[The eBPF Runtime in the Linux Kernel]

1 - Introduzione

Extended Berkeley Packet Filter (eBPF) è un ambiente di runtime che consente agli utenti di caricare programmi all'interno del kernel del sistema operativo ed eseguirli in modo sicuro ed efficiente su specifici hook del kernel.

Ogni programma passa attraverso un *verificatore*, che valuta le garanzie di sicurezza necessarie per la sua esecuzione. A differenza dell'approccio più comune che prevede di aggirare o sostituire interamente il kernel, eBPF offre agli utenti la flessibilità di modificare il kernel in tempo reale, sperimentare e iterare rapidamente, e distribuire soluzioni che soddisfano i requisiti specifici del carico di lavoro, collaborando con il kernel piuttosto che sostituirlo.

eBPF (*Extended Berkeley Packet Filter*) abilita la capacità di programmazione dinamica direttamente nei kernel monolitici.
(I kernel monolitici sono suddivisi in tre aree principali: User, Kernel e Hardware. Esempi: Windows e Linux). eBPF funziona come una macchina virtuale programmabile sicura, eseguita su un runtime ad alte prestazioni all'interno del kernel. Consente agli utenti di scrivere programmi, caricarli nel kernel e associarli a specifici hook per iniziare l'esecuzione. Per garantire la sicurezza, ogni programma viene sottoposto a un'analisi statica attraverso un *verificatore* al momento del caricamento. Per garantire le prestazioni, tutti i programmi vengono compilati *Just-in-time (JIT)* in istruzioni macchina native.

2 - Sfide nella personalizzazione del Kernel

Gli sviluppatori spesso incontrano ostacoli significativi quando devono apportare modifiche dirette al kernel di Linux. Di seguito sono riportate alcune delle principali sfide.

Modificare il Kernel
Linux offre un'enorme quantità di opzioni di configurazione (knobs), sia durante la compilazione che a runtime. Tuttavia, queste opzioni sono adatte per regolare i parametri delle politiche esistenti del kernel, ma non sono sufficientemente espressive per codificare nuove politiche. In tali casi, quando la personalizzazione del kernel è necessaria, gli sviluppatori devono apportare modifiche al codice del kernel e/o scrivere moduli del kernel. Questo però rende il testing e il debugging delle modifiche molto più complessi, poiché intervenire sul kernel richiede una conoscenza approfondita della sua complessa codebase, comportando un enorme onere di manutenzione.

A meno che tali modifiche non vengano accettate dalla comunità del kernel, è necessario eseguire il *forward porting* su eventuali aggiornamenti del kernel a causa delle API instabili. Qualsiasi bug non rilevato nel codice potrebbe facilmente causare crash del sistema e mettere fuori uso server di produzione, traducendosi direttamente in tempi di inattività e perdita di entrate.

Distribuire le modifiche al kernel
Distribuire una modifica al kernel, anche minima, su un insieme di server è un processo lungo. Sostituire il kernel in esecuzione con uno nuovo comporta l'interruzione dei carichi di lavoro ospitati sulla macchina, poiché devono essere arrestati. La macchina deve quindi avviarsi con il nuovo kernel e reinizializzare tutti i servizi. Questo ciclo è particolarmente dannoso per i carichi di lavoro che comportano *cold starts* e hanno un significativo tempo di avvio.

Dopo tutti questi passaggi, è necessario eseguire una fase di test estensivi per verificare e qualificare queste modifiche, correggendo eventuali regressioni rilevate. Questo processo deve essere ripetuto per ogni server che riceve un aggiornamento del kernel, rendendo il processo costoso e prolungando il tempo totale per il rilascio di un nuovo kernel sulla flotta, che può richiedere settimane o mesi.

Approcci alternativi e limitazioni
Come già accennato, gli sviluppatori affrontano complessità quando apportano modifiche al kernel. Di seguito discutiamo metodi alternativi che promettono benefici in termini di prestazioni, ma che presentano anche limitazioni e sfide di manutenzione.

Aggirare o sostituire completamente il kernel
Sebbene i framework di *kernel bypass* e i sistemi operativi a libreria (*non monolitici*) specializzati per un determinato carico di lavoro siano attraenti in termini di prestazioni, comportano anche degli svantaggi. Le soluzioni di *kernel bypass* richiedono il completo controllo dell'hardware e sprecano cicli della CPU in *busy polling*.
I sistemi operativi a libreria sono adattati per carichi di lavoro specifici e richiedono modifiche al codice per adattarsi a requisiti differenti. Inoltre, entrambi questi approcci rendono difficile o impossibile per due carichi di lavoro coesistere e condividere risorse hardware, il che non è accettabile per alcuni utenti. Infine, richiedono la riscrittura della logica dell'applicazione, il che rappresenta un notevole onere di manutenzione.

eBPF soddisfa questi requisiti consentendo la personalizzazione efficiente del kernel Linux in modo sicuro e dinamico a runtime, riducendo il rischio di introdurre bug nel kernel quando si apportano modifiche e accelerando la velocità di sviluppo e distribuzione.
Prima dell'introduzione di eBPF, il kernel disponeva di più macchine virtuali basate su registri specifiche per dominio, che servivano a casi d'uso dedicati nel sottosistema di rete, ma nessuna di queste era progettata per essere di uso generale.

Principi di progettazione

Personalizzazione sicura e dinamica del sistema operativo
Ospitare un ambiente di runtime di macchina virtuale sicura ed efficiente all'interno del kernel consente di personalizzare dinamicamente il kernel, raggiungendo prestazioni simili a quelle del codice già presente nel kernel, garantendo al contempo che l'integrità del kernel non venga compromessa in nessuna circostanza. La garanzia di sicurezza aumenta la fiducia quando tali programmi vengono implementati nel kernel, rispetto alle modifiche dirette al codice del kernel o ai moduli del kernel, riducendo così il rischio di introdurre bug che potrebbero causare interruzioni del servizio.

Distribuzione e aggiornamenti rapidi
Modificare il kernel e distribuirlo su una vasta flotta di server è un processo molto costoso, poiché il riavvio di una macchina comporta l'interruzione e la

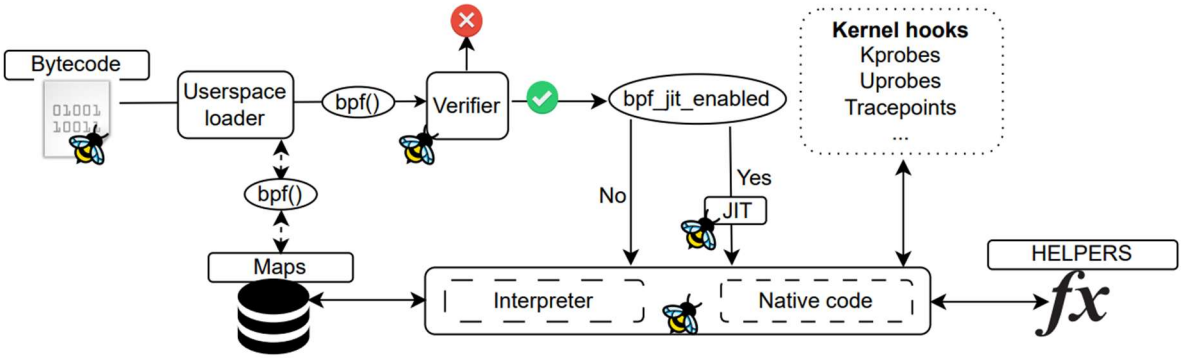
re-inizializzazione dei carichi di lavoro. Caricare dinamicamente i programmi nel kernel a runtime consente una distribuzione rapida su una flotta senza interruzioni del servizio. Allo stesso tempo, la risoluzione dei bug e la reiterazione basata sul feedback raccolto tramite la telemetria diventa molto più rapida, permettendo di scaricare e ricaricare il programma a runtime. Ciò semplifica notevolmente la distribuzione, accelera il ciclo di feedback per il test e la qualificazione delle modifiche al kernel e velocizza lo sviluppo.

Integrazione con il kernel
Sebbene i programmi si colleghino al kernel per introdurre comportamenti alternativi, dovrebbero poter interagire in sicurezza con lo stato del kernel esistente, e manipolarlo se necessario. Allo stesso tempo, i programmi dovrebbero avere la possibilità di ritornare al normale processamento del kernel nel caso in cui non abbiano nulla di rilevante da fare. Questo modello offre all'utente flessibilità, consentendo di utilizzare l'implementazione del kernel esistente se necessario, senza dover reimplementare codice simile all'interno del programma.

3 - Panoramica

eBPF è definita come una macchina virtuale astratta che supporta il set di istruzioni eBPF. La macchina virtuale ha un set di 11 registri e uno stack di dimensioni fisse. Un programma eBPF opera all'interno di un ambiente di macchina virtuale limitato fornito dal kernel Linux. Il set di istruzioni eBPF è una raccolta piccola ma versatile di istruzioni a 64 bit. Queste istruzioni offrono un'ampia gamma di funzionalità, permettendo ai programmi eBPF di eseguire in modo efficiente diverse operazioni nello spazio del kernel.

Il set di istruzioni supporta operazioni aritmetiche come addizione, sottrazione, moltiplicazione e operazioni bitwise (ad esempio: AND, OR, XOR). Inoltre, supporta istruzioni di caricamento e memorizzazione, oltre a istruzioni di salto che includono sia salti condizionali che incondizionati per alterare il flusso del programma, così come chiamate di funzione e uscite. Il set di istruzioni include anche operazioni atomiche progettate per garantire un accesso sicuro alla memoria e la sua modifica. Queste operazioni atomiche assicurano che l'accesso concorrente non porti a incoerenze o corruzione dei dati.



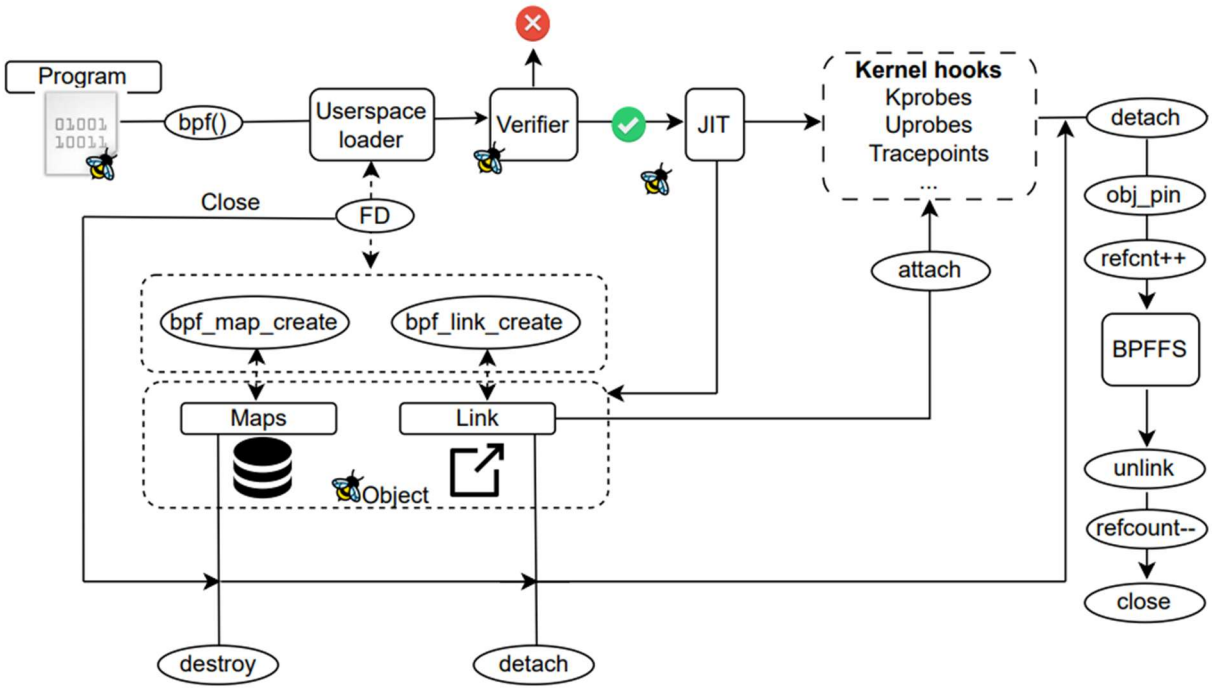
Runtime di eBPF

- **Bytecode di eBPF**
  Il bytecode eBPF è definito come una sequenza finita di istruzioni eBPF. La macchina virtuale eBPF esegue i programmi eBPF, che sono codificati utilizzando bytecode eBPF. Ogni programma è composto da uno o più sottoprogrammi (o *subprogs*). Questi sono semplicemente unità indipendenti di bytecode analoghe alle funzioni. L'esecuzione di un programma inizia dal sottoprogramma principale.
- **Loader in Userspace di eBPF**
  eBPF dispone di un ampio ecosistema di *loader* in userspace che caricano il bytecode eBPF nel kernel utilizzando la chiamata di sistema BPF_PROG_LOAD, e collegano il programma agli hook rilevanti gestendo eventuali mappe associate. (Loader in userspace: BCC, Bptrace e libbpf). Viene restituito un file descriptor che rappresenta il programma caricato, il quale può quindi essere utilizzato per collegare il programma agli specifici hook del kernel. (Considereremo la toolchain LLVM, il suo frontend C e libbpf).
- **Verificatore eBPF**
  Il *verificatore*, un componente cruciale nei sistemi eBPF, ispeziona il bytecode prima che venga accettato nel kernel, garantendo le proprietà di sicurezza associate a eBPF e assicurando che il caricamento dei programmi non comprometta l'integrità e la sicurezza del kernel in nessuna circostanza.
- **JIT Compiler e Interprete di eBPF**
  Al termine del processo di verifica, il programma viene compilato in istruzioni macchina native utilizzando il compilatore Just-In-Time (JIT). Nei casi in cui il JIT sia disabilitato o non supportato, l'interprete eBPF assume la responsabilità dell'esecuzione del programma, decodificando ed eseguendo dinamicamente il bytecode in tempo reale.
- **Hook di eBPF**
  Il flusso del programma eBPF è determinato dagli eventi, che vengono eseguiti quando il kernel o un'applicazione incontrano specifici *hook points*. Questi punti di aggancio predefiniti sono collocati in varie parti del kernel e coprono una vasta gamma di eventi, inclusi le chiamate di sistema, l'entrata e l'uscita delle funzioni, i socket di rete, i tracepoint, e molto altro. In base al tipo di collegamento e/o al tipo di programma, il programma viene collegato all'hook dove deve essere eseguito. Se un hook predefinito non soddisfa le necessità, gli sviluppatori possono creare punti di aggancio personalizzati chiamati kernel probes (*kprobes*) o user probes (*uprobes*). Queste probe consentono di collegare i programmi eBPF a quasi qualsiasi punto all'interno del kernel o delle applicazioni userspace.
- **Tipi di programmi eBPF**
  Il BPF_PROG_TYPE è la categorizzazione di un programma eBPF che ne determina la funzione, i parametri di input, le azioni accettabili e i punti di collegamento nel kernel. Ogni tipo di programma ha caratteristiche che definiscono il suo comportamento e la sua interazione con il sistema. Ad esempio, il tipo di programma *socket filter* (BPF_PROG_TYPE_SOCKET_FILTER) è progettato per esaminare e gestire i pacchetti di

rete a livello di socket. Gli sviluppatori possono progettare logiche personalizzate all'interno di questi programmi per analizzare e modificare i pacchetti in ingresso e in uscita secondo necessità. Al contrario, i tipi di programma di tracciamento (BPF_PROG_TYPE_TRACING) sono in grado di monitorare eventi del kernel e fornire informazioni preziose sul funzionamento del sistema. Questi tipi di programmi e i loro punti di collegamento sono definiti nel codebase del kernel e fungono da modelli per la creazione di programmi eBPF per soddisfare diversi requisiti.

- **Helper di eBPF**
  Gli helper di eBPF sono funzioni specializzate accessibili ai programmi eBPF, che consentono l'interazione con il sistema e il contesto di esecuzione. Questi helper facilitano compiti come la stampa di messaggi di debug, la manipolazione dei pacchetti di rete e l'interazione con le mappe eBPF. Ogni tipo di programma eBPF può accedere a un sottoinsieme specifico di questi helper, adeguato al contesto e ai requisiti del programma.
- **Mappe di eBPF**
  Una mappa eBPF è una struttura dati astratta di un certo tipo, come un array o una hash map, che facilita lo scambio di dati tra lo userspace e il kernel. I programmi che vengono eseguiti all'interno della macchina virtuale eBPF possono accedere a una o più mappe tramite istruzioni di caricamento specifiche per la piattaforma.

-Oggetti eBPF e il loro ciclo di vita



Ogni oggetto eBPF ha una rappresentazione per la gestione del programma eBPF all'interno del kernel, ed è esposto allo spazio utente tramite un *file descriptor*. Il ciclo di vita dell'oggetto eBPF è legato al ciclo di vita del *file descriptor*. Una volta che l'ultimo file descriptor corrispondente all'oggetto eBPF viene rilasciato, anche il suo stato all'interno del kernel viene rilasciato. Per abilitare la persistenza oltre la durata di vita di un processo, il kernel consente di *pinnare* questi file descriptor su uno pseudo-file system speciale chiamato **bpffs**. Ogni operazione di pinning prende un riferimento sull'oggetto eBPF, estendendone quindi il ciclo di vita.

- **Programmi eBPF**
  Questi oggetti rappresentano il programma effettivo che viene caricato nel kernel. Un file descriptor che rappresenta il programma viene restituito dal comando BPF_PROG_LOAD della chiamata di sistema bpf, dopo che il verificatore eBPF ha verificato e compilato JIT il programma e ha creato la sua rappresentazione interna nel kernel. Una volta caricato e verificato, il programma è pronto per essere collegato a un hook del kernel designato.
- **Mappe eBPF**
  Quando si creano programmi eBPF, le mappe vengono definite con il comando BPF_MAP_CREATE della system call bpf, che restituisce un file descriptor. Questo descrittore viene usato nel programma eBPF per fare riferimento alla mappa attraverso specifiche istruzioni. Durante la verifica del programma, il verificatore converte il file descriptor nella mappa effettiva presente nel kernel e considera il registro collegato a queste istruzioni come un puntatore alla mappa eBPF nel programma.
- **Link eBPF**
  I link eBPF vengono creati utilizzando il comando BPF_LINK_CREATE della chiamata di sistema bpf. Invece di collegarsi direttamente a un hook, la creazione di un link eBPF associa il ciclo di vita del collegamento del programma a un file descriptor, semplificando la gestione dei riferimenti del programma e mantenendo attiva la probe, anche se l'applicazione che la carica termina inaspettatamente. Il file descriptor associato a un link eBPF ne controlla il ciclo di vita. Quando l'ultimo file descriptor viene chiuso, il link stacca il programma dall'hook del kernel, consentendo la pulizia delle risorse. Solo il proprietario del link può staccarlo o aggiornarlo, garantendo l'integrità del sistema e prevenendo modifiche non autorizzate.

- **BTF**
  BPF Type Format (BTF) è un formato di informazioni di debug specificamente progettato per l'uso con i programmi eBPF. Viene generato dal compilatore quando il kernel o un programma eBPF viene compilato. Questo formato contiene informazioni sui tipi di dati C (*come strutture e funzioni*) e fornisce annotazioni specifiche per il contesto che aiutano il verificatore eBPF a garantire la sicurezza e la correttezza dei programmi eBPF.

Ma perché è importante?

1. Ottimizzazione della memoria: Il formato di debug standard, DWARF, consuma molta memoria quando viene incorporato nel kernel. BTF, d'altra parte, ha un'impronta di memoria molto più piccola grazie a un algoritmo che elimina le ridondanze, rendendolo più efficiente e adatto per essere incluso nel kernel senza causare problemi di memoria.
2. Miglior introspezione e debugging: BTF consente al verificatore eBPF di comprendere meglio il codice e di eseguire un'analisi più approfondita. Facilita anche il debugging, stampando informazioni utili come la linea esatta di codice che causa errori. Questo aiuta gli sviluppatori a correggere rapidamente i problemi nei loro programmi.
3. CO-RE (Compile Once, Run Everywhere): BTF consente ai programmi eBPF di essere portabili su kernel e architetture diversi senza la necessità di ricompilazione. Le modifiche necessarie per adattarsi a diversi ambienti vengono gestite dinamicamente dal verificatore o da *libbpf* (una libreria per lavorare con eBPF), rendendo il codice eBPF più flessibile e adattabile.

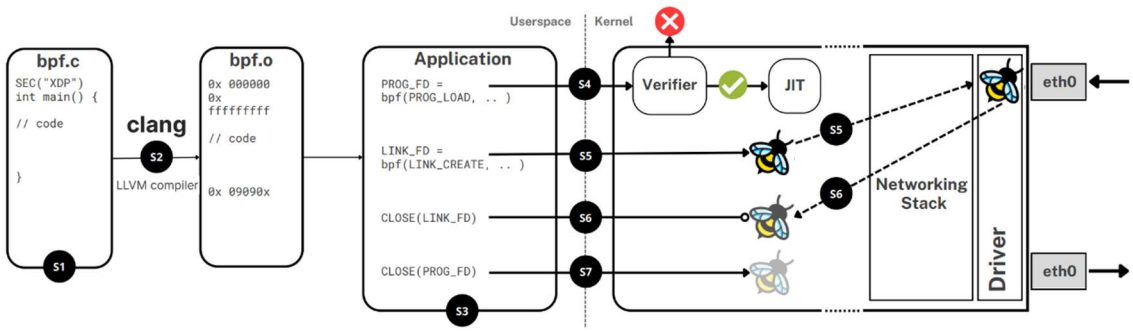
-Set istruzioni eBPF

Il set di istruzioni eBPF è definito in termini della macchina virtuale eBPF. Supporta due tipi di codifica delle istruzioni (64-bit e 128-bit), istruzioni generali (come operazioni aritmetiche, salto, chiamata, caricamento e memorizzazione), e l'indirizzamento di un set di 11 registri a 64-bit (r0-r10) con una convenzione di chiamata ben definita, in cui il registro r10 è di sola lettura e punta alla cima dello stack. Un principio chiave adottato durante la progettazione del set di istruzioni è il mantenimento di una quasi-equivalenza con le architetture dei set di istruzioni (ISA) dell'hardware reale, il che semplifica l'implementazione degli interpreti e dei compilatori JIT.

Inoltre, questa quasi-equivalenza consente ai back-end dei compilatori ottimizzanti di emettere codice assembly eBPF con prestazioni vicine a quelle dei programmi compilati nativamente. Questo perché i compilatori JIT possono tradurre le istruzioni eBPF in istruzioni macchina native principalmente con una corrispondenza uno-a-uno, senza introdurre ulteriori strumentazioni per gestire la traduzione. Uno dei punti di forza della separazione tra il set di istruzioni e il runtime eBPF nel kernel Linux è la possibilità di utilizzarlo anche al di fuori del sistema operativo. L'ISA eBPF può essere supportato direttamente dall'hardware o tradotto all'architettura target dell'hardware. Questo ha anche un grande potenziale per l'hardware computazionale, poiché il runtime eBPF può controllare e decidere se eseguire l'offloading, mentre i programmi per un determinato hook possono essere scritti in modo trasparente rispetto al contesto dell'hardware.

-Tipo di formato BPF

4 - Workflow



In questa sezione, illustriamo il modello di programmazione ed esecuzione ad alto livello di eBPF. La **Figura 3** illustra l'intera sequenza coinvolta nel processo di scrittura ed esecuzione di un programma eBPF per il nostro esempio scelto, dall'inizio alla fine.

Tipicamente, l'utente inizia al passo **S1** scrivendo un programma eBPF in un linguaggio di programmazione ad alto livello.

```
SEC("xdp")
int bpf_program(struct xdp_md *ctx)
{
  void *data_end = (void *)(long)ctx->
      data_end;
  void *data = (void *)(long)ctx->data;
  struct ethhdr *eth = data;

  if (eth + 1 < data_end) {
    if (eth->h_proto == bpf_htons(ETH_P_IP))
        {
      struct iphdr *iph = (void *)(eth + 1);
      if (iph + 1 < data_end && iph->
          protocol == IPPROTO_UDP)
        return XDP_DROP;
    }
  }
  return XDP_PASS;
}
```

*Per il nostro esempio, consideriamo il programma in C presente, scritto per l'hook **XDP**, che invoca il programma per i pacchetti di rete a livello del driver del dispositivo di rete, prima che vengano elaborati dallo stack di rete. L'argomento ctx di tipo struct xdp_md rappresenta il pacchetto di rete grezzo a cui il programma accede. Le variabili puntatore data e data_end puntano rispettivamente all'inizio e alla fine (o subito oltre) dell'area dati del pacchetto di rete. I rami condizionali che confrontano data e data_end vengono utilizzati per garantire che ci sia abbastanza spazio, assicurando che gli accessi alla memoria dei dati del pacchetto siano sicuri.*

Il prossimo passo, **S2**, riguarda la compilazione di questo programma C utilizzando il compilatore **clang** della toolchain LLVM in un file oggetto. Il target per la compilazione viene impostato su **bpf**, che istruisce il compilatore a usare il backend eBPF per produrre il codice binario nel file oggetto generato.

Il passo **S3** si occupa dell'elaborazione del file oggetto prodotto e dell'invio del programma codificato al suo interno al kernel tramite la chiamata di sistema bpf(2) per il caricamento.

*Per il nostro esempio, utilizziamo lo strumento in userspace **bpftool**, che a sua volta usa la libreria **libbpf** per eseguire il caricamento del programma.*

Una volta che il file oggetto è stato elaborato e il programma estratto, il passo **S4** lo invia al kernel utilizzando il comando BPF_PROG_LOAD della chiamata di sistema bpf(2), che invoca il verificatore eBPF. Il verificatore eBPF esegue quindi la verifica del programma per determinare se può essere eseguito in sicurezza all'interno del kernel. Se il verificatore non riesce a determinare la sicurezza del programma, lo rifiuta e restituisce un errore allo

spazio utente. Altrimenti, se la verifica ha successo, il programma viene compilato Just-In-Time (JIT) e viene restituito un file descriptor corrispondente al programma eBPF allo spazio utente.

Una volta che lo spazio utente ha ottenuto il file descriptor per il programma, esso può essere collegato all'hook **XDP** del dispositivo di rete. Per il nostro esempio, il dispositivo di rete è **eth0**, e il passo **S5** prevede l'invocazione del comando BPF_LINK_CREATE della chiamata di sistema bpf(2) per collegare il programma al dispositivo di rete. Se tutti i parametri del comando sono validi, il kernel restituisce un file descriptor corrispondente al link eBPF allo spazio utente. A questo punto, il programma eBPF è collegato all'interfaccia di rete **eth0** e rifiuta tutto il traffico IPv4 UDP in entrata. Viene invocato per ogni pacchetto di rete grezzo ricevuto dal driver di rete del kernel e lo elabora. Il resto del traffico passa allo stack di rete del kernel.

Nel passo **S6**, una volta che l'applicazione userspace chiude il file descriptor del link eBPF, il programma viene scollegato dall'interfaccia di rete.

Una volta che il file descriptor del programma eBPF viene chiuso nel passo **S7**, il kernel libererà le risorse che occupava, come la memoria per il codice del programma.

5 - Sicurezza di un programma eBPF

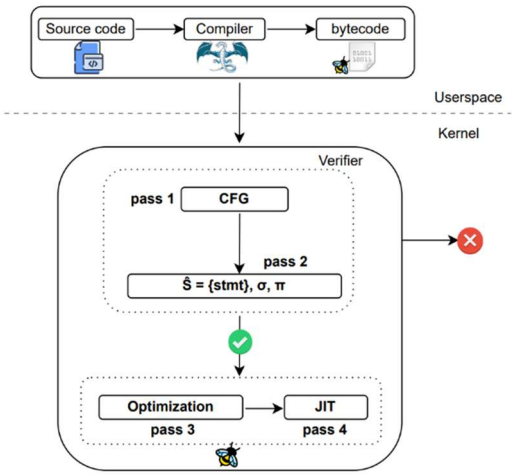
La sicurezza dei programmi è un aspetto critico dei programmi eBPF, garantendo che vengano eseguiti correttamente e in sicurezza, senza compromettere la stabilità e la sicurezza del kernel Linux. Nel contesto dei programmi eBPF, la sicurezza del programma si riferisce a un insieme di proprietà che devono essere soddisfatte per proteggere l'integrità del runtime del kernel e rispettare gli invarianti del contesto del kernel in cui il programma eBPF verrà eseguito. I programmi che violano una di queste proprietà di sicurezza devono essere rifiutati quando vengono caricati nel kernel.

Proprietà di Sicurezza

1. **Sicurezza della memoria**: Il verificatore garantisce che non si verifichino accessi a memoria fuori dai limiti, non validi o *use-after-free*. Traccia i limiti della memoria e proibisce l'accesso alla memoria una volta liberata o l'uso di valori arbitrari come puntatori.
2. **Sicurezza dei tipi**: Il verificatore mantiene una stretta conoscenza dei tipi per i registri e gli oggetti nello stack, prevenendo confusione tra tipi e garantendo l'uso sicuro della memoria del kernel. Utilizza inoltre il **BTF** per far rispettare i limiti dei tipi, specialmente per strutture dati complesse come le strutture (*structs*).
3. **Sicurezza delle risorse**: Prima che un programma termini, il verificatore si assicura che tutta la memoria allocata, i lock e i riferimenti agli oggetti del kernel siano correttamente rilasciati per prevenire perdite di risorse.
4. **Prevenzione delle fughe di informazioni**: Il verificatore impedisce la fuga di informazioni dal kernel allo spazio utente analizzando i puntatori e assicurandosi che non facciano riferimento alla memoria del kernel. Rifiuta anche i tentativi di accesso a regioni non inizializzate dello stack.
5. **Assenza di corse di dati**: Il verificatore garantisce che lo stato del kernel venga manipolato in modo sicuro con una corretta sincronizzazione, ma non verifica la presenza di corse di dati nella memoria gestita dai programmi eBPF (ad esempio, i valori nelle mappe eBPF).
6. **Terminazione**: Il verificatore impone un limite al numero di istruzioni che un programma può eseguire, rifiutando programmi con potenziali loop infiniti o loop non limitati senza una dimostrata terminazione.
7. **Assenza di deadlock**: Per evitare i deadlock, il verificatore vieta ai programmi di mantenere più di un lock contemporaneamente.
8. **Invarianti del contesto di esecuzione**: Il verificatore garantisce che il programma rispetti gli invarianti del contesto di esecuzione del kernel, prevenendo qualsiasi violazione delle assunzioni fatte dal codice del kernel.

Tuttavia, è importante comprendere che questi standard non sono fissi; evolvono costantemente per mantenere l'integrità e la sicurezza dei programmi eBPF nel kernel. Verranno aggiornati regolarmente man mano che l'ecosistema cambia e si sviluppano nuovi metodi, riflettendo i progressi continui in quest'area.

-Il Verificatore eBPF



Il **verificatore eBPF** è un analizzatore statico che svolge un ruolo cruciale nell'assicurare la sicurezza dei programmi eBPF prima che vengano caricati nel kernel. La sua funzione principale è verificare che il programma sia sicuro da eseguire nel contesto del kernel. Una volta confermata la sicurezza, il verificatore invia il bytecode per la compilazione JIT (*Just-In-Time*), convertendolo in istruzioni macchina native.

Il processo di verifica si compone di quattro passaggi chiave, che garantiscono sia la sicurezza che l'efficienza dei programmi eBPF eseguiti nel kernel:

1. Validazione del **grafo di controllo del flusso**.
2. Esecuzione simbolica esaustiva per garantire la sicurezza del programma.
3. Ottimizzazioni e trasformazioni del programma.
4. Compilazione JIT per generare codice macchina nativo.

6 - Validazione del grafo di controllo del flusso

Il **verificatore eBPF** analizza il **grafo del flusso di controllo** (CFG) del programma, in cui ogni istruzione è rappresentata come un nodo e le connessioni di flusso come archi. Ogni istruzione, tranne la prima, ha almeno un collegamento in ingresso, mentre tutte, tranne l'ultima (EXIT), hanno un collegamento in uscita verso l'istruzione successiva. Per le istruzioni di salto (JUMP) e chiamata (CALL), il flusso si divide in base alle condizioni o alla funzione chiamata.

Il verificatore utilizza un algoritmo di ricerca in profondità per percorrere il grafo e etichetta le istruzioni raggiungibili, verificando che non ci siano **istruzioni non raggiungibili** o **cicli infiniti**. Inoltre, garantisce che ogni sottoprogramma termini correttamente con un'istruzione EXIT o JUMP, evitando che il flusso passi inavvertitamente a un altro sottoprogramma.

Per ottimizzare l'elaborazione dei programmi complessi, il verificatore utilizza una tecnica chiamata **state pruning** (potatura degli stati). Questa tecnica riduce il numero di percorsi da esplorare, confrontando lo stato attuale del programma con stati già verificati in punti specifici (detti **pruning points**). Se uno stato equivalente è già stato verificato come sicuro, il verificatore interrompe l'esplorazione del percorso corrente, evitando di eseguire calcoli inutili.

7 - Esecuzione Simbolica

Il verificatore eBPF esegue un'**esecuzione simbolica** del bytecode, esplorando tutti i percorsi possibili del programma per garantirne la sicurezza. Durante l'analisi, tiene traccia dello **stato dello stack** e dei **registri**, utilizzando una struttura simbolica che rappresenta i valori concreti o astratti. Quando il programma incontra istruzioni di salto condizionale, il verificatore esamina entrambi i rami possibili, gestendo l'**esplosione dei percorsi** attraverso una tecnica di **potatura dello stato**, che evita di analizzare percorsi ridondanti.

Il verificatore controlla vari aspetti, tra cui:

- **Sicurezza della memoria**, garantendo accessi corretti e allineamento dei puntatori.
- **Precisione dei valori scalari**, specialmente nei salti condizionali e nelle chiamate a funzioni helper.
- **Gestione delle risorse**, assicurandosi che tutte le risorse (come memoria e riferimenti) vengano correttamente rilasciate prima della fine del programma.
- **Cicli**: analizza cicli con limiti noti attraverso lo srotolamento (unrolling), mentre per cicli con limiti sconosciuti si affida a specifici helper.

Infine, il verificatore gestisce risorse simboliche come **spin lock** e altri oggetti del kernel, evitando deadlock e verificando che ogni risorsa venga rilasciata correttamente prima della terminazione del programma.

8 – Ottimizzazione dopo la verifica

Dopo la verifica della sicurezza di un programma eBPF attraverso l'esecuzione simbolica, si procede con ottimizzazioni e correzioni specifiche. Queste vengono effettuate su ogni programma come parte del processo di ottimizzazione e correzione, che segue la fase di verifica.

-Eliminazione DeadCode

Il verificatore durante l'esecuzione simbolica non esplora i rami non percorsi delle istruzioni di salto condizionato se riesce a determinare, analizzando il flusso dei dati del programma, che un dato ramo non verrà mai preso. Ogni istruzione simulata almeno una volta viene etichettata come "vista". Se il verificatore conclude che un'istruzione non è raggiungibile a runtime, può eliminarla in sicurezza come codice morto.

Alcune condizioni nel programma possono essere risolte solo durante la fase di verifica, ad esempio i valori delle opzioni di configurazione del kernel su cui viene caricato il programma. Per questo motivo, il verificatore eBPF può eliminare il codice morto in modo più aggressivo rispetto al compilatore, grazie alle informazioni più ricche sul flusso dei dati di cui dispone. Questa ottimizzazione è sfruttata per migliorare la portabilità dei programmi eBPF tra diverse versioni del kernel, dove percorsi di codice differenti vengono eseguiti in base alla versione o configurazione. Inoltre, riduce il sovraccarico a runtime eliminando rami e salti condizionali mai eseguiti.

Per i programmi che operano su percorsi critici per la latenza del kernel (ad esempio XDP), l'eliminazione del codice morto è essenziale per risparmiare cicli di CPU, garantendo al contempo fallback condizionali risolti al momento del caricamento del programma.

-Inlining e Rewriting istruzioni

In alcuni casi, il sovraccarico delle chiamate dirette inizia ad accumularsi nei programmi. Quando l'operazione sottostante è molto semplice, il sovraccarico di chiamare una funzione può essere maggiore rispetto all'esecuzione diretta delle istruzioni. Questo vale anche per operazioni sensibili al tempo, come il helper bpf_jiffies64. Tuttavia, gli helper eBPF sono usati anche per far rispettare l'uso delle API e garantire l'integrità del contesto del programma. Il verificatore non consente ai programmi di manipolare direttamente le strutture dati del kernel.

In questi casi, il verificatore sostituisce automaticamente la chiamata all'helper con una serie di istruzioni eBPF equivalenti. Questa tecnica viene utilizzata per l'accesso alle mappe. Durante la verifica, il verificatore conosce staticamente a quali oggetti mappa eBPF il programma ha accesso. Quindi, quando il programma chiama gli helper eBPF che manipolano le mappe (come bpf_map_lookup_elem, bpf_map_update_elem e bpf_map_delete_elem), il verificatore può tradurre queste chiamate indirette in chiamate dirette all'implementazione della mappa sottostante, basandosi sul tipo di mappa noto durante la verifica.

9 – Compilazione Just in Time

Dopo le ottimizzazioni post-verifica, tutti i programmi eBPF vengono compilati Just-In-Time (JIT), ovvero convertiti in istruzioni macchina native prima di essere eseguiti. Il compilatore JIT traduce direttamente le istruzioni eBPF in istruzioni della macchina fisica, sfruttando la somiglianza tra l'architettura eBPF e quelle hardware. La compilazione avviene per ogni sottoprogramma e si articola in quattro fasi principali:

1. Allocazione dell'immagine: Si stima la dimensione dell'immagine JIT e si alloca la memoria per scrivere le istruzioni macchina tradotte.
2. Emissione del prologo: Si emette il codice necessario affinché il kernel possa chiamare in modo sicuro il programma eBPF. Questo include il salvataggio dei registri che devono essere preservati.
3. Emissione del corpo: Si traduce ogni istruzione eBPF in istruzioni macchina native. Per le istruzioni di caricamento da registri non sicuri, il compilatore JIT prepara una tabella di eccezioni che gestisce eventuali errori di accesso alla memoria, evitando crash del kernel.
4. Emissione dell'epilogo: Si ripristinano i registri salvati e si chiude il programma in modo sicuro. L'immagine finale è resa di sola lettura per evitare modifiche non autorizzate.

Quando è richiesta maggiore sicurezza (hardening), il compilatore JIT usa una tecnica chiamata "constant blinding", che consiste nell'offuscare i valori immediati (costanti) usati nelle istruzioni. Questi valori vengono XORati con una costante casuale per prevenire attacchi noti come JIT spraying, rendendo più difficile manipolare il codice compilato.

10 – Casi d’uso

-Networking

I programmi eBPF offrono diverse funzionalità chiave nel campo del networking, consentendo agli sviluppatori di gestire reti ad alte prestazioni, garantire sicurezza e flessibilità e migliorare l'efficienza grazie a "hook" personalizzabili nel kernel Linux. Principali funzionalità:

- **XDP e TC**: Permettono di gestire i pacchetti di rete in modo molto veloce, saltando alcune parti del kernel. Questo è utile per bloccare attacchi come il DDoS, distribuire il carico su più server, o inviare direttamente i pacchetti senza passare per tutto il sistema.
- **Socket Lookup**: Aiuta a decidere quale socket (la porta di comunicazione) userà un pacchetto di rete. Questo rende più efficiente il processo di indirizzamento del traffico di rete.
- **Socket Reuseport**: Permette di scegliere tra più socket che condividono lo stesso indirizzo IP e porta. Con eBPF, si può scegliere in modo intelligente quale socket usare, ad esempio in base alla posizione della memoria o altre esigenze.
- **Control Groups**: Questi "hook" sono legati ai container (piccoli ambienti isolati). Con essi puoi filtrare il traffico in entrata e in uscita, limitare la velocità di invio dei dati o applicare regole su come un'applicazione comunica.
- **User-Level Protocol (ULP)**: Consente di far rispettare le regole sui dati inviati da un'app, come quando si inviano file o messaggi. Funziona anche con traffico crittografato, mantenendo le politiche di sicurezza attive.
- **Controllo della congestione**: eBPF permette di testare e modificare facilmente gli algoritmi che controllano la congestione delle reti TCP, rendendo più efficiente la gestione del traffico nei server.
- **Operazioni sui socket**: eBPF può intervenire durante varie operazioni sui socket (le connessioni di rete), come quando un server accetta connessioni o stabilisce nuove connessioni. Può modificare opzioni o impostare regole dinamicamente per ottimizzare la connessione.

-Profiling

Le applicazioni utilizzano programmi eBPF collegati agli eventi di performance per raccogliere dati dai contatori delle prestazioni hardware e catturare tracce di esecuzione (stack traces) sia del kernel che dello spazio utente. Grazie al suo basso impatto sulle prestazioni, eBPF viene utilizzato in molte applicazioni di profilazione continua senza influenzare in modo significativo le prestazioni del sistema.

-Tracing

eBPF permette di tracciare le funzioni del kernel e i tracepoint (punti di monitoraggio predefiniti) eseguendo programmi all'ingresso e all'uscita di queste funzioni. I programmi possono accedere ai parametri delle funzioni, offrendo una visione dettagliata di ciò che accade nel sistema. Il basso impatto sulle prestazioni rende il tracciamento efficiente anche per applicazioni con carichi elevati, consentendo un monitoraggio continuo senza rallentare il sistema.

-Security

Il **sottosistema LSM BPF** permette di collegare i programmi eBPF agli **hook LSM** (Linux Security Module) presenti nel kernel, che servono per applicare politiche di sicurezza e monitorare il sistema. Grazie agli **LSM hook programmabili**, è possibile definire dinamicamente le politiche di sicurezza e controllare diversi aspetti del sistema in modo flessibile.

Un concetto chiave è l'uso delle **mappe di storage locale agli oggetti** (*object-local storage maps*), che associano dati specifici di una politica a oggetti del kernel, come **cgroups**, **inodes** o **socket**, che sono soggetti agli hook LSM. Questo permette di collegare informazioni di sicurezza direttamente agli oggetti che vengono controllati.

Inoltre, i programmi LSM BPF possono essere collegati a contesti **cgroup**, limitando l'ambito di applicazione delle politiche a specifici gruppi di processi. Queste capacità fanno di LSM BPF un elemento fondamentale per costruire framework di sicurezza flessibili e modulari, adattabili a diverse esigenze.

-Emerging

Alcune applicazioni emergenti e innovazioni utilizzano eBPF per estendere le funzionalità oltre i casi d'uso tradizionali del kernel Linux:

- Driver di dispositivo: Il framework HID-BPF permette di implementare parti dei driver per dispositivi HID usando programmi eBPF. Questo consente di filtrare eventi, correggere problemi nei driver senza modificare il kernel e aggiungere nuovi eventi di input.
- Schedulazione: Lo scheduler ghOSt delega la gestione della schedulazione a un'applicazione in userspace, utilizzando eBPF per comunicare gli eventi dello scheduler tramite memoria condivisa. Un altro approccio, SCHED-EXT, implementa completamente la logica di schedulazione all'interno di programmi eBPF come callback sincroni per il kernel Linux.
- Storage: XRP accelera le applicazioni di archiviazione collegando programmi eBPF al livello del driver NVMe. I programmi possono eseguire operazioni di lettura direttamente, bypassando lo stack di archiviazione del kernel, mantenendo comunque sincronizzato lo stato del file system.

11 – Sfide attuali eBPF

-Usabilità

Navigare tra le complessità di collegare i programmi eBPF ai punti di aggancio nel kernel Linux può essere complicato. È necessario capire quali hook (punti di aggancio) siano disponibili, valutarne l'idoneità per compiti specifici e integrare senza problemi i programmi eBPF, richiedendo una conoscenza approfondita del kernel e delle metodologie eBPF.

Inoltre, la scarsità di documentazione esaustiva e strumenti di sviluppo user-friendly aggrava le difficoltà di utilizzo. Gli sviluppatori spesso faticano a trovare risorse e indicazioni pertinenti, rallentando i progressi nello sviluppo di eBPF. Vi sono anche preoccupazioni relative alla compatibilità e stabilità tra diverse versioni del kernel. Le modifiche nelle API del kernel o nei punti di aggancio possono alterare il comportamento dei programmi eBPF, costringendo gli sviluppatori a dover continuamente adattare il codice per garantirne la compatibilità. Questo introduce complessità e un carico di lavoro extra non desiderato.

-Scalabilità del verificatore

Un verificatore più scalabile si traduce direttamente in una maggiore capacità per eBPF, poiché un insieme più ampio di programmi validi può essere approvato. Attualmente, la teoria di analisi statica del verificatore non è sufficientemente avanzata per gestire i casi estremi di **path explosion** (esplosione dei percorsi), dove il numero di stati da esaminare cresce esponenzialmente. Anche una gestione più scalabile dei cicli è fondamentale per aumentare l'espressività dei programmi eBPF. Invece di affidarsi all'unrolling dei cicli, sarebbe utile esplorare ulteriormente tecniche come l'**analisi delle invarianti dei cicli** o la **sintesi** dei cicli.

-Correttezza del verificatore

Le garanzie di sicurezza di eBPF dipendono dalla correttezza e affidabilità del verificatore. Il verificatore, essendo l'arbitro finale che decide se i programmi sono sicuri da eseguire, presenta alcune sfide. In primo luogo, il verificatore non può essere troppo conservativo, altrimenti rischia di rifiutare un gran numero di programmi validi e sicuri. In secondo luogo, l'algoritmo di verifica dovrebbe terminare idealmente entro un tempo fisso, senza superare il limite di complessità durante l'esecuzione simbolica del programma.

Nella pratica, il verificatore può ridurre significativamente l'impatto di questi problemi utilizzando gli algoritmi descritti in precedenza. Tuttavia, eventuali bug logici nell'algoritmo di verifica, la mancata considerazione di comportamenti pericolosi in percorsi non esplorati a causa di una potatura eccessiva dei percorsi, o il fallimento nel catturare e imporre correttamente gli invarianti specifici del kernel, potrebbero far passare programmi eBPF non sicuri o dannosi. Questo comprometterebbe le forti garanzie di sicurezza di eBPF. La complessità e la grandezza del codice del verificatore, insieme alle frequenti modifiche con ogni nuova versione del kernel, rendono sempre più difficile mantenere la correttezza della logica di verifica per gli sviluppatori di eBPF.

-Verifica formale

Non esiste ancora un'indagine formale completa sul verificatore e sulla validità delle sue garanzie di sicurezza. Questo rappresenta un problema di ricerca aperto, reso ancora più complesso dal grande numero di funzionalità supportate dal verificatore e dalle frequenti modifiche apportate al suo codice con ogni nuova versione del kernel.

Nonostante queste difficoltà, ci sono stati tentativi promettenti. Vishwanathan et al. hanno formalmente specificato il dominio astratto numerico del verificatore, fornendo prove di correttezza e ottimalità, e parte di questo lavoro è stato adottato nel kernel Linux. Bhat et al. hanno creato un framework di verifica formale automatizzato per controllare la correttezza della logica di analisi degli intervalli nell'implementazione in C del verificatore. Nelson et al. hanno sviluppato il framework Serval, che produce un verificatore automatizzato per il set di istruzioni eBPF, e hanno anche applicato tecniche di verifica automatizzata per garantire la correttezza delle implementazioni JIT di eBPF.

-Sicurezza

I bug critici in eBPF che influenzano la sicurezza e l'integrità del kernel, e quindi dell'intero sistema, sono evidenti dai vulnerabilità segnalate in passato. Queste vulnerabilità hanno spesso sfruttato eBPF come vettore di attacco in modalità non privilegiata. Di conseguenza, gli sviluppatori di eBPF hanno deciso di disabilitare la modalità non privilegiata di default. Anche quando questa modalità è attivata esplicitamente, le restrizioni sono molto severe per ridurre il rischio di attacchi da parte di utenti non attendibili o potenzialmente dannosi. Per questo motivo, eBPF rimane oggi ampiamente utile solo in un contesto di utenti fidati.

L'introduzione di **CAP_BPF** con Linux 5.8 ha l'obiettivo di separare le funzionalità BPF dalla capacità più ampia **CAP_SYS_ADMIN**. Questo significa che un utente con questa capacità può, tra le altre cose, creare mappe BPF o caricare programmi **SK_REUSEPORT**. Tuttavia, capacità come **CAP_NET_ADMIN** e **CAP_PERFMON** sono ancora necessarie per caricare programmi di rete e tracing, il che dimostra che alcune operazioni eBPF richiedono ancora privilegi elevati.

Migliorare le garanzie di correttezza del verificatore e dell'implementazione JIT di eBPF potrebbe, tra le altre cose, consentire di ridurre le restrizioni imposte ai programmi eBPF in modalità non privilegiata, rendendo eBPF utile per un numero maggiore di casi d'uso che non richiedono privilegi elevati.

-Riutilizzo del codice

Il riutilizzo del codice nei programmi eBPF ha vantaggi e svantaggi. Da un lato, c'è **CO-RE** (Compile Once, Run Everywhere), che consente di usare gli stessi programmi compilati su diverse versioni di Linux, correggendo gli offset di memoria per le strutture dati al momento del caricamento. Questo rende i programmi più portabili tra diverse versioni del kernel.

Dall'altro lato, anche se le chiamate a funzioni sono supportate nei programmi eBPF, non esiste il supporto per librerie statiche o dinamiche. Ciò significa che, se due o più programmi si trovano in file sorgente separati, potrebbero dover reimplementare le stesse funzioni, aumentando la duplicazione del codice e la complessità.

Table 1: List of eBPF Operations

Operation	Description
BPF_ALU_ADD	Add two registers
BPF_ALU_SUB	Subtract two registers
BPF_ALU_MUL	Multiply two registers
BPF_ALU_DIV	Divide two registers
BPF_ALU_MOD	Modulus of two registers
BPF_ALU_OR	Bitwise OR of two registers
BPF_ALU_AND	Bitwise AND of two registers
BPF_ALU_LSH	Shift left of register
BPF_ALU_RSH	Shift right of register
BPF_ALU_NEG	Negate value of register
BPF_ALU_XOR	Bitwise XOR of two registers
BPF_ALU_MOV	Move a value from one register to another
BPF_ALU_ARSH	Arithmetic right shift of register
BPF_ALU_END	End marker for ALU operations
BPF_JMP_JEQ	Jump if equal
BPF_JMP_JNE	Jump if not equal
BPF_JMP_JA	Jump always
BPF_JMP_JGT	Jump if greater than
BPF_JMP_JGE	Jump if greater than or equal
BPF_JMP_JLT	Jump if less than
BPF_JMP_JLE	Jump if less than or equal
BPF_JMP_JSET	Jump if bitwise AND with immediate is true
BPF_JMP_CALL	Call function
BPF_JMP_EXIT	Terminate execution
BPF_JMP_ALU64	Perform 64-bit arithmetic and jump
BPF_JMP_X	Reserved for future use
BPF_JMP_ADD	Add offset to register
BPF_JMP_MUL	Multiply register by scale
BPF_JMP_NEG	Negate register
BPF_JMP_AND	Bitwise AND with register
BPF_JMP_OR	Bitwise OR with register
BPF_JMP_XOR	Bitwise XOR with register
BPF_JMP_MOV	Move register to another
BPF_JMP_ARSH	Arithmetic right shift of register
BPF_JMP_END	End marker for JMP operations
BPF_STX	Store into memory
BPF_LDX	Load from memory
BPF_ST	Store value
BPF_LD	Load value