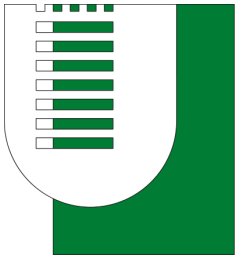

TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Progetto e realizzazione di un Transpiler per la programmazione di dataplane eBPF

Candidato:
Bernini Andrea

Relatore: **Prof. Loreti Pierpaolo**
Correlatore: **Prof. Salsano Stefano**

Corso di laurea triennale in Informatica
Università degli studi di Roma "Tor Vergata"

2021

Indice

Capitolo 1

Introduzione

Con l'avvento dell'era del software di rete, eBPF è diventata una tecnologia molto utilizzata per l'elaborazione efficiente dei pacchetti, su hardware generico. Esistono diversi software eBPF che offrono caratteristiche prestazionali senza rivali (come ad esempio il framework Cilium).

Tuttavia, lo sviluppo di soluzioni eBPF personalizzate è un processo impegnativo che richiede risorse umane altamente qualificate, il che ostacola il pieno sfruttamento del potenziale di eBPF. Per questo il netgroup di Tor Vergata propone il framework **HIKe** (Heal, Improve and desKill eBPF), che si basa su un'astrazione aggiuntiva (la VM HIKe) al di sopra della VM eBPF e sul concetto di concatenamento dei programmi HIKe eBPF. Il framework HIKe facilita la scrittura di programmi eBPF, migliorando così la produttività anche per sviluppatori esperti.

Inoltre, per facilitare ulteriormente lo sviluppo di applicazioni di rete, il netgroup di Tor Vergata ha progettato il framework **eCLAT** (eBPF Chains Language And Toolset), fornendo un'astrazione di programmazione di alto livello al framework HIKE. Con eCLAT lo sviluppatore può scrivere degli script che compongono i programmi HIKe eBPF, senza penalizzazioni prestazionali rispetto alla composizione dei programmi utilizzando il framework HIKe. Questo approccio semplifica il riutilizzo dei componenti, rendendo possibile la creazio-

ne di servizi di inoltro/elaborazione complessi, utilizzando componenti di base e un approccio di programmazione/composizione di alto livello, piuttosto che micro-programmazione a livello di codice sorgente eBPF.

Per scrivere gli script eCLAT viene utilizzato un linguaggio di alto livello, derivato dalla grammatica di Python, in modo tale da risultare familiare ai programmatori di rete. Per essere compatibili con il mondo HIKe, gli script eCLAT vengono tradotti in C grazie all'utilizzo di un **Transpiler**, ovvero un tipo di compilatore che prende in input il codice sorgente di un programma e produce il codice equivalente in un altro linguaggio di programmazione. eCLAT non si limita solo alla traduzione infatti consente di automatizzare i passaggi di configurazione di un sistema HIKe/eBPF.

Keywords: eBPF, HIKe, eCLAT, Transpiler.

Capitolo 2

Background

2.1 eBPF - eXtended Berkeley Packet Filter

L'eXtended Berkeley Packet Filter (eBPF) [1] è un linguaggio di programmazione di basso livello che può essere eseguito in una Virtual Machine (VM) in esecuzione nel kernel Linux. eBPF è stato spesso utilizzato per gestire in modo efficiente, sicuro e flessibile i pacchetti, senza richiedere alcun cambiamento nel codice sorgente del kernel o caricare i moduli di quest'ultimo. I programmi eBPF possono essere scritti utilizzando le istruzioni Assembly successivamente convertite in bytecode oppure in un C "ristretto" e compilati utilizzando il compilatore LLVM Clang. Il bytecode può essere caricato nel sistema tramite la syscall `bpf()` che forza il programma a superare una serie di controlli di integrità/sicurezza, per verificare se il programma può essere dannoso per il sistema.

In effetti, i programmi eBPF sono considerati estensioni del kernel non attendibili e solo i programmi eBPF *sicuri* possono essere caricati nel sistema. Per essere considerato sicuro, un programma deve soddisfare una serie di vincoli [2], che possono avere un impatto sulla capacità di creare potenti programmi di rete, come numero limitato di istruzioni, uso limitato di salti all'indietro, uso limitato dello stack e così via. La fase di verifica assicura che il pro-

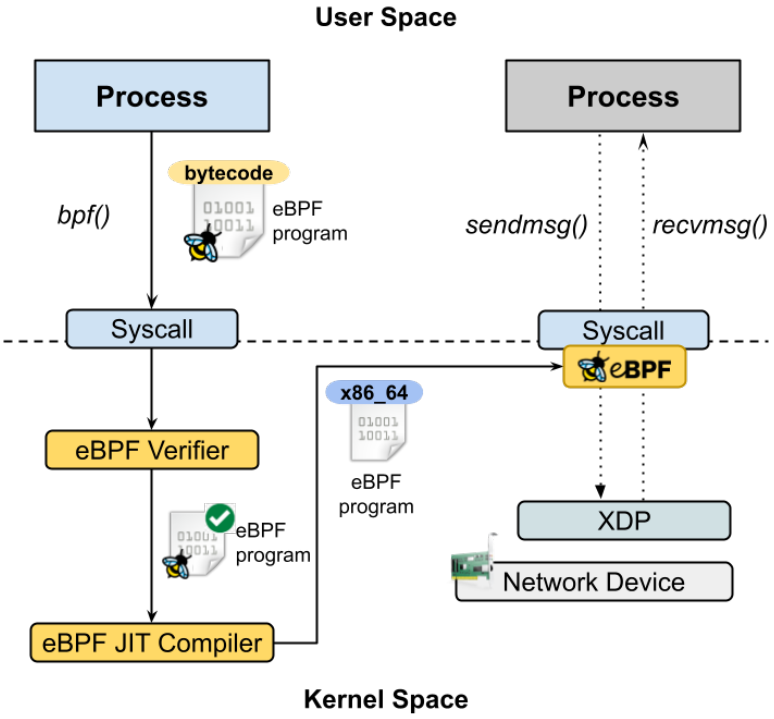

Figura 2.1: Compilazione e verifica dei programmi eBPF

gramma non vada in crash, che nessuna informazione possa essere trapelata dal kernel allo spazio utente e termina sempre. Quindi, un programma eBPF deve essere compilato e verificato prima di essere caricato in un sistema Linux in esecuzione. Questo processo è descritto in Figura 2.1. I programmi eBPF sono progettati per essere stateless in modo che ogni esecuzione sia indipendente dalle altre. L'infrastruttura eBPF fornisce strutture dati specifiche, chiamate **mappe BPF**, a cui possono accedere i programmi eBPF e lo spazio utente quando hanno bisogno di condividere alcune informazioni.

I programmi eBPF sono attivati da alcuni eventi interni e/o esterni che vanno dall'esecuzione di una specifica chiamata di sistema fino all'arrivo di un pacchetto di rete. Pertanto, i programmi eBPF sono collegati a diversi tipi di eventi e ognuno viene fornito con un contesto di esecuzione specifico. A seconda del contesto, esistono programmi che possono eseguire legittimamente operazioni di filtro dei socket mentre altri possono eseguire solo la classificazione del traffico a livello di controllo del traffico (TC) e così via.

Concentrandosi sull'elaborazione dei pacchetti, il programma eBPF può essere collegato a diversi hook e pacchetti che ne attivano l'esecuzione. Tra questi hook, ci concentriamo per i nostri scopi sul cosiddetto hook eXpress Data Path (XDP).

XDP

L'eXpress Data Path (XDP) è un componente di elaborazione dei pacchetti ad alte prestazioni basato su eBPF unito al kernel Linux dalla versione 4.8. XDP introduce un primo hook nel percorso RX del kernel, inserito nel driver NIC, prima che avvenga qualsiasi allocazione di memoria. Ogni pacchetto in arrivo viene intercettato prima di entrare nello stack di rete di Linux e, cosa importante, prima che allochi le sue strutture di dati, in primis `sk_buff`. Questo spiega la maggior parte dei vantaggi in termini di prestazioni ampiamente dimostrato in letteratura (ad esempio, [3] [4]). eBPF/XDP può essere vista come un'importante tecnologia abilitatrice, che guida i router software basati su Linux a elaborare i pacchetti con alta efficienza e raggiunge (relativamente) alta dappertutto.

2.2 HIKe - Heal, Improve and desKill eBPF

Come descritto nei capitoli precedenti eBPF, ha una curva di apprendimento molto ripida, in quanto nel suo processo di sviluppo e nella catena di strumenti, vi sono molte insidie. L'idea è quella di comporre programmi eBPF utilizzando un modello di chiamata di funzione e senza i problemi della fase di verifica. Tuttavia ciò non è possibile con l'attuale framework eBPF, neanche andando a migliorarlo direttamente, in quanto si andrebbero a violare alcuni presupposti fondamentali, come le garanzie di sicurezza fornite dal verificatore.

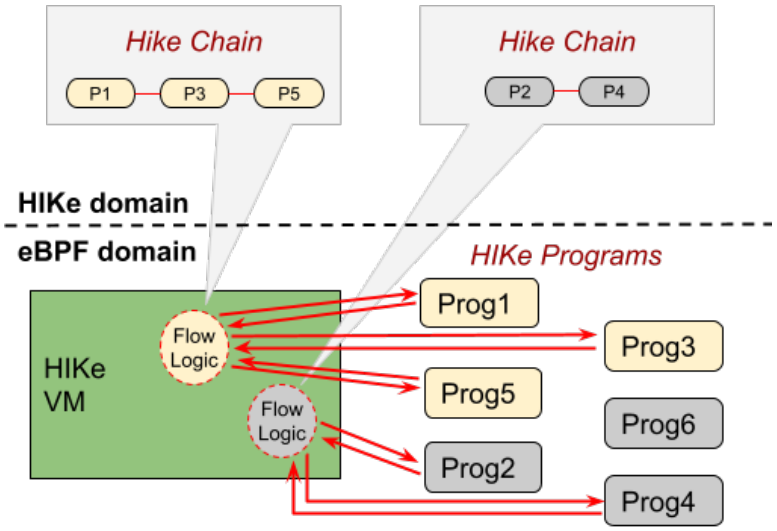

Figura 2.2: HIKe eBPF/XDP Architecture

La svolta proposta dal team di Tor Vergata consiste nell'aggiungere una nuova astrazione al di sopra della VM eBPF, chiamata VM HIKe (Heal, Improve and desKill eBPF). All'interno della quale, possiamo comporre ed eseguire programmi eBPF (in realtà, programmi HIKe eBPF) utilizzando il pattern di chiamata di funzione, senza passare per la fase di verifica e rispettando allo stesso tempo tutti i vincoli di sicurezza eBPF (Figura 2.2).

Dal punto di vista del framework eBPF, HIKe VM è un normale programma eBPF. La VM HIKe è in grado di eseguire (interpretare) un bytecode che chiamiamo HIKe Chain che descrive la composizione dei programmi HIKe eBPF. Per trasformare un programma eBPF in un programma HIKe eBPF dobbiamo "decorarlo" con alcune funzioni di supporto contenute nella libreria HIKe e poi ricompilarlo / verificarlo / caricarlo. Il bytecode della catena HIKe eseguito all'interno della VM HIKe è responsabile della decisione della logica di flusso (ovvero, i programmi eBPF da eseguire e la relativa logica). Diversamente dall'approccio tail-call i programmi eBPF che sono composti non sono a conoscenza della struttura delle interconnessioni, la logica di flusso può essere basata sui risultati dell'esecuzione dei programmi eBPF, ma può essere modificata in modo flessibile e modulare, indipendentemente dai programmi HIKe eBPF che

vengono compilati e verificati una volta per sempre, e infine la modifica della logica di flusso NON è soggetta al verificatore eBPF, perché la nuova catena HIKe è compilata per la VM HIKE, non per la VM eBPF.

Il bytecode di una catena HIke viene interpretato dalla HIke VM, che è un programma eBPF, il quale è stato precedentemente verificato prima di essere caricato. Di conseguenza, qualsiasi istruzione in quella catena che viene interpretata dalla VM HIKe risulta in delle operazioni eBPF che sono già state verificate come sicure, dal verificatore eBPF stesso.

La VM HIKe supporta tutte le operazioni di base necessarie per implementare la logica arbitraria: if, jump, jump condizionale, operazioni aritmetiche. Nella VM HIKe, una catena HIKe può anche richiamare direttamente altre catene HIKe. L'invocazione di una catena HIKe utilizza lo stesso modello di chiamata di funzione utilizzato per comporre programmi eBPF HIKe, mantenendo la stessa sintassi e semantica. La VM HIKe, quindi, fornisce un modello di chiamata unificato e semplice per invocare sia altre catene HIKe che programmi HIKe eBPF, promuovendo e massimizzando il riutilizzo del codice. Considerando che il dominio applicativo della VM HIKE è la gestione dei pacchetti di rete, nella VM HIKE sono state integrate alcune funzioni di supporto per manipolare (leggere e scrivere) i byte del pacchetto.

La Figura 2.3 illustra l'architettura complessiva del framework HIKe, che si basa sulle seguenti componenti:

- **Programma HIKe - eBPF**: Programma eBPF/XDP che si "decora" con le Librerie HIKe e può essere composto (concatenato) con altri Programmi HIKe per elaborare un pacchetto;
- **HIKe Chain**: La composizione (concatenamento) dei Programmi HIKe, delle operazioni logico/aritmetiche e delle istruzioni di controllo; una ca-

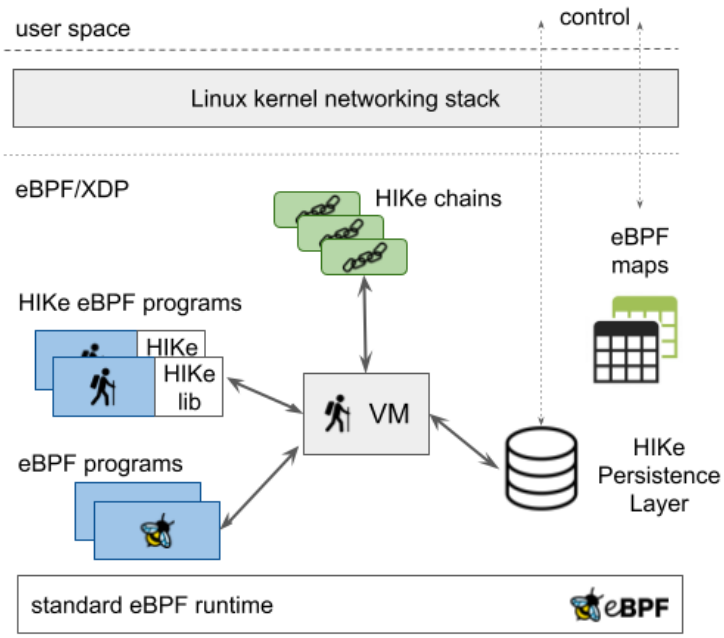

Figura 2.3: HIKe eBPF/XDP Architecture

tena HIKe è scritta in linguaggio C e compilata nel bytecode HIKe VM;

- **HIKe VM**: Un programma eBPF che si comporta come una Macchina Virtuale ed esegue il bytecode delle Catene HIKe;
- **HIKe Chain Loader**: Un programma eBPF (o un insieme di programmi) che associa una catena HIKe a un pacchetto e consegna il pacchetto alla VM HIKe che eseguirà la catena;
- **HIKe Persistence Layer (PL)**: Utilizzato per registrare i Programmi HIKe eBPF in modo che possano essere richiamati all'interno delle HIKe Chains e per memorizzare il bytecode delle HIKe Chains; l'HIKe PL è basato su mappe eBPF.

Capitolo 3

eCLAT

eCLAT è progettato per sfruttare le caratteristiche di modularità e componibilità del framework HIKe senza i problemi e le insidie della macchina virtuale eBPF e la sua programmazione in C e linguaggio assembly. eCLAT sposta la programmazione eBPF dai programmi C / assembly / bytecode verso script di alto livello simili a Python. Sebbene HIKe consenta al programmatore di strutturare il proprio codice in modo flessibile e modulare, la barriera tecnica all'ingresso per uno sviluppatore che voglia utilizzare questa concatenazione per lo sviluppo delle funzioni di rete è ancora alta. Infatti, utilizzando HIKe c'è ancora la necessità di interagire direttamente con il codice dei programmi eBPF e delle tabelle / mappe eBPF che servono per memorizzare la configurazione e lo stato dei diversi programmi e HIKe.

3.1 Funzioni di eCLAT

Le principali funzioni del framework eCLAT sono:

- **Importare i programmi HIKE** e umanizzarli associando ai loro ID dei nomi a misura d'uomo (stringhe univoche). Quindi eCLAT preleva dai programmi HIKe lo schema delle mappe che sono specifiche per ogni

programma HIKe e descrivono lo stato necessario per l'esecuzione del programma (i programmi eBPF sono stateless);

- **Configurare e interagire con i programmi HIKE** popolando e interagendo con le mappe dei programmi HIKE in modo semplice e intuitivo;
- **Definire un entry point** che sia il programma eBPF responsabile dell'attivazione di una catena HIKe. Questo può essere ad esempio un classificatore come il classificatore HIKe che si basa sull'indirizzo IPV6 di destinazione: in base a questo, possono essere eseguite diverse catene, implementando il modello di programmazione di rete SRv6.
- **Definire e caricare le catene HIKE** che possono essere liste di programmi HIKE o di catene HIKE (chain-in-chain).

eCLAT automatizza e semplifica il processo di sviluppo di una Networking App con HIKe (Figura 3.1), ovvero inizializza automaticamente il sistema (passo 0), interpreta lo script simil python, carica i programmi HIKe eBPF richiesti (passo 1), imposta le catene (passo 2) e configura il classificatore (passo 3, semplificando notevolmente la vita del programmatore eCLAT.

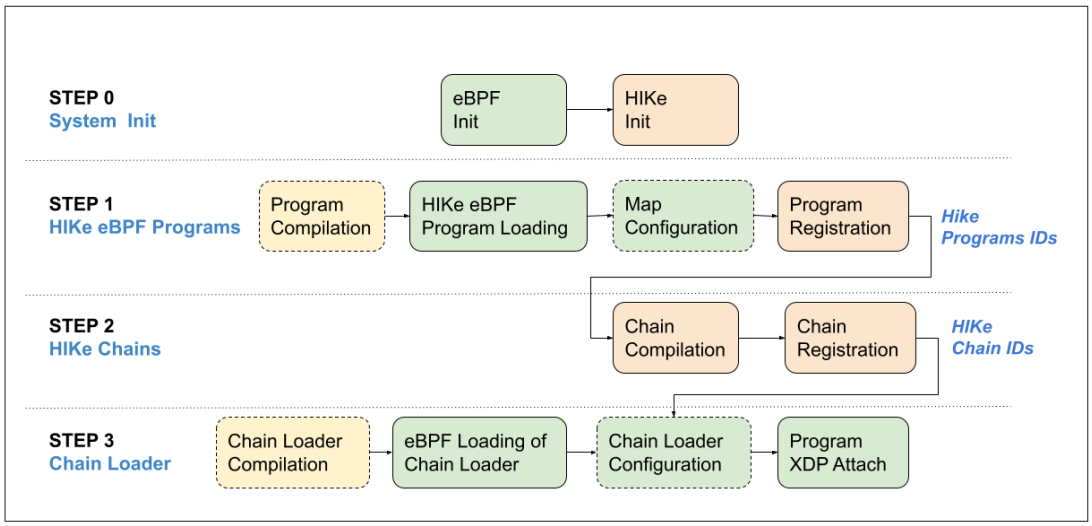

Figura 3.1: Passaggi necessari per configurare un sistema eBPF/HIKe completo

Infatti quest'ultimo non deve preoccuparsi degli ID dei programmi/catene HIKe ma può semplicemente fare riferimento ad essi utilizzando i loro nomi. Il framework eCLAT si occupa di derivare ID univoci e associarli ai programmi/-catene.

3.2 Architettura

La Figura 3.2 mostra la vista architetturale di eCLAT. Alla base dell'architettura abbiamo il dominio eBPF/XDP dove possiamo trovare i programmi HIKe eBPF, le catene HIKe e le mappe eBPF utilizzate da HIKe e dai programmi HIKe. Eclat è stato implementato come un demone (eclatd) che si occupa di compila-

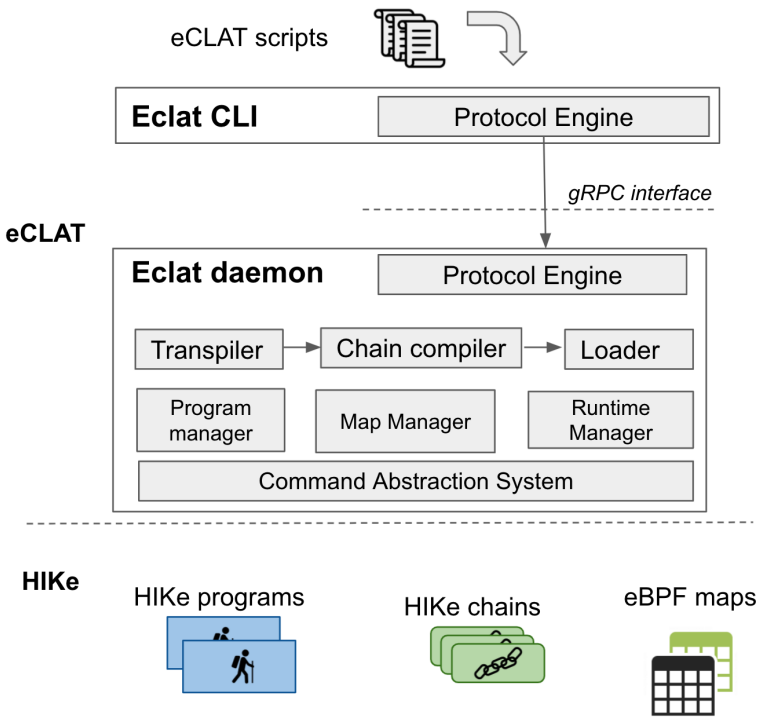

Figura 3.2: Architettura di eCLAT

re e configurare i programmi HIKe, mantenendo lo stato di quelli in esecuzione e interfacciandosi con quello ad esempio per acquisire informazioni sul loro stato (es. caso di programmi di monitoraggio). Uno dei motivi principali dietro

l'architettura basata su demoni risiede in una limitazione di sicurezza di eBPF. In effetti, solo il processo che ha montato il filesystem eBPF (o uno dei suoi figli) può avere accesso al filesystem stesso. Pertanto, l'architettura del demone semplifica la gestione di mappe e programmi e fornisce anche un accesso serializzato all'ambiente eBPF. Il demone Eclat riceve comandi utente da una CLI (eclat-cli) attraverso un'interfaccia gRPC. La struttura dei dati è descritta attraverso un linguaggio buffer di protocollo. Abbiamo implementato un servizio di esecuzione che trans-compila uno script eCLAT in un programma HIKe, compila il programma HIKe e lo carica in memoria. Come mostrato in figura 7, internamente il demone eCLAT è composto dai seguenti blocchi funzionali:

- **Transpiler**: si occupa di tradurre il codice sorgente, dallo script Eclat (simile a Python) a un programma (C) HIKe. Questo modulo è composto da un lexer e un parser. Crea l'Abstract Syntax Tree (AST) e quindi genera il codice sorgente del corrispondente programma HIKE, pronto per essere compilato;
- **Chain Compiler**: prende l'output del componente precedente e lo compila, per generare gli artefatti (file oggetto) attraverso l'esecuzione di un Makefile dedicato;
- **Chain Loader**: carica in memoria l'output del componente precedente. Questo viene fatto attraverso l'esecuzione di uno script della shell del caricatore;
- **Command Abstraction System (CAS)**: fornisce un'astrazione sui diversi comandi della shell che devono essere invocati sul sistema operativo;
- **Configuratore di sistema**: si occupa di impostare il sistema operativo. Ciò include il montaggio della directory /sys/fs/bpf/ e /sys/kernel/tracing;
- **Protocol Engine**: implementa il servizio di protocollo gRPC ed è responsabile della comunicazione con il CLI;

- **HIKe eBPF Program Manager (PM)**: carica e scarica i programmi HIKe eBPF. In particolare carica i programmi importati dallo script eCLAT (se non già caricati) e genera/recupera i loro ID (che vengono passati al Runtime Manager). Questi numeri identificativi saranno fondamentali per la fase di transpiler poiché i programmi HIKe si basano sul concetto di ID numerici piuttosto che sui nomi come nel mondo eCLAT;

- **eBPF Map Manager (MM)**: gestisce le mappe a livello eCLAT. Durante la compilazione dei programmi HIKe tutte le informazioni sulla struttura del programma (variabili, funzioni, struct, ecc.) vengono registrate in un file JSON. Il file viene anche analizzato per ottenere tutte le associazioni mappa-programma;

- **Runtime Manager (RM)**: raccoglie tutte le informazioni sullo stato di runtime. Le informazioni sullo stato sono registrate attraverso una struttura JSON, che contiene informazioni riguardanti:

  1. classificatori che sono stati scaricati e che sono stati selezionati per il caso in esame;
  2. una struttura statica per l'associazione degli ID ai programmi (coppie nome_programma - ID);
  3. i programmi HIKE scaricati/installati;
  4. le catene scaricate, insieme all'elenco dei programmi HIKE utilizzati e l'eventuale installazione di ciascuna catena;
  5. le mappe definite, insieme all'elenco dei programmi associati, contrassegnando i programmi già installati e la privacy dei programmi (condivisa o privata);
  6. l'ambiente (porte, interfacce, ecc.)

Capitolo 4

Transpiler eCLAT

La parte svolta dall'autore della tesi è quella riguardante il modulo del Transpiler, che si occupa dell'interpretazione dello script eCLAT, scritto in un linguaggio simil Python, e della sua traduzione in un codice sorgente equivalente scritto in un *C* ristretto, in modo tale da essere compatibile con il mondo eBPF/Hike.

4.1 Panoramica

Un **transpiler** è un tipo di compilatore che prende come input il codice sorgente di un programma scritto in un determinato linguaggio di programmazione e produce il codice sorgente equivalente nello stesso o in un linguaggio di programmazione diverso [5].

Un transpiler analizza il codice ed estrae i token, che sono gli elementi costitutivi di base del linguaggio (ad esempio le parole chiave della lingua, le variabili, i letterali e gli operatori, ecc.). Questo passaggio è una combinazione di **analisi lessicale** e **analisi sintattica** . Il Transpiler è in grado di fare ciò perché comprende le regole della sintassi della lingua di input. Data questa comprensione, il transpiler costruisce quello che viene chiamato Abstract Syntax

Tree (AST) . Successivamente trasforma quest'ultimo per adattarlo alla lingua di destinazione, per poi utilizzarlo per generare il codice tradotto.

I Transpiler si possono suddividere in due gruppi principali in base alla tipologia del risultato della traduzione.

- I transpiler del primo gruppo possono mantenere la struttura del codice tradotto il più vicino possibile al codice sorgente. La vicinanza qui si riferisce alla facilità con cui i blocchi di codice tradotto possono essere mappati al codice originale.
- I transpiler del secondo gruppo possono modificare la struttura del codice originale così tanto che il codice tradotto non assomiglia al codice sorgente. Questi transpiler hanno l'opportunità di ridurre le dimensioni ed eseguire ottimizzazioni complesse sul codice poiché non è necessaria la comprensibilità e la leggibilità del codice.

Il Transpiler eCLAT appartiene al primo gruppo in quanto si ha una traduzione uno a uno dei blocchi di codice.

4.2 Struttura del Transpiler eCLAT

Il Transpiler eCLAT può essere suddiviso logicamente in tre parti, ovvero:

- **Lexer**: prende in input lo script eCLAT e restituisce la lista dei token;
- **Parser**: prende in input la lista dei token e restituisce l'albero sintattico;

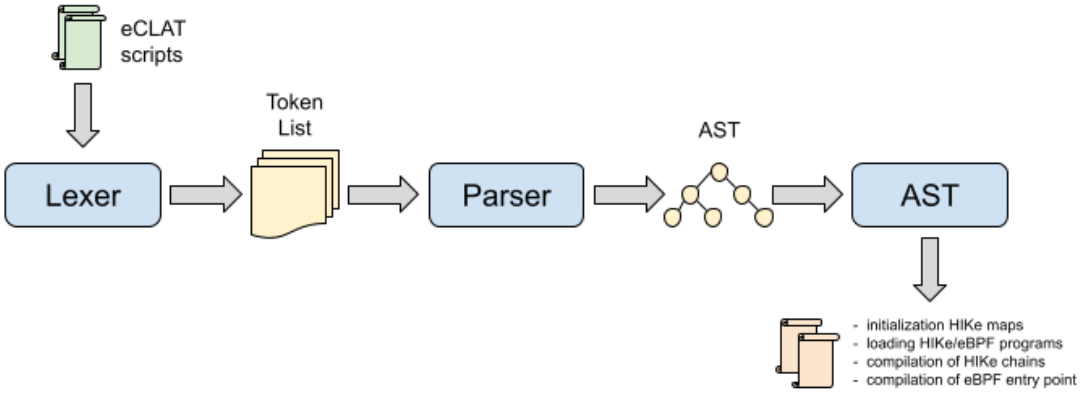

Figura 4.1: Struttura Transpiler

- **AST (Abstract Syntax Tree)**: viene eseguita una visita dell'albero per effettuare la traduzione dello script e l'esecuzione dei quattro step del processo di configurazione del sistema eBPF/HIKe (Figura 3.1).

4.2.1 Lexer

Il **Lexer** è un *Analizzatore Lessicale*, ovvero un programma che riceve in input uno stream di caratteri e li categorizza, formando così i **Token**, questo processo viene chiamato *tokenizzazione*. I **Token** sono definiti attraverso espressioni regolari e sono quindi costituiti da una coppia di valori, ovvero il nome del token e il *lessemma*, cioè una sequenza di caratteri indivisibili. Di conseguenza possiamo dire che i **token** corrispondono ai simboli terminali della grammatica di eCLAT.

Lexer Preprocessor

Un aspetto importante del lexer è la gestione degli spazi bianchi e dei commenti [6]. Nella maggior parte delle linguaggi di programmazione, la semantica del linguaggio è indipendente dagli spazi bianchi, ovvero gli spazi sono neces-

sari solo per contrassegnare la fine di un token e quindi sono anche chiamati **"trivia"** o **"minutiae"** poiché hanno poco valore per l'AST.

Tuttavia, questo non è il caso di tutti i linguaggi di programmazione, perché gli spazi bianchi possono avere significati semantici, come in Python, e di conseguenza anche eCLAT visto che la sua grammatica è un sottoinsieme di quest'ultima. Esistono due metodi di approccio differenti per gestire gli spazi bianchi e i commenti, ovvero scartarli o considerarli come Token. L'approccio utilizzato per il Transpiler eCLAT è stato il primo, infatti il Lexer ignora tutti i caratteri che identificano gli spazi bianchi ('\n', '\t', '\s', ecc), per quanto riguarda i commenti invece, la stringa in input prima di essere passata al Lexer, viene analizzata da un *preprocessore*, il quale si occupa di rimuovere i commenti utilizzando le espressioni regolari e inserire i *token per gestire l'indentazione*.

Più nello specifico gli spazi bianchi iniziali (spazi e tabulazioni) all'inizio di una riga logica vengono utilizzati per calcolare il livello di indentazione della riga, che a sua volta viene utilizzato per determinare il raggruppamento delle istruzioni [7]. Le tabulazioni vengono sostituite (da sinistra a destra) con quattro spazi in modo tale che il numero totale di caratteri fino alla sostituzione inclusa sia un multiplo di quattro. Il numero totale di spazi che precedono il primo carattere non vuoto determina quindi il rientro della riga.

I livelli di indentazione di righe consecutive vengono utilizzati per generare token INDENT e DEDENT, utilizzando uno stack, come segue. L'algoritmo per la gestione dell'indentazione e dei commenti, definito all'interno della funzione `lexer_preprocessor()` cicla per ogni riga e controlla (funzione `remove_comment()`) se è presente un commento, in caso positivo lo rimuove, se la riga non è vuota effettua il controllo dell'indentazione altrimenti passa alla successiva.

Per fare ciò l'algoritmo utilizza uno stack in cui viene memorizzato il livello di indentazione. I numeri inseriti nello stack saranno sempre rigorosamente crescenti dal basso verso l'alto. La parte superiore dello stack viene poi confrontata con il numero di spazi precedenti al primo carattere della riga (funzione `indentation()`). Se il numero calcolato è uguale, non succede nulla, se invece è più grande, significa che è stato "aperto" un blocco logico ("`def`", "`if`", "`while`", ecc) e quindi viene inserito nello stack e il Token di inizio blocco cioè INDENT viene generato. Invece se il numero è più piccolo, deve essere presente all'interno della pila; tutti i numeri nello stack più grandi vengono estratti e per ogni numero estratto viene generato un token DEDENT. Alla fine dello script, viene generato un token DEDENT per ogni numero rimasto nello stack.

Lista di definizione dei token eCLAT

```
INDENT;r'_indent(?!\w)'
DEDENT;r'_dedent(?!\w)'
U8;r'u8'
U16;r'u16'
U32;r'u32'
U64;r'u64'
S8;r's8'
S16;r's16'
S32;r's32'
S64;r's64'
HEX;r'0[xX][0-9a-fA-F]+(?!\w)'
FLOAT;r'-?\d+\.\d+'
INTEGER;r'-?\d+'
STRING;r'(""".*?""")|(".*?")|(\'.*?\')'
BOOLEAN;r'true(?!\w)|false(?!\w)'
IF;r'if(?!\w)'
ELIF;r'elif(?!\w)'
ELSE;r'else(?!\w)'
END;r'end(?!\w)'
AND;r'and(?!\w)'
```

```
21 OR;r'or(?!\w)'
22 NOT;r'not(?!\w)'
23 DEF;r'def(?!\w)'
24 FOR;r'for(?!\w)'
25 WHILE;r'while(?!\w)'
26 BREAK;r'break(?!\w)'
27 CONTINUE;r'continue(?!\w)'
28 RETURN;r'return(?!\w)'
29 IMPORT;r'import(?!\w)'
30 FROM;r'from(?!\w)'
31 IN;r'in(?!\w)'
32 IDENTIFIER;r'[a-zA-Z_][a-zA-Z0-9_]*'
33 >>;r'>>'
34 <<;r'<<'
35 ~;r'~'
36 &;r'&'
37 PIPE;r'\|'
38 ^;r'^'
39 ==;r'=='
40 !=;r'!='
41 >=;r'>='
42 <=;r'<='
43 >;r'>'
44 <;r'<'
45 =;r'='
46 [;r'\['
47 ];r'\]'
48 {;r'\{'
49 };r'\}'
50 ,;r','
51 :;r':'
52 .;r'\.'
53 DOT;r'\.'
54 PLUS;r'\+'
55 MINUS;r'-'
56 MUL;r'\*'
57 DIV;r'/'
```

```
58 MOD; r '%'
59 (; r '\('
60 ); r '\)'
61 NEWLINE; r '\n'
```

I Token utilizzati per il linguaggio eCLAT (che corrispondono ai simboli terminali della grammatica) sono più di *60* e sono definiti all'interno di un file CSV attraverso delle espressioni regolari. Il Lexer itererà carattere per carattere (scartando gli spazi) per decidere dove inizia e finisce ogni token, per poi attribuirgli il giusto tipo in base al confronto con le espressioni regolari dei token definiti. Se il Lexer non è in grado di farlo, segnalerà un errore per un token non valido.

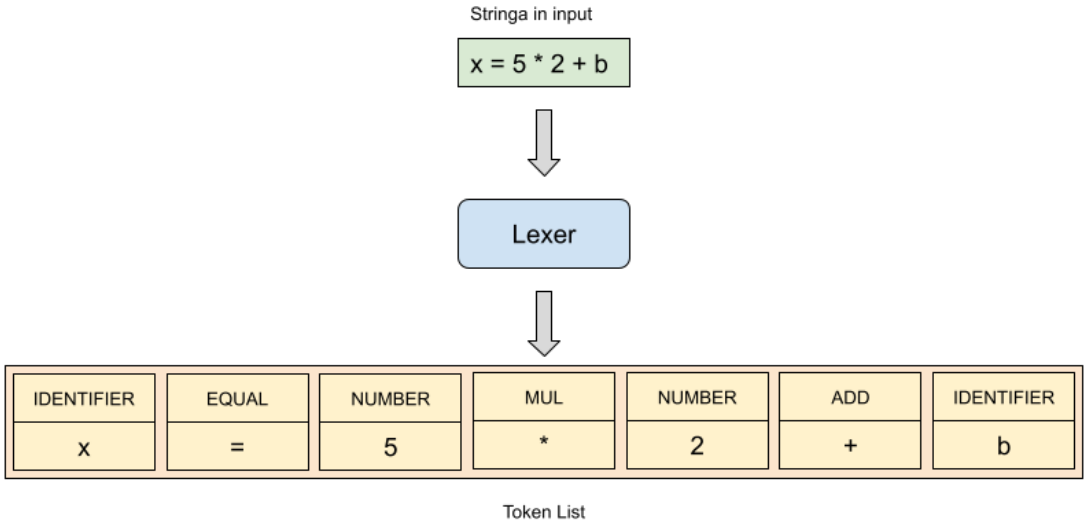

Figura 4.2: Esempio di tokenizzazione del Lexer

4.2.2 Parser

Il **Parser** si occuperà di analizzare la lista dei token, generata dal Lexer, controllando che rispettino l'ordinamento descritto dalla nostra grammatica, proprio come le frasi in italiano seguono strutture specifiche di verbi e sostantivi.

Esistono due diverse tipologie di approcci di base per l'analisi:

- **Analisi top-down**: Il parser analizza l'input cercando le regole di grammatica formale dall'alto verso il basso. In altre parole, la struttura di input di alto livello viene decisa prima di analizzare i suoi figli. Nell'analisi top-down, la derivazione più a sinistra di una produzione grammaticale viene trovata per prima. Pertanto, i token vengono consumati e abbinati da sinistra a destra. LL (Left-to-right, Leftmost derivation) parser sono esempi di parser top-down.
- **Analisi bottom-up**: Il parser tenta di analizzare prima i nodi foglia, quindi, a seconda di questi nodi analizzati, decide il genitore e lo analizza. Nell'analisi dal basso verso l'alto, la derivazione più a destra di una produzione grammaticale viene trovata per prima. LR (Left-to-right, Rightmost derivation) parser sono esempi di parser bottom-up.

Sia i parser LL che LR sono generalmente indicati con un numero ad esso associato come LL(1), LR(1), LL(k), ecc. Questo numero (generalmente 'k') indica il numero di lookahead utilizzati dal parser (simboli analizzati). Cioè, prendendo ad esempio l'analisi dall'alto verso il basso, il parser deve guardare avanti alcuni token per determinare quale nodo di livello superiore dovrebbe essere analizzato.

Il Parser eCLAT appartiene al secondo gruppo in quanto è stato costruito utilizzando la libreria **rPLY**, che basa la sua analisi su un algoritmo LALR (Il Look-Ahead LR è una versione semplificata di un parser LR canonico).

Il **Parser** esegue il controllo della sintassi costruendo una struttura dati, chiamata **Abstract Syntax Tree (AST)**, ovvero una rappresentazione ad albero della struttura sintattica del codice sorgente. L'AST viene costruito utilizzando la grammatica predefinita del linguaggio e la stringa di input. Se quest'ulti-

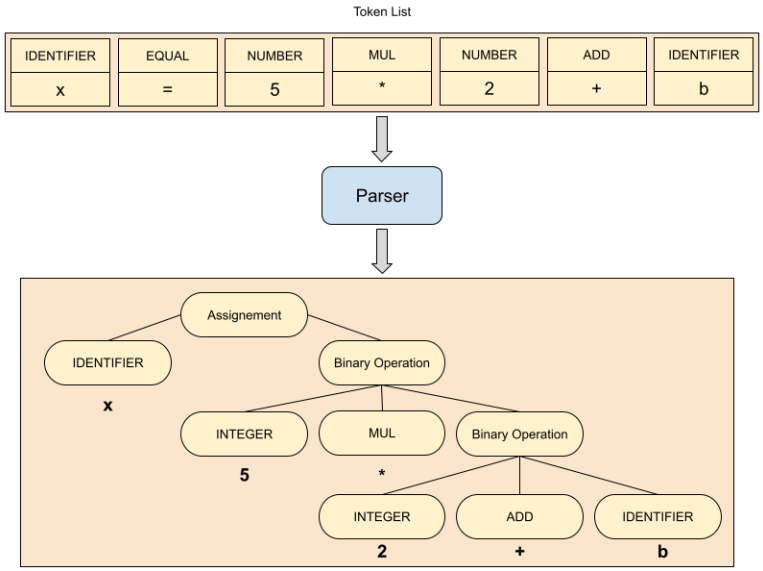

Figura 4.3: Esempio di analisi Sintattica

ma può essere prodotta con l'aiuto dell'albero della sintassi (nel processo di derivazione), la stringa di input si trova nella sintassi corretta. in caso contrario, l'errore viene segnalato dall'analizzatore di sintassi. Ogni nodo dell'albero denota un costrutto che si verifica nel codice sorgente, denominato **statement**.

Libreria rPLY

Nel nostro caso per facilitare la scrittura del Parser è stata utilizzata la libreria Python **rPLY**, la quale permette una migliore gestione degli errori ed una maggior efficienza rispetto ad un Parser scritto totalmente a mano. PLY non è altro che una semplice implementazione di lex/yacc, comunemente usati per scrivere parser e compilatori. L'analisi si basa sullo stesso algoritmo **LALR** utilizzato da molti strumenti yacc [8]. rPLY consente di rappresentare le regole di produzione attraverso il *decoratore*[1] `production()`, al cui interno viene definita la regola sotto forma di stringa.

[1]Decoratore: funzione che prende come parametro un'altra funzione, aggiunge delle funzionalità e restituisce un'altra funzione, senza appunto, alterare il codice sorgente della funzione passata come parametro.

Quindi per ogni **statement** (ovvero operazioni binarie, if, while, assegnazioni, ecc) avremo una funzione preceduta da diverse regole di produzione (definite attraverso il decoratore `production()` prima di quest'ultima), le quali selezionano la lista di Token da passare alla funzione, la quale ritorna un oggetto che rappresenta lo statement, creato selezionando i Token opportuni dalla lista passata alla funzione. Alla fine dell'analisi avremo così un oggetto `Program` contenente la lista di tutti gli statement trovati durante l'analisi, che corrisponde alla radice dell'AST.

ESEMPIO: Definizione della grammatica con rPLY

```
@self.pg.production('expression : expression PLUS expression')
@self.pg.production('expression : expression MINUS expression')
@self.pg.production('expression : expression MUL expression')
@self.pg.production('expression : expression DIV expression')
def expression_binop(p):
    return BinaryOperation(p[1].getstr(), left=p[0], right=p[2])

@self.pg.production('statement : IDENTIFIER = expression')
def statement_assignment(p):
    return Assignment(Variable(p[0].getstr()), p[2])
```

4.2.3 AST

Ogni nodo dell'**AST** rappresenta un costrutto che si verifica nel codice sorgente. La sintassi è "astratta" nel senso che non rappresenta ogni dettaglio che appare nella sintassi reale, ma solo i dettagli strutturali o relativi al contenuto [9]. Spesso, un albero di sintassi astratto (AST) viene confuso con un albero di analisi/sintassi, ovvero una rappresentazione concreta del codice sorgente, che conserva tutte le informazioni, comprese quelle banali come separatori, spazi bianchi, commenti, ecc. Un AST invece è una rappresentazione astratta e po-

trebbe non contenere alcune delle informazioni presenti nel codice sorgente [6] (Figura 4.5).

```
{
    int a = 5;
}
```

Figura 4.4: Codice di Esempio

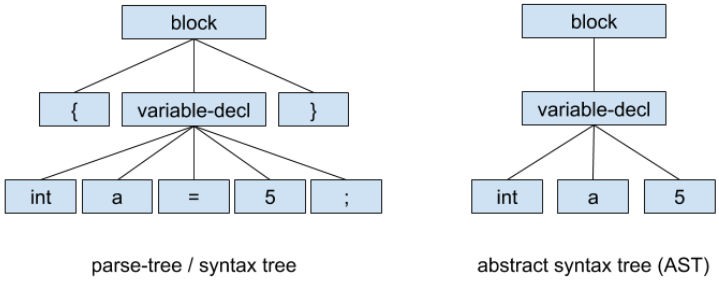

Figura 4.5: AST vs Parse Tree

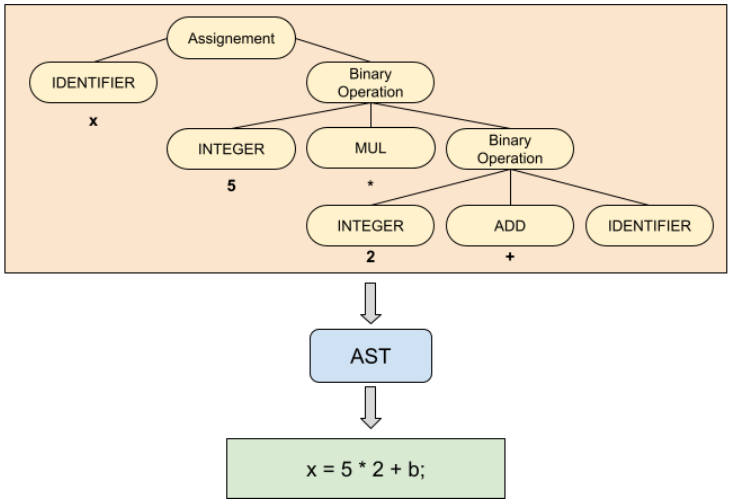

Figura 4.6: Visita dell'AST

Ogni nodo dell'AST corrisponde ad uno **statement** (while, if, operazioni binarie, ecc) che è rappresentato attraverso un oggetto Python che avrà un determinato numero di parametri e una funzione per la traduzione dello statement in C. L'esecuzione dell'albero procede con una visita ricorsiva dei nodi partendo dalla radice, ogni volta che un nodo viene visitato viene richiamata la funzione *to_c()* per la traduzione, la quale ritorna la stringa del codice C tradotto, che sarà concatenata al resto del codice.

ESEMPIO: Classe per lo statement delle operazioni binarie.

```
class BinaryOperation():
    def __init__(self, operator, left, right):
        self.operator = operator
        self.left = left
        self.right = right

    def to_c(self, env):
        return ' ' + self.left.to_c(env) + ' ' \
            + self.operator + ' ' + self.right.to_c(env) + ' '
```

Come si può notare lo statement delle operazioni binarie rappresentato attraverso la classe `BinaryOperation()` è composto da tre parti, la parte destra, la parte sinistra e l'operatore. Entrambi le parti a loro volta possono essere altre operazioni binarie, come nel caso si espressioni matematiche, o statement semplici come ad esempio un Intero o un Float.

4.3 Grammatica degli script eCLAT

La grammatica di un linguaggio di programmazione è formalizzata attraverso una quadrupla G = $\langle N, \Sigma, P, S \rangle$:

- N, sono i simboli non terminali che possono essere riscritti;
- Σ, sono i simboli terminali disgiunti da N (i nostri Token);
- S $\in$ N detto assioma è il simbolo non terminale iniziale.
- P sono le regole di produzione, una relazione binaria di cardinalità finita su (N $\cup$ Σ)* o N o (N $\cup$ Σ)* x (N $\cup$ Σ)*

La grammatica del linguaggio di programmazione eCLAT è un sottoinsieme di quella di *Python 3.9.5*, con in più delle piccole aggiunte (come i tipi `u8`, `u16`, ecc.). Infatti come in Python le variabili non sono tipizzate e viene usata l'indentazione per la sintassi delle specifiche, al posto delle più comuni parentesi.

4.3.1 Motivazioni

Il motivo di questa scelta è dovuto al fatto che Python gioca un ruolo essenziale nella programmazione di rete. Infatti, la libreria standard di Python ha il pieno supporto per i protocolli di rete, la codifica e la decodifica di dati e altri concetti di rete, ed è più semplice scrivere programmi di rete in Python che in C++. Essendo quindi un linguaggio molto usato in tale ambito, l'obiettivo è quello di fornire un ambiente familiare ai programmatori di rete oltre ad un'astrazione di programmazione di alto livello del framework HIKe, nascondendo i dettagli implementativi, visto che tutto il mondo eBPF sottostante è sviluppato interamente in C.

4.3.2 Struttura della grammatica

La grammatica del linguaggio eCLAT è definita nella forma EBNF (Extended Backus–Naur Form), una notazione meta-sintattica per le grammatiche Context-Free. In questo tipo di notazione l'insieme dei simboli terminali Σ è composto da tutte le stringhe tra virgolette (" "), invece l'insieme dei simboli non terminali N è composto da tutte le parole minuscole racchiuse dalle parentesi angolari (< >), l'assioma S è il non terminale `<main>`, infine le regole di produzione P sono le seguenti:

```
<main> := <program> ;

<program> := <statement_full>
    | <statement_full> <program> ;

<block> := "_indent" <blocks> "_dedent" ;

<blocks> := <statement_full>
    | <statement_full> <blocks> ;

<statement_full> := <statement> "\n"
    | <statement> "$end"
    | <statement> ;

<statement> := <expression>
    | "from" <module_path> "import" <module_list>
    | "import" <module_list>
    | <identifier> ":" <type> "=" <expression>
    | <identifier> ":" <type>
    | <identifier> "=" <expression>
    | "def" <identifier> "(" <arglist> ")" ":" "\n" <block>
    | "def" <identifier> "(" ")" ":" "\n" <block>
    | "while" <expression> ":" "\n" <block>  ;
    | "if" <expression> ":" "\n" <block> <elif_stmt>
    | "if" <expression> ":" "\n" <block> <else_stmt>
    | "if" <expression> ":" "\n" <block>
    | "if" <expression> ":" <statement> "\n" <elif_stmt>
    | "if" <expression> ":" <statement> "\n" <else_stmt>
    | "if" <expression> ":" <statement_full>
    | "return"
    | "return" <expression>

<elif_stmt> := "elif" <expression> ":" "\n" <block> <elif_stmt>
    | "elif" <expression> ":" "\n" <block> <else_stmt>
    | "elif" <expression> ":" "\n" <block>
    | "elif" <expression> ":" <statement> "\n" <elif_stmt>
    | "elif" <expression> ":" <statement> "\n" <else_stmt>
```

```
38      | "elif" <expression> ":" <statement_full> ;
39
40  <else_stmt> := "else" ":" <statement_full>
41      | "else" ":" "\n" <block> ;
42
43  <module_list> := <module_path>
44      | <module_path> "," <module_list> ;
45
46  <module_path> := <identifier>
47      | <identifier> "." <module_path> ;
48
49  <type> := "u8" | "u16" | "u32" | "u64"
50      | "s8" | "s16" | "s32" | "s64" ;
51
52  <arglist> := <identifier>
53      | <identifier> "," <arglist>
54      | <identifier> ":" <type>
55      | <identifier> ":" <type> "," <arglist> ;
56
57  <expression> := <const>
58      | <identifier>
59      | <identifier> "(" ")"
60      | <identifier> "." <identifier> "(" ")"
61      | <identifier> "(" expressionlist ")"
62      | <identifier> "." <identifier> "(" <expressionlist> ")"
63      | "(" <expression> ")"
64      | "not" <expression>
65      | "~" <expression>
66      | <expression> "+" <expression>
67      | <expression> "-" <expression>
68      | <expression> "*" <expression>
69      | <expression> "/" <expression>
70      | <expression> "!" "=" <expression>
71      | <expression> "=" "=" <expression>
72      | <expression> ">" "=" <expression>
73      | <expression> "<" "=" <expression>
74      | <expression> ">" <expression>
```

```
75      | <expression> "<" <expression>
76      | <expression> "and" <expression>
77      | <expression> "or" <expression>
78      | <expression> "&" <expression>
79      | <expression> "|" <expression>
80      | <expression> "^" <expression>
81      | <expression> ">" ">" <expression>
82      | <expression> "<" "<" <expression> ;
83
84  <expressionlist> := <expression>
85      | <expression> "," <expressionlist> ;
86
87  <const> := <float>
88      | <boolean>
89      | <integer>
90      | <hex>
91      | <string> ;
92
93  // Regular Expression
94  <identifier> := r'[a-zA-Z_][a-zA-Z0-9_]*' ;
95  <string> := r'(""".*?""")|(".*?")|(\'.*?\')' ;
96  <boolean> := r'true(?!\w)|false(?!\w)' ;
97  <integer> := r'-?\d+' ;
98  <hex> := r'0[xX][0-9a-fA-F]+(?!\w)' ;
99  <float> := r'-?\d+\.\d+' ;
```

4.4 Esempio di Traduzione

A seguire è riportato un esempio di codice eCLAT, il cui scenario di funzionamento è composto da tre nodi collegati tra loro. Il primo nodo classifica i pacchetti in base al TOS. Il secondo nodo eCLAT/HIKe riceverà questi pacchetti e attiverà una catena in base al numero di TOS riscontrato. Il programma `mon()` conterà i pacchetti che hanno un determinato TOS. Infine il terzo nodo

è quello ricevente i pacchetti.

```
from net import Packet
from hike import allow, mon, slow, fast

__offset_ttl = 64

def chain1(tos: s16):
    cnt: u32 = mon(tos)
    if (cnt < 1000):
        fast()
    else:
        slow()
    return -1

def chain2(tos: s16):
    mon(tos)
    slow()
    return -1

def chain3(tos: s16):
    Packet.writeU8(__offset_ttl, 10)
    chain2(tos)
    return -1

def chain_tos():
    eth_type: u16 = Packet.readU16(12)
    tos: s16 = -1
    if (eth_type == 0x800):
        tos = Packet.readU8(16)
    return tos

def chain_main():
    tos: s16 = chain_tos()
    if (tos < 0):
        slow()
        return -1
    # IPv4 traffic, go ahead
```

```
37      if (tos == 4):
38          chain1(tos)
39      elif (tos == 12):
40          chain2(tos)
41      elif (tos == 16):
42          chain3(tos)
43      elif (tos == 28):
44          fast()
45      else:
46          allow()
47      return -1
```

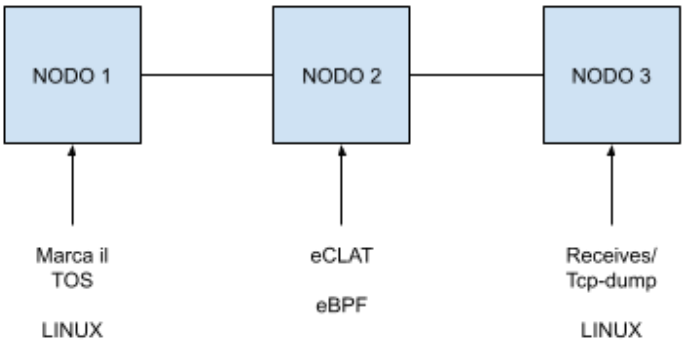

Figura 4.7: Scenario di Funzionamento

Il codice eCLAT precedente viene poi tradotto dal Transpiler eCLAT, il quale oltre a restituire il file C, si occupa anche di inizializzare le mappe HIKE a livello di sistema con pinning, caricare i programmi HIKe, internamente ciò richiede la compilazione del bytecode eBPF, la generazione di un ID programma (righe 7-10), il caricamento del programma, il pinning del programma e delle sue mappe e la registrazione al framework HIKe. Una volta che tutti i programmi HIKe eBPF sono stati registrati con successo, dobbiamo occuparci delle catene HIKe. Ciò richiede la compilazione delle catene con gli ID dei programmi all'interno, la generazione degli ID delle catene (righe 1-5) e il loro caricamento nella catena HIKE.

```
1 #define HIKE_CHAIN_75_ID 75
```

```
#define HIKE_CHAIN_76_ID 76
#define HIKE_CHAIN_77_ID 77
#define HIKE_CHAIN_78_ID 78
#define HIKE_CHAIN_79_ID 79

#define HIKE_EBPF_PROG_ALLOW_ANY 11
#define HIKE_EBPF_PROG_MON 13
#define HIKE_EBPF_PROG_SLOW 15
#define HIKE_EBPF_PROG_FAST 14

#define __OFFSET_TTL 64

HIKE_CHAIN_3(HIKE_CHAIN_75_ID, __s16, tos) {
    __u32 cnt;
    cnt = hike_elem_call_2(HIKE_EBPF_PROG_MON, tos);
    if ( cnt < 1000 ) {
        hike_elem_call_1(HIKE_EBPF_PROG_FAST);
    }
    else {
        hike_elem_call_1(HIKE_EBPF_PROG_SLOW);
    }
    return -1;
}

HIKE_CHAIN_3(HIKE_CHAIN_76_ID, __s16, tos) {
    hike_elem_call_2(HIKE_EBPF_PROG_MON, tos);
    hike_elem_call_1(HIKE_EBPF_PROG_SLOW);
    return -1;
}

HIKE_CHAIN_3(HIKE_CHAIN_77_ID, __s16, tos) {
    hike_packet_write_u8(__OFFSET_TTL, 10);
    hike_elem_call_2(HIKE_CHAIN_76_ID, tos);
    return -1;
}

HIKE_CHAIN_1(HIKE_CHAIN_78_ID) {
```

```
39      __u16 eth_type;
40      __s16 tos;
41      hike_packet_read_u16(&eth_type, 12);
42      tos = -1;
43      if ( eth_type == 0x800 ) {
44          hike_packet_read_u8(&tos, 16);
45      }
46      return tos;
47  }
48
49  HIKE_CHAIN_1(HIKE_CHAIN_79_ID) {
50      __s16 tos;
51      tos = hike_elem_call_1(HIKE_CHAIN_78_ID);
52      if ( tos < 0 ) {
53          hike_elem_call_1(HIKE_EBPF_PROG_SLOW);
54          return -1;
55      }
56      if ( tos == 4 ) {
57          hike_elem_call_2(HIKE_CHAIN_75_ID, tos);
58      }
59      else {
60          if ( tos == 12 ) {
61              hike_elem_call_2(HIKE_CHAIN_76_ID, tos);
62          }
63          else {
64              if ( tos == 16 ) {
65                  hike_elem_call_2(HIKE_CHAIN_77_ID, tos);
66              }
67              else {
68                  if ( tos == 28 ) {
69                      hike_elem_call_1(HIKE_EBPF_PROG_FAST);
70                  }
71                  else {
72                      hike_elem_call_1(HIKE_EBPF_PROG_ALLOW_ANY);
73                  }
74              }
75          }
```

```
76      }
77      return -1;
78  }
```

In questo esempio si può notare come eCLAT nasconde i dettagli implementativi di HIKe facilitando la scrittura di un programma e rendendo quindi meno ripida la curva di apprendimento per un programmatore. Lo si può notare, ad esempio, con le chiamata di programmi o chain, che lato eCLAT avvengono attraverso semplici chiamate a funzioni, mentre lato HIKe deve essere usata la funzione C specifica `hike_elem_call_x()` che permette tale costrutto lato eBPF. Tale traduzione è del tutto automatica, così come la definizione di nuove chain attraverso il costrutto `HIKE_CHAIN_X()`

Inoltre il Transpiler è in grado di risolvere alcuni problemi legati alla mancata corrispondenza di costrutti dal C al Python, e viceversa. Un esempio è il problema dell'inesistenza in C del costrutto `elif`, invece presente in Python, il quale viene risolto dal Transpiler combinando i costrutti `if` e `else`. Nonostante ciò la struttura dell'AST risultante non viene cambiata, in quanto il costrutto if viene incapsulato all'interno dell'`else`.

Capitolo 5

Test del Transpiler

Per testare il Transpiler è stato utilizzato un approccio molto pratico, ovvero sono stati scritti decine di programmi C per HIKe, che successivamente sono stati ”riscritti” in linguaggio eCLAT e quindi dati in pasto al traduttore, per poi confrontare il risultato della traduzione con il codice C originario.

La tecnica di test utilizzata è quella appartenente alla famiglia **Black Box**, ovvero il software è acceduto unicamente attraverso la sua interfaccia, senza accedere in maniera diretta al codice del componente da testare. Più precisamente è stata utilizzata la tecnica del *Testing delle Partizioni (o delle Classi di Equivalenza)*, in cui i dati di input ed output sono suddivisi in classi dove tutti i membri di una stessa partizione sono in qualche modo correlati, ed il programma si comporterà (verosimilmente) nello stesso modo per ciascun membro della classe.

La suddivisione in partizioni utilizzata per il test del Transpiler è quella in cui ci sono un insieme di classi di equivalenza corrispondenti all’insieme dei valori considerati validi per l’input ed un altro insieme corrispondente a quelli non validi. Entrambi gli insiemi possono essere suddivisi in ulteriori classi di equivalenza, corrispondenti alla tipologia di statement da testare, ovvero a tutte le produzioni base della grammatica (`if`, `while`, ecc.).

La strategia di test utilizzata corrisponde alla *Copertura minima delle classi di equivalenza*, cioè ogni partizione è coperta almeno da un caso di test. Tale tecnica garantisce la copertura delle classi di equivalenza con il minor numero di casi di test, avendo così un'elevata efficienza, tuttavia ha uno svantaggio, ovvero nel caso in cui si riscontri un difetto può essere difficile individuarne la causa.

I test sono stati fatti attraverso due tipi di input ovvero corretto e volutamente errato, che rappresentano le rispettive classi di equivalenza.

5.1 Input Errato

Quello che è stato fatto è stato tradurre script incorretti, cioè codice in cui erano presenti Token non ammessi o in cui la sintassi era errata, verificando che i messaggi di errore restituiti dal transpiler fossero quelli attesi, infatti per ogni test errato sono stati definiti dei meta-dati che definiscono l'output previsto, cioè il messaggio di errore specifico che il transpiler dovrebbe produrre.

5.2 Input Corretto

A differenza del caso precedente, è stato testato un insieme di script che rappresentano vari input corretti, i cui output sono stati confrontati con i file C scritti a mano. Sono state definite diverse classi di script, ognuna delle quali si "preoccupava" della traduzione di specifici `statement`, per poi passare a script più corposi aventi diversi tipi di statement. Tuttavia ciò non è sufficiente, infatti oltre al semplice controllo della traduzione degli input, occorre controllare che l'AST prodotto sia corretto. Per risolvere questo problema, per ogni

script è stato definito l'AST previsto come output per la traduzione, che poi sarà utilizzato per essere confrontato con quello risultante dalla traduzione.

5.3 Esempio di Test

A seguire è riportato un piccolo esempio di codice eCLAT per la gestione dei pacchetti IPv4 e IPv6. Il codice può essere suddiviso in tre parti principali:

- la prima (righe 1-2) consiste nell'importazione dei programmi HIKe (`allow` e `drop`) per poi poter essere utilizzati nello script, e della classe `Packet` usata per poter leggere e scrivere i pacchetti.
- la seconda (righe 4-5), in cui vengono definite delle variabili globali (in eCLAT al di fuori della definizione delle chain possono essere definite esclusivamente le variabili, non si può scrivere altro tipo di codice).
- la terza (righe 7-19) consiste nella definizione della chain di nome `mychain1`, all'interno della quale viene letto il pacchetto e se questo è di tipo IPv4 viene scartato (`drop()`), altrimenti se è IPv6 viene letto il campo `TTL` [1], se questo è diverso da 64 viene lasciato passare (`allow()`) e la chain termina, altrimenti viene riscritto il campo `TTL`.

Questo codice serve a testare un classe di programmi molto semplice in cui viene definita una chain e al cui interno vengono scritte semplici istruzioni per il controllo del flusso, oltre ad utilizzare delle funzioni per leggere/scrivere i pacchetti e chiamare programmi eBPF.

```
1 from net import Packet
```

[1] Time to live (TTL) o limite di hop è un meccanismo che limita la durata o la durata dei dati in un computer o in una rete. TTL può essere implementato come contatore o marca temporale allegato o incorporato nei dati.

```
from hike import drop, allow

__eth_proto_type_abs_off = 12
__ipv6_hop_lim_abs_off = 21

def mychain1():
    Packet.readU16(__eth_proto_type_abs_off)
    eth_type = Packet.readU16(__eth_proto_type_abs_off)
    if eth_type == 0x800:
        drop(eth_type)
        return

    if (eth_type == 0x86dd):
        hop_lim = Packet.readU8(__ipv6_hop_lim_abs_off)
        if hop_lim != 64:
            allow(eth_type)
            return
        Packet.writeU8(__ipv6_hop_lim_abs_off,17)
```

Il risultato della traduzione del codice eCLAT è il seguente:

```
#define HIKE_CHAIN_77_ID 77

#define HIKE_EBPF_PROG_DROP_ANY 12
#define HIKE_EBPF_PROG_ALLOW_ANY 11

#define __ETH_PROTO_TYPE_ABS_OFF 12
#define __IPV6_HOP_LIM_ABS_OFF 21

HIKE_CHAIN_1(HIKE_CHAIN_77_ID){
    __u16 eth_type;
    __u8 hop_lim;
    hike_packet_read_u16(&eth_type, __ETH_PROTO_TYPE_ABS_OFF);
    if (eth_type == 0x800){
        hike_elem_call_2(HIKE_EBPF_PROG_DROP_ANY, eth_type);
        return 0;
    }
```

```
17      if (eth_type == 0x86dd){
18          hike_packet_read_u8(&hop_lim, __IPV6_HOP_LIM_ABS_OFF);
19          if (hop_lim != 64){
20              hike_elem_call_2(HIKE_EBPF_PROG_ALLOW_ANY, eth_type);
21              return 0;
22          }
23          hike_packet_write_u8(__IPV6_HOP_LIM_ABS_OFF, 17);
24      }
25      return 0;
26  }
```

Invece il codice C originario scritto a mano è il seguente, differisce dall'output del Transpiler esclusivamente per la presenza del costrutto `goto`, il quale non è presente in Python, tuttavia per come è stato scritto lo script eCLAT la logica e il funzionamento del programma rimane la stessa.

```
1   #define HIKE_CHAIN_77_ID 77
2
3   #define __ETH_PROTO_TYPE_ABS_OFF 12
4   #define __IPV6_HOP_LIM_ABS_OFF 21
5
6   HIKECHAIN1(HIKE_CHAIN_77_ID){
7       __u16 eth_type;
8       __u8 hop_lim;
9       hike_packet_read_u16(&eth_type, __ETH_PROTO_TYPE_ABS_OFF);
10      if (eth_type == 0x800)
11          goto drop;
12
13      if (eth_type == 0x86dd){
14          /* change the TTL of the IPv4 packet */
15          hike_packet_read_u8(&hop_lim, __IPV6_HOP_LIM_ABS_OFF);
16          if (hop_lim != 64)
17              goto allow;
18
19          /* rewrite the hop_limit */
20          hike_packet_write_u8(__IPV6_HOP_LIM_ABS_OFF, 17);
21      }
```

```
22
23      /* by default allow any protocol */
24  allow:
25      hike_elem_call_2(HIKE_EBPF_PROG_ALLOW_ANY, eth_type);
26      goto out;
27  drop:
28      hike_elem_call_2(HIKE_EBPF_PROG_DROP_ANY, eth_type);
29  out:
30      return 0;
31  }
```

Capitolo 6

Conclusioni

In questa tesi è stato introdotto eCLAT, un framework e un linguaggio di alto livello per la composizione di programmi di elaborazione dei pacchetti eBPF. eCLAT fornisce un'astrazione del framework HIKe che supporta il concatenamento efficiente di programmi eBPF in un singolo nodo. Il chiaro vantaggio di eCLAT è la possibilità di programmare componenti nel framework eBPF senza la difficoltà di interazione di basso livello con il linguaggio di programmazione C e i vincoli eBPF. La seconda innovazione di eCLAT è il supporto della composizione a livello di rete dei programmi eBPF (HIKe).

Il modulo eCLAT di cui si è occupato l'autore della tesi è quello del Transpiler, il quale permette di tradurre il codice eCLAT di alto livello, in un codice C ristretto compatibile con HIKe, e inizializzare automaticamente il sistema, caricare i programmi HIKe eBPF richiesti, impostare le catene e configurare il classificatore, semplificando notevolmente la vita del programmatore eCLAT, al quale vengono nascosti i dettagli più complicati e non deve necessariamente interagire direttamente con il codice dei programmi eBPF e delle tabelle/mappe eBPF che servono per memorizzare la configurazione e lo stato dei diversi programmi e HIKe.

Il Transpiler, in base alla grammatica definita allo stato attuale, è stato implementato del tutto. In futuro in base all'ulteriore sviluppo della HIKe VM,

sarà possibile estendere la grammatica e implementare quindi nuove classi dell'AST per la traduzione.

L'intero progetto del Transpiler è composto da oltre 1800 righe di codice Python e l'AST conta oltre 25 classi ognuna rappresentante un costrutto della grammatica di eCLAT (”if”, ”while”, ecc...)

Riferimenti

[1] Linux Socket Filtering aka Berkeley Packet Filter (BPF).

[2] Sebastiano Miano et al. Creating Complex Network Services with eBPF:Experience and Lessons Learned. In *IEEE International Conference on High Performance Switching and Routing (HPSR2018)*. IEEE, 2018.

[3] N. Van Tu et al. evnf - hybrid virtual network functions with linux express data path. In *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2019.

[4] D. Scholz et al. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, 2018.

[5] Medium transpiler. available online at `https://aarnavjindal.medium.com/transpilers-source-to-source-compilers-b9affc27e148`.

[6] Write a parser. available online at `https://medium.com/swlh/writing-a-parser-getting-started-44ba70bb6cc9`.

[7] Lexical analysis. available online at `https://docs.python.org/3/reference/lexical_analysis.html`.

[8] Libreria rply. available online at `https://rply.readthedocs.io/en/latest/`.

[9] Introduction to a parser. available online at `https://medium.com/@chetcorcos/introduction-to-parsers-644d1b5d7f3d`.