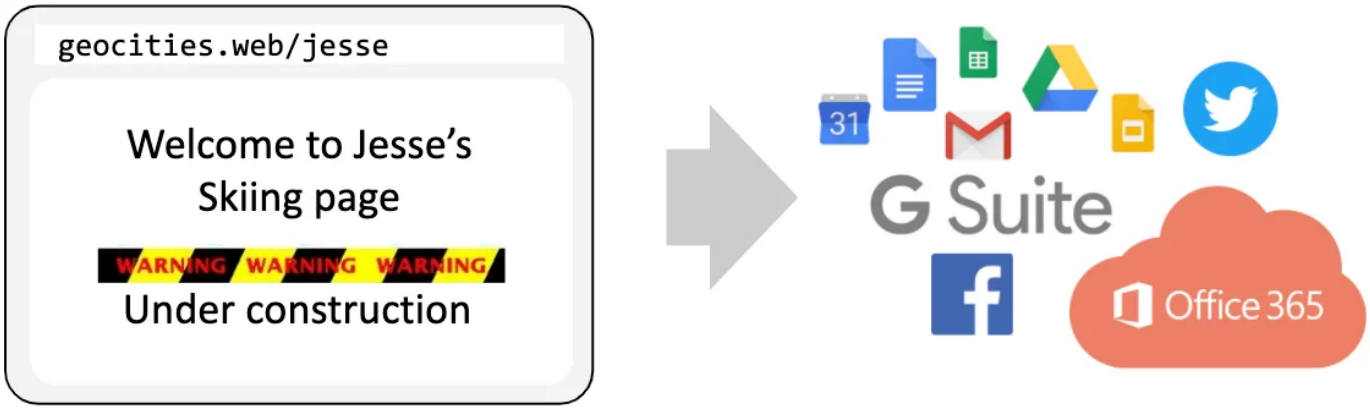
I Programmi eBPF non possono accedere direttamente ad arbitrarie locazioni di memoria. L'accesso ai dati e alle strutture dati che risiedono al di fuori del contesto del programma, deve essere ottenuto per mezzo di specifiche funzioni helper di eBPF. Ciò garantisce un accesso ai dati coerente e rende ogni accesso subordinato agli stessi privilegi del programma eBPF, e.g. un programma eBPF in esecuzione può modificare i dati di specifiche strutture dati soltanto se la modifica è garantita sicura. Un programma eBPF non può arbitrariamente modificare strutture nel kernel.

Perché eBPF?

La Potenza della Programmabilità

Iniziamo da una analogia. Ricordi GeoCities? 20 anni fa le pagine web erano scritte quasi esclusivamente in un linguaggio statico a markup (HTML). Una pagina web consisteva essenzialmente in un documento che una applicazione (browser) era in grado di interpretare e mostrare a video. Dando un'occhiata a come sono strutturate le pagine web odierne, si può notare come una pagina web incorpori vere e proprie applicazioni e come la tecnologia web

notare come una pagina web incorpori vere e proprie applicazioni e come la tecnologia web abbia preso il posto di una quantità di applicazioni scritte in linguaggi compilati. Cosa ha reso possibile questa evoluzione?



Web 1999 Web Today

La risposta breve è la programmabilità, con l'introduzione di JavaScript. Ha reso possibile una rivoluzione che ha fatto diventare i browser dei veri e propri sistemi operativi indipendenti.

Perché questa evoluzione ha potuto aver luogo? I programmatori non erano più limitati dal particolare browser che gli utenti impiegavano. Invece di cercar di convincere gli organismi preposti alla standardizzazione che un nuovo tag HTML era necessario, la disponibilità dei necessari blocchi costruttivi ha disaccoppiato le velocità dell'innovazione del browser sottostante da quella dell'applicazione che il browser eseguiva. Questa ovviamente è una descrizione sommaria e molto semplificata, dato che pure l'HTML è evoluto nel corso del tempo e pure la sua evoluzione ha contribuito al successo, ma da sola non sarebbe stata sufficiente.

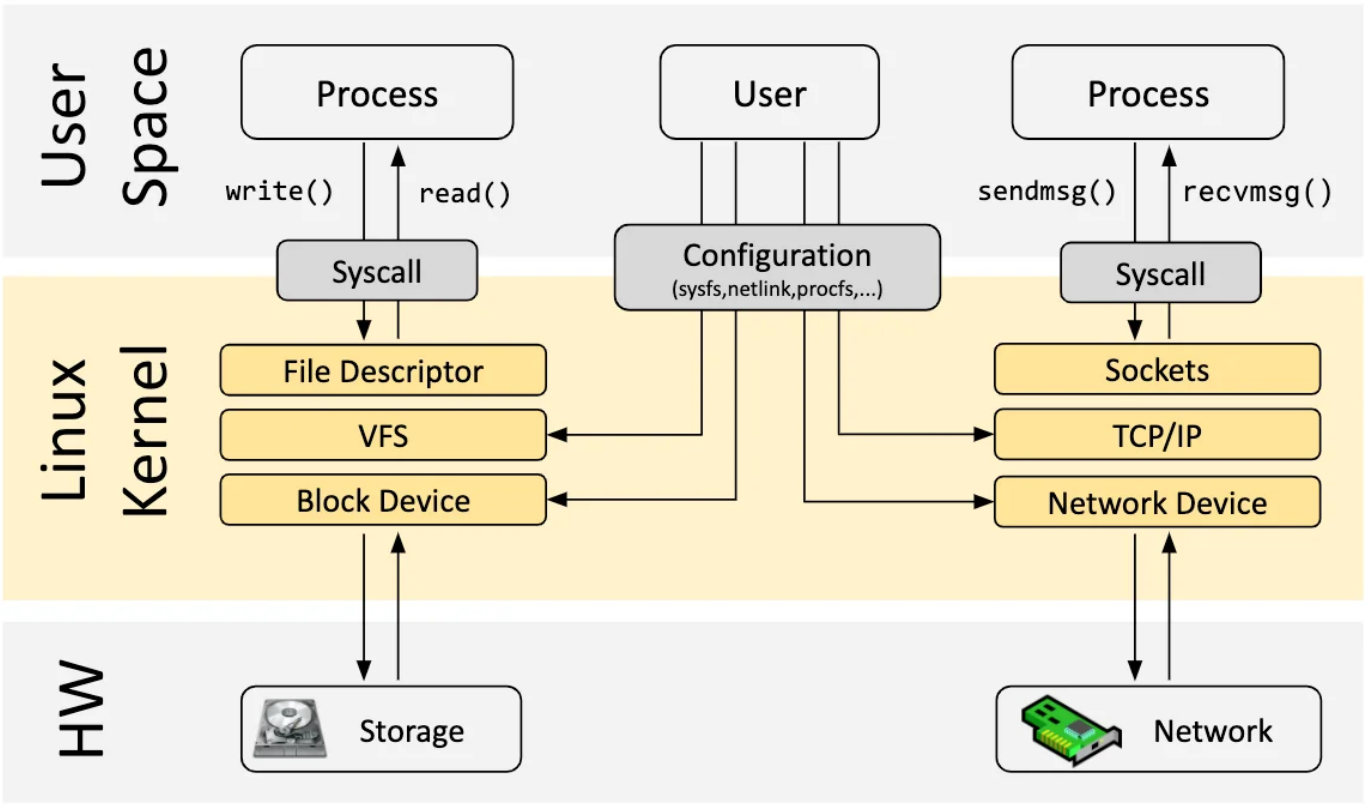
Prima di prendere questo esempio e applicarlo all'eBPF, diamo un'occhiata ad alcuni aspetti chiave che furono vitali nell'introduzione di JavaScript:

- **Sicurezza**: codice non verificato viene eseguito nel browser dell'utente. Questo problema è stato risolto eseguendo i programmi JavaScript in una sandbox e astraendo l'accesso ai dati del browser.
- **Continuous Delivery**: L'evoluzione della logica dei programmi deve essere possibile senza che sia necessario rilasciare sempre nuove versioni del browser. Questo problema è stato risolto fornendo i necessari blocchi costruttivi a basso livello, sufficienti per costruire qualunque logica a partire da essi.

- **Prestazioni**: La programmabilità deve essere garantita con un overhead minimale. Questo problema è stato risolto con l'introduzione di un compilatore Just-in-Time (JIT). Per tutte queste ragioni è possibile individuare le medesime controparti in eBPF, introdotte per le medesime motivazioni.

L'impatto di eBPF sul Kernel di Linux

Ora torniamo a eBPF. Al fine di comprendere l'impatto della programmabilità di eBPF sul kernel di Linux, può aiutare farsi un'idea ad alto livello dell'architettura del kernel di Linux, e come questo interagisce con le applicazioni e con l'hardware.



Lo scopo primario del kernel di Linux è astrarre dall'hardware o dall'hardware virtuale e fornire un'API coerente (system call) che permetta alle applicazioni di eseguire e condividere le risorse. Per ottenere questo risultato, un ampio numero di sottosistemi e strati sono mantenuti al fine di distribuire queste responsabilità. Ogni sottosistema tipicamente consente, per un certo livello di configurazione, di tenere conto dei vari bisogni degli utenti. Se un certo comportamento non può essere configurato, è richiesta una modifica del kernel, e ciò storicamente conduce a due opzioni possibili:

Supporto Nativo

- Supporto Nativo
  1. Modifica il codice sorgente del kernel e convinci la community del kernel di Linux che il cambiamento è necessario.
  2. Attendi diversi anni che la nuova versione del kernel divenga una commodity.

Modulo del Kernel

- Modulo del Kernel
  1. Scrivi un modulo del Kernel
  2. Predisponiti a doverlo rivedere frequentemente, dato che ogni nuova versione del kernel può comprometterne il funzionamento.
  3. Corri il rischio di compromettere il kernel a causa dell'assenza di limiti operativi di sicurezza.

Con eBPF è disponibile una nuova opzione che consente la riprogrammazione del comportamento del kernel di Linux senza che siano necessarie modifiche al codice sorgente del kernel o il caricamento di moduli del kernel. Sotto diversi aspetti ciò è equiparabile al modo in cui JavaScript e altri linguaggi di scripting hanno consentito l'evoluzione di sistemi la cui modifica era divenuta troppo complessa o troppo costosa.

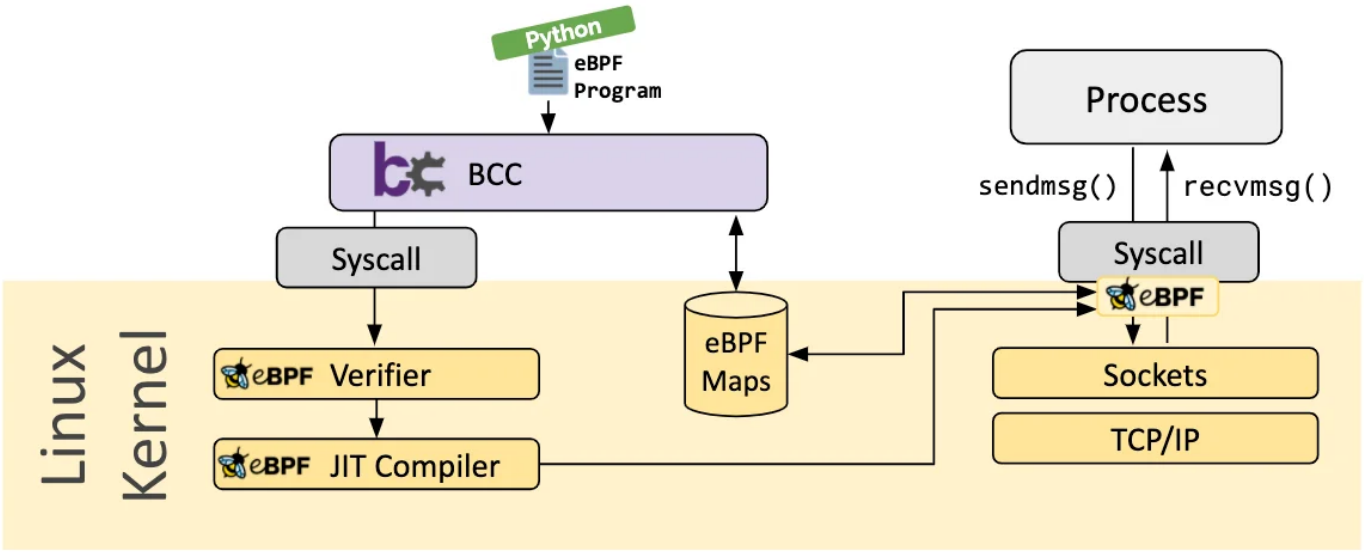
Toolchain di sviluppo

Esistono molteplici toolchain di sviluppo a supporto dello sviluppo e della gestione dei programmi eBPF. Ognuna è dedicata ad assolvere specifiche necessità degli utenti:

bcc

BCC è un framework che permette agli utenti di scrivere programmi in Python che possono incorporare programmi eBPF al loro interno. Il framework è principalmente orientato a casi d'uso

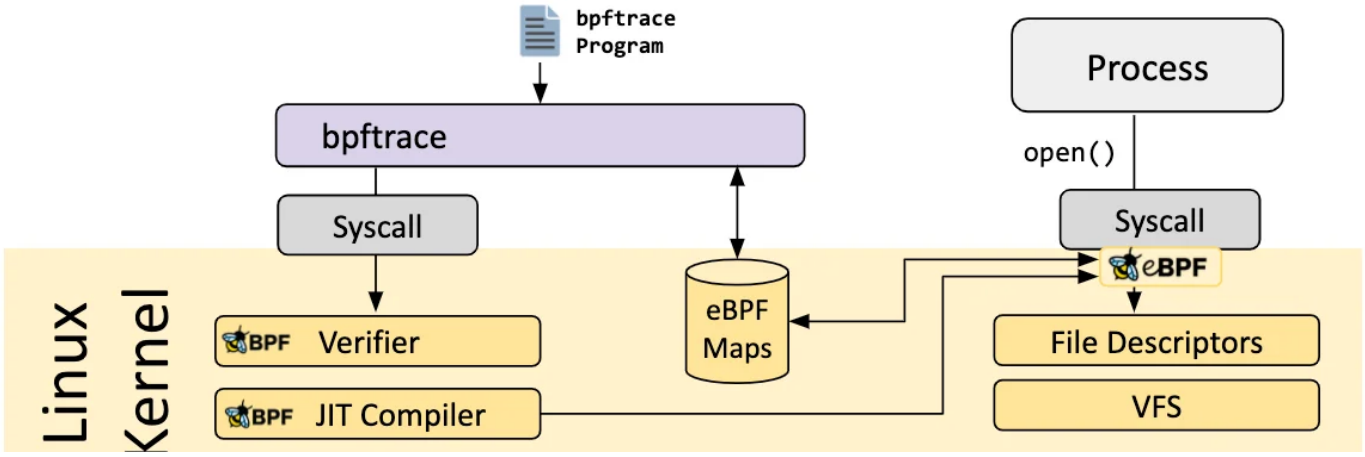
inerenti alle applicazioni e a sistemi di profilatura/tracciamento, dove un programma eBPF viene usato per raccogliere dati statistici e generare eventi, messi a disposizione di una controparte nello spazio utente che raccoglie i dati e li visualizza in una forma leggibile. L'esecuzione del programma in Python genera il codice bytecode eBPF e lo carica nel kernel.



bpftrace

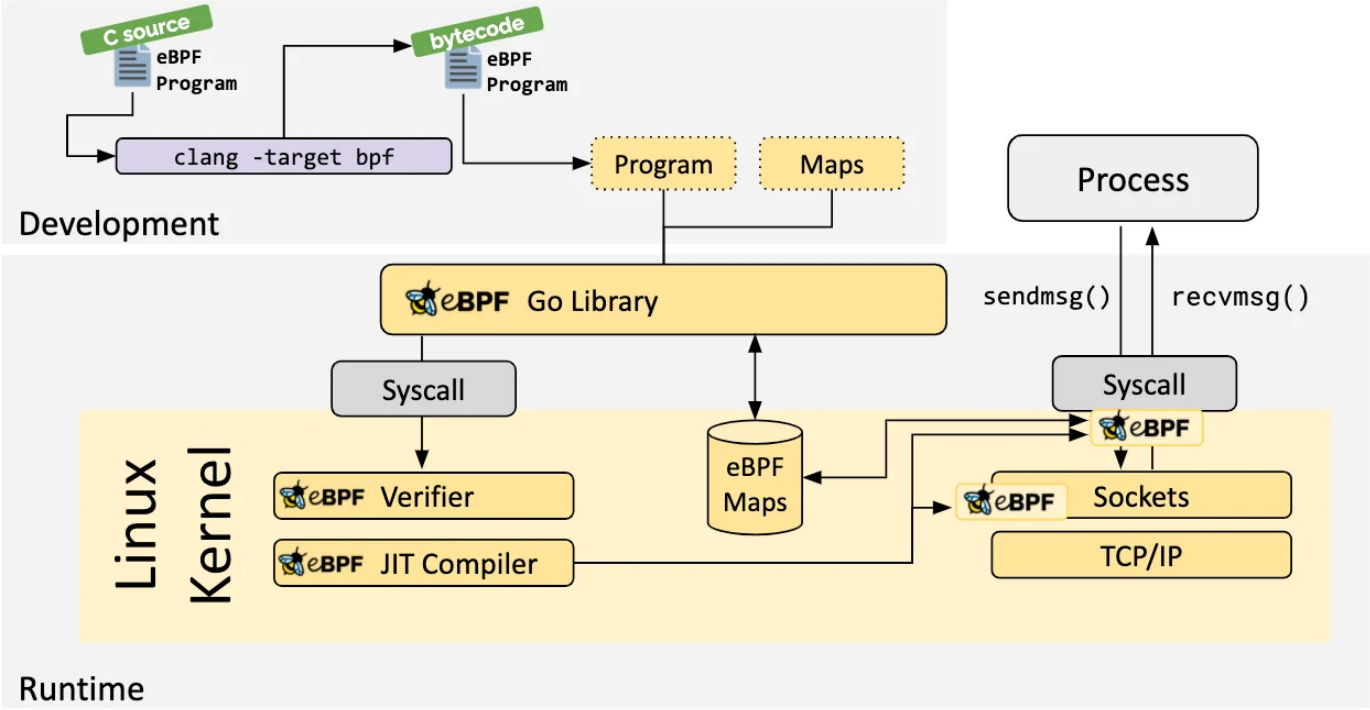
bpftrace è un linguaggio ad alto livello usato per il tracing per eBPF su Linux ed è disponibile nei kernel di Linux semi-recenti (4.x).

Bpftrace usa la LLVM come backend per compilare script direttamente in bytecode eBPF, fa uso di BCC per l'interazione con i sottosistemi eBPF di Linux, ed è pure in grado di sfruttare le funzionalità di Linux per il tracing: tracing dinamico a livello kernel (kprobes), tracing dinamico a livello utente (uprobes), e tracepoint. Il linguaggio bpftrace trae ispirazione da awk, C e da sistemi di tracciamento preesistenti come DTrace e SystemTap.



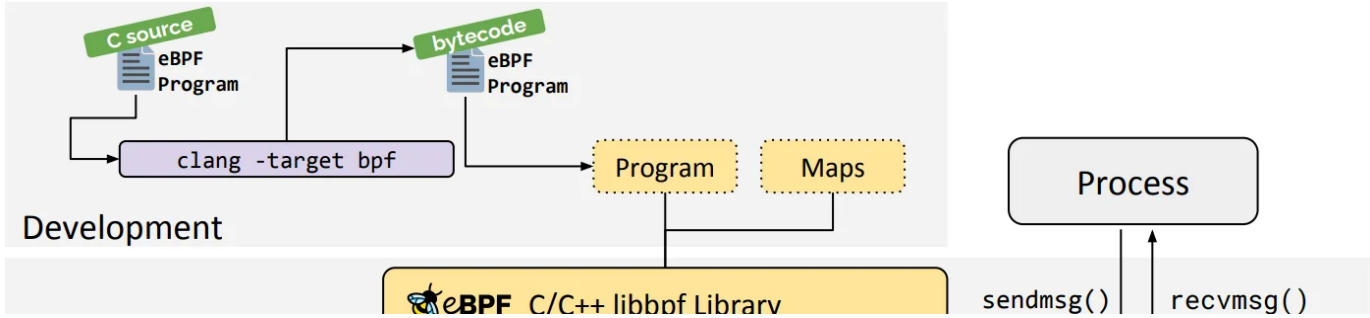
Libreria eBPF Go

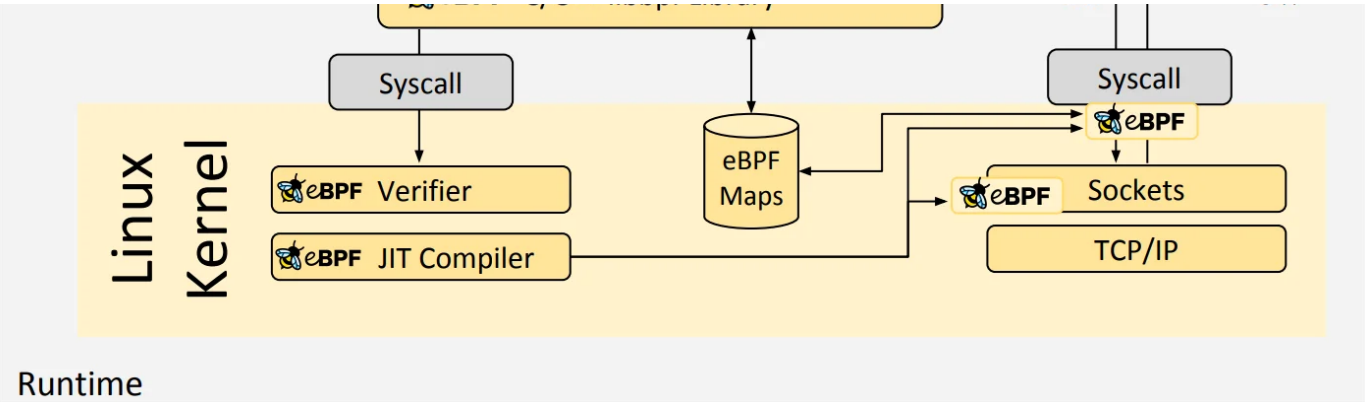
La libreria eBPF per il Go fornisce una libreria generica eBPF che disaccoppia il processo di ottenimento del bytecode unitamente al caricamento e alla gestione dei programmi eBPF. I programmi eBPF sono tipicamente implementati mediante un linguaggio a livello più alto e poi compilati in bytecode per mezzo del compilatore clang/LLVM.



libbpf C/C++ Library

La libreria libbpf è una libreria eBPF generica basata su C/C++, che aiuta a disaccoppiare il caricamento dei file oggetto eBPF generati dal compilatore clang/LLVM, nel kernel e in generale astrae l'interazione con le chiamate di sistema BPF fornendo alle applicazioni delle API facili da usare.





Ulteriori approfondimenti

Se volessi imparare altro su eBPF, prosegui la lettura con il seguente materiale aggiuntivo:

Documentazione

- **eBPF Docs**, Technical documentation for eBPF
- **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
- **eBPF Tutorial: Learning eBPF Step by Step with Examples**, Various authors, 2024

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

Cilium

- **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

Libri

- **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

Articoli & Blog

- **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

Mantenuto dalla community eBPF.

Hai notato un errore? Invia una segnalazione

Per Iniziare

Cos'è eBPF?

eBPF Tech Talks

Casi di Studio

Labs

Panorama dei Progetti

Applicazioni

Infrastruttura

Informazioni

Blog

Acquista Gadget

Marchio

Newsletter

Contribuisci

Italiano