SPIEGAZIONE SVILUPPO PROGRAMMI eBPF IN libbpf-bootstrap

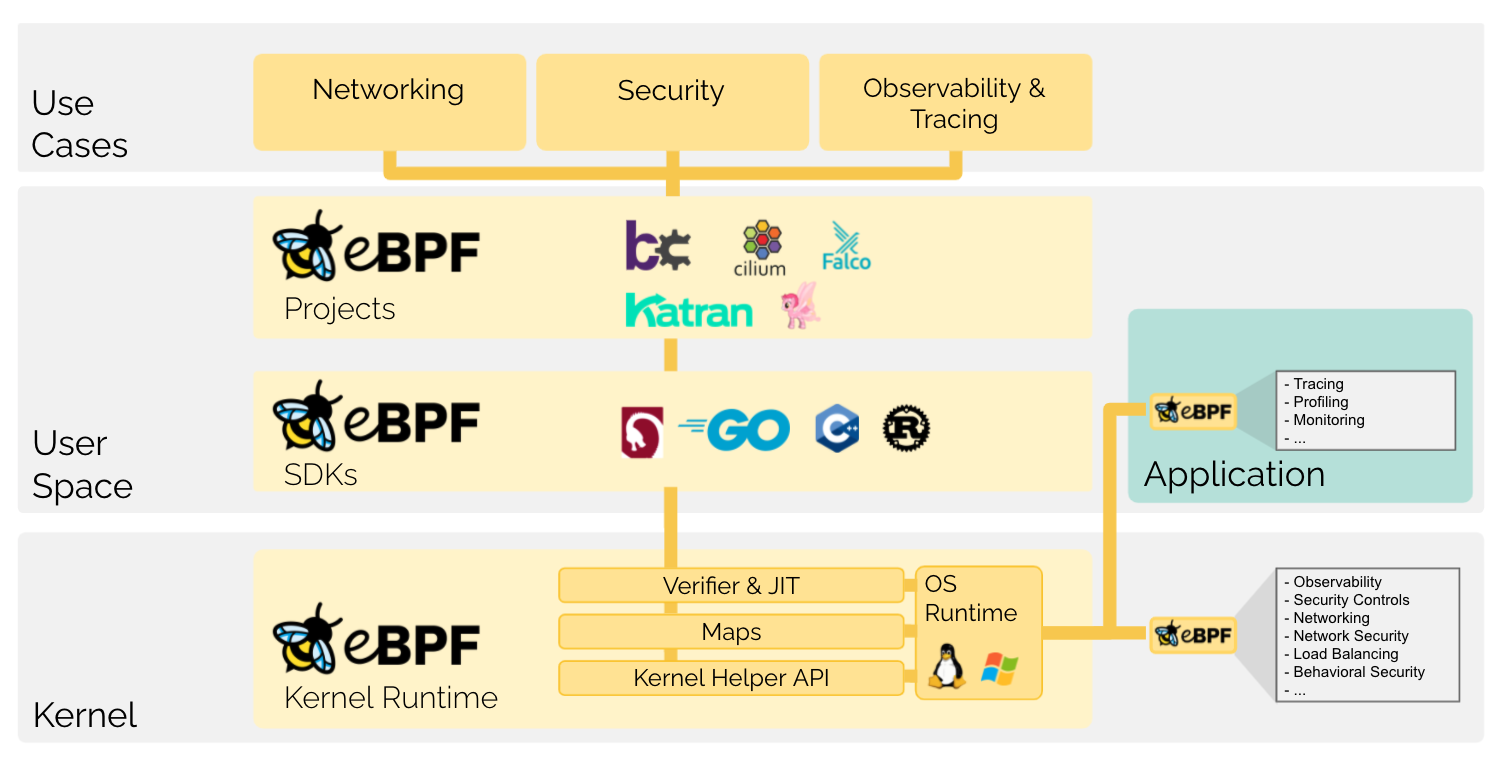
<https://www.sobyte.net/post/2022-07/c-ebpf/#i-introduction-to-ebpf>

INTRODUZIONE

Il predecessore della tecnologia eBPF è BPF (Berkeley Packet Filter), iniziato alla fine del 1992 con un documento intitolato "The BSD PacketFilter: A New Architecture for User-Level Packet Capture" (https://www.tcpdump.org/papers/bpf-usenix93.pdf). Il documento proponeva una soluzione tecnica per implementare il filtraggio dei pacchetti di rete nel kernel Unix, una nuova tecnologia che era 20 volte più veloce della più avanzata tecnologia di filtraggio dei pacchetti dell'epoca.

Nel 1997, la tecnologia BPF è stata incorporata nel kernel Linux e successivamente utilizzata in tcpdump (un tool comune per il debug delle reti di computer che funziona da riga di comando che consente all'utente di intercettare pacchetti e trasmissioni ad esempio nel protocollo TCP/IP condotti attraverso una rete al quale il computer è collegato).

All'inizio del 2014, Alexei Starovoitov ha implementato eBPF, che estende il classico BPF e apre le porte a una gamma più ampia di tecnologie BPF.



Come possiamo vedere dal diagramma sopra: i programmi eBPF vengono eseguiti nello stato del kernel (kernel), non è necessario ricompilare il kernel o compilare e montare i moduli del kernel. eBPF può essere inserito dinamicamente nel kernel ed eseguito e disinstallato in qualsiasi momento. **Una volta nel kernel, eBPF ha una visione dall'alto e può monitorare il kernel così come i programmi sullo stato dell'utente** . E la tecnologia eBPF fornisce una serie di strumenti (Verifier) ​​per rilevare la sicurezza del codice eBPF e impedire a programmi dannosi di entrare nello stato del kernel e di essere eseguiti.

In sostanza, la tecnologia BPF è in realtà un'apertura del kernel per lo stato dell'utente (il kernel ha già raggiunto il “burial point”). Inserendo programmi eBPF e registrando eventi da guardare, attivazione di eventi (**callback** del kernel ai programmi eBPF iniettati) e scambio di dati tra il kernel e lo stato utente per ottenere la logica desiderata.

L'eBPF di oggi non è più limitato alle classiche applicazioni BPF (cBPF) in rete. Alla tecnologia eBPF è stata data una nuova definizione: “a New Generation of Networking, Security, and Observability Tools”, ovvero una nuova generazione di tecnologie di rete, sicurezza e osservabilità. Questa definizione viene dal Chief Open Source Officer di Isovalent Liz Rice. Si tratta della società madre del progetto Cilium, una startup tecnologica che guida il networking nativo del cloud, la sicurezza e l'osservabilità con la tecnologia eBPF.

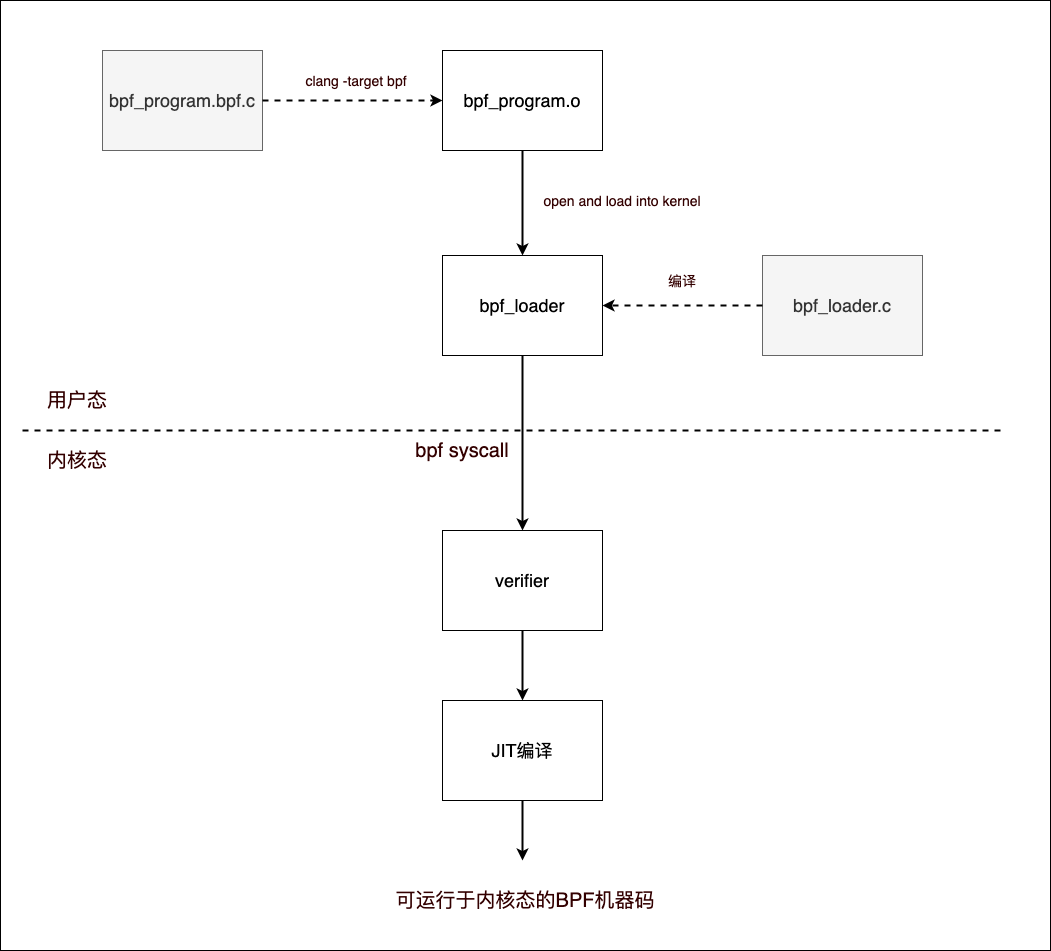
SVILUPPARE UN PROGRAMMA eBPF

1. FORMA

Un progetto finalizzato allo sviluppo di programmi BPF di solito **è costituito da due tipi di file sorgente** , uno è il file del codice sorgente del programma BPF in esecuzione nello stato del kernel (ad esempio, bpf\_program.bpf.c). L'altro è il file del codice sorgente del programma di stato utente utilizzato per caricare il programma BPF nel kernel, scaricare il programma BPF dal kernel, interagire con lo stato del kernel e presentare la logica del programma di stato utente (ad esempio, bpf\_loader. c nella figura seguente).

Attualmente, i programmi BPF in esecuzione nello stato del kernel possono essere sviluppati solo in C (file di codice sorgente bpf\_program.bpf.c), o più precisamente in sintassi C ristretta, e l' **unico** che può compilare perfettamente il codice sorgente C nei file di destinazione BPF è il [compilatore clang](https://clang.llvm.org/) (clang è un front-end del compilatore per C, C++, Objective-C e altri linguaggi di programmazione, che utilizza LLVM come back-end).

Diagramma del processo di compilazione e caricamento del programma BPF nel kernel.



Il file di destinazione BPF (bpf\_program.o) è essenzialmente un file ELF (un formato di file standard comune per file eseguibili , codice oggetto , librerie condivise e core dump).

Un formato di file è un modo standard in cui le informazioni vengono codificate per l'archiviazione in un file di computer . Specifica come vengono utilizzati i bit per codificare le informazioni in un supporto di archiviazione digitale.

 il programma BPF non viene caricato nel kernel come istruzione macchina, ma come codice byte, che è ovviamente una barriera aggiunta alla macchina virtuale BPF per motivi di sicurezza. Quando il programma BPF viene caricato nel kernel, la VM BPF verifica il bytecode BPF ed esegue una compilazione JIT per compilare il bytecode in codice macchina.

I programmi di stato utente utilizzati per caricare e scaricare i programmi BPF possono essere sviluppati in più linguaggi, C o Python, Go, Rust, ecc.

1. SVILUPPO

La BPF si è evoluta nel corso degli anni e, sebbene siano stati compiuti sforzi per migliorarla, l'esperienza nello sviluppo e nella costruzione di programmi BPF non è ancora l'ideale. Per questo motivo la comunità ha anche creato framework e raccolte di librerie come BPF Compiler Collection (BCC) per semplificare lo sviluppo di BPF e librerie come bpftrace che forniscono un linguaggio di sviluppo BPF avanzato (comprensibilmente un linguaggio DSL per lo sviluppo di BPF)

**BCC** è un toolkit per la creazione di programmi efficienti di tracciamento e manipolazione del kernel e include numerosi strumenti ed esempi utili. Fa uso di BPF (Berkeley Packet Filters) esteso, formalmente noto come eBPF, una nuova funzionalità che è stata aggiunta per la prima volta a Linux 3.15. Gran parte di ciò che utilizza BCC richiede Linux 4.1 e versioni successive.

eBPF è stato descritto da Ingo Molnár come:

*Una delle caratteristiche più interessanti di questo ciclo è la possibilità di allegare programmi eBPF (bytecode definito dall'utente, sandboxed eseguito dal kernel) a kprobes. Ciò consente la strumentazione definita dall'utente su un'immagine del kernel live che non può mai arrestarsi in modo anomalo, bloccarsi o interferire negativamente con il kernel.*

BCC rende i programmi BPF più facili da scrivere, con la strumentazione del kernel in C (e include un wrapper C attorno a LLVM) e front-end in Python e lua. È adatto a molte attività, tra cui l'analisi delle prestazioni e il controllo del traffico di rete.

**bpftrace** è un linguaggio di tracciamento di alto livello per il Berkeley Packet Filter (eBPF) potenziato da Linux disponibile nei recenti kernel Linux. bpftrace utilizza LLVM come back-end per compilare script in BPF-bytecode e utilizza BCC per interagire con il sistema Linux BPF, oltre alle funzionalità di tracciamento Linux esistenti: tracciamento dinamico del kernel (kprobes), tracciamento dinamico a livello di utente (uprobes), e tracepoints. Il linguaggio bpftrace si ispira a awk e C e ai traccianti precedenti come DTrace e SystemTap. bpftrace è stato creato da Alastair Robertson .

**Molte** volte non abbiamo bisogno di sviluppare i nostri programmi BPF, progetti open source come bcc e bpftrace ci forniscono molti programmi BPF di alta qualità. Ma una volta che dobbiamo svilupparli noi stessi, la soglia per lo sviluppo basato su bcc e bpftrace in realtà non è bassa. Devi comprendere la struttura del framework bcc e devi imparare il linguaggio di scripting fornito da bpftrace, che inevitabilmente si aggiunge all'onere di sviluppare BPF da solo.

Man mano che la BPF diventa più ampiamente utilizzata, emerge gradualmente la questione della portabilità della BPF. Il kernel Linux si sta evolvendo rapidamente e i tipi e le strutture dati nel kernel cambiano costantemente. I **campi** dello stesso tipo di struttura in diverse versioni del kernel possono essere riorganizzati, possono essere rinominati o eliminati, possono essere modificati in campi completamente diversi, ecc. Per i programmi BPF che non hanno bisogno di esaminare le strutture dati interne del kernel, potrebbero non esserci problemi di portabilità. Tuttavia, per quei programmi BPF che devono fare affidamento su determinati campi nella struttura dati del kernel, è importante considerare i problemi causati al programma BPF dai cambiamenti nella struttura dati interna delle diverse versioni del kernel.

Inizialmente, il modo per risolvere questo problema era compilare il programma BPF localmente sulla macchina di destinazione in cui era distribuito il programma BPF per garantire che il layout del campo del tipo di kernel a cui accedeva il programma BPF fosse coerente con il kernel dell'host di destinazione. Ma questo è ovviamente ingombrante: i vari pacchetti di sviluppo da cui dipende BPF, i compilatori utilizzati devono essere installati sulla macchina di destinazione e il processo di compilazione può richiedere molto tempo, rendendo il processo di test e distribuzione dei programmi BPF molto doloroso, specialmente se usi bcc e bpftrace per sviluppare programmi BPF.

Per risolvere il problema della portabilità BPF, il kernel ha introdotto due nuove tecnologie, **BTF** (BPF Type Format) e **CO-RE** (Compile Once - Run Everywhere). BTF fornisce informazioni strutturali per evitare la dipendenza da Clang e dalle intestazioni del kernel e CO-RE rende rilocabile il bytecode BPF compilato, evitando la necessità di ricompilazione LLVM.

I programmi BPF creati utilizzando queste nuove tecniche funzionano su diverse versioni del kernel Linux senza la necessità di ricompilarlo per un kernel specifico sulla macchina di destinazione. Inoltre, non è necessario installare centinaia di megabyte di dipendenze LLVM, Clang e dell'intestazione del kernel sulla macchina di destinazione come avveniva in precedenza.

Naturalmente queste nuove tecnologie sono trasparenti per il programma BPF stesso. L'API utente libbpf fornita dal codice sorgente del kernel Linux incapsula tutte le nuove tecnologie di cui sopra e fintanto che il caricatore dello stato utente è sviluppato sulla base di libbpf, allora libbpf aiuterà silenziosamente il programma BPF a riposizionarsi nei campi corrispondenti del kernel struttura di cui ha bisogno nel kernel dell'host di destinazione, rendendo libbpf la scelta preferita per lo sviluppo di caricatori BPF.

1. SVILUPPARE PROGRAMMI BPF SULLA BASE DI libbpf

Lo sviluppatore del kernel BPF Andrii Nakryiko ha reso open source un progetto bootstrap libbpf-bootstrap per lo sviluppo di programs e loaders BPF direttamente basati su libbpf (su github /libbpf-bootstrap). Questo progetto contiene esempi di sviluppo di programmi BPF e programmi di stato utente utilizzando c e rust.

Esempio di un programma BPF helloworld e il suo caricatore di stato utente per vedere il "modo" di implementare un programma BPF basato sulla struttura suggerita da libbpf-bootstrap.

Immagine che contiene diagramma

Descrizione generata automaticamente

libbpf si riferisce a tools/lib/bpf nella base di codice del kernel linux, che è una libreria C fornita dal kernel a sviluppatori esterni per la creazione di programmi di stato utente BPF. Gli sviluppatori del kernel bpf hanno creato un repository mirror per libbpf su github.com per comodità degli sviluppatori che utilizzano la libreria libbpf: <https://github.com/libbpf/libbpf> in modo che gli sviluppatori BPF non debbano scaricare l'intera quantità di codice del kernel Linux. Naturalmente, il repository mirror contiene anche alcune delle intestazioni del kernel da cui dipende tools/lib/bpf, che sono mappate sui percorsi sorgente del kernel linux come mostrato nel codice seguente (il lato sinistro del segno di uguale è il percorso sorgente in il kernel Linux, il lato destro del segno uguale è il percorso sorgente in github.com/libbpf/libbpf).

|  |  |
| --- | --- |
|  | // https://github.com/libbpf/libbpf/blob/master/scripts/sync-kernel.sh  PATH\_MAP=( **\**  [tools/lib/bpf]=src **\**  [tools/include/uapi/linux/bpf\_common.h]=include/uapi/linux/bpf\_common.h **\**  [tools/include/uapi/linux/bpf.h]=include/uapi/linux/bpf.h **\**  [tools/include/uapi/linux/btf.h]=include/uapi/linux/btf.h **\**  [tools/include/uapi/linux/if\_link.h]=include/uapi/linux/if\_link.h **\**  [tools/include/uapi/linux/if\_xdp.h]=include/uapi/linux/if\_xdp.h **\**  [tools/include/uapi/linux/netlink.h]=include/uapi/linux/netlink.h **\**  [tools/include/uapi/linux/pkt\_cls.h]=include/uapi/linux/pkt\_cls.h **\**  [tools/include/uapi/linux/pkt\_sched.h]=include/uapi/linux/pkt\_sched.h **\**  [include/uapi/linux/perf\_event.h]=include/uapi/linux/perf\_event.h **\**  [Documentation/bpf/libbpf]=docs **\**  ) |

Il bpftool nella figura corrisponde a tools/bpf/bpftool nel repository del codice del kernel linux, che è anche il repository mirror corrispondente creato su github, un programma helper bpf utilizzato in libbpf-bootstrap per generare xx.skel.h. Il mirror repository contiene anche tools/lib/bpf/. La mappatura tra bpftool e i percorsi sorgente del kernel linux è mostrata nel codice seguente (la parte sinistra del segno uguale è il percorso sorgente nel kernel linux, la parte destra del segno uguale è la sorgente percorso in github.com/libbpf/bpftool)

|  |  |
| --- | --- |
|  | // https://github.com/libbpf/bpftool/blob/master/scripts/sync-kernel.sh  PATH\_MAP=( **\**  [${BPFTOOL\_SRC\_DIR}]=src **\**  [${BPFTOOL\_SRC\_DIR}/bash-completion]=bash-completion **\**  [${BPFTOOL\_SRC\_DIR}/Documentation]=docs **\**  [kernel/bpf/disasm.c]=src/kernel/bpf/disasm.c **\**  [kernel/bpf/disasm.h]=src/kernel/bpf/disasm.h **\**  [tools/include/uapi/asm-generic/bitsperlong.h]=include/uapi/asm-generic/bitsperlong.h **\**  [tools/include/uapi/linux/bpf\_common.h]=include/uapi/linux/bpf\_common.h **\**  [tools/include/uapi/linux/bpf.h]=include/uapi/linux/bpf.h **\**  [tools/include/uapi/linux/btf.h]=include/uapi/linux/btf.h **\**  [tools/include/uapi/linux/const.h]=include/uapi/linux/const.h **\**  [tools/include/uapi/linux/if\_link.h]=include/uapi/linux/if\_link.h **\**  [tools/include/uapi/linux/netlink.h]=include/uapi/linux/netlink.h **\**  [tools/include/uapi/linux/perf\_event.h]=include/uapi/linux/perf\_event.h **\**  [tools/include/uapi/linux/pkt\_cls.h]=include/uapi/linux/pkt\_cls.h **\**  [tools/include/uapi/linux/pkt\_sched.h]=include/uapi/linux/pkt\_sched.h **\**  [tools/include/uapi/linux/tc\_act/tc\_bpf.h]=include/uapi/linux/tc\_act/tc\_bpf.h **\**  ) |

**helloworld.bpf.c** è il codice sorgente del programma bpf, che viene compilato nel file bytecode ELF BPF **helloworld.bpf.o** da clang -target=bpf . libbpf-bootstrap non utilizza il user state loader per caricare direttamente helloworld.bpf.o, ma genera invece il file **helloworld.skel.h** basato su helloworld.bpf.o tramite il comando bpftool gen. Il file helloworld.skel.h generato **contiene il bytecode del programma BPF** e le funzioni per caricare e scaricare il corrispondente programma BPF, che possiamo richiamare direttamente dal programma user state.

**helloworld.c** è il programma di stato utente BPF, deve solo includere helloworld.skel.h e load e hook il programma BPF al buried point corrispondente nel livello del kernel come da set. Poiché il programma BPF è incorporato nel programma stato utente, si ha solo bisogno di distribuire il programma stato utente quando si distribuisce il programma BPF.