UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMĂ

vTapping cu eBPF

Giorgiana-Lavinia Bălan

Coordonator științific:

Conf. Dr. Ing. Răzvan Deaconescu

BUCUREȘTI

2021

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

vTapping with eBPF

Giorgiana-Lavinia Bălan

Thesis advisor:

Conf. Dr. Ing. Răzvan Deaconescu

BUCHAREST
2021

CUPRINS

LISTĂ DE FIGURI

LISTINGS

LISTĂ DE NOTAȚII ȘI ABREVIERI

vTapping	Virtual Tapping
BPF	Berkeley Packet Filter
eBPF	extended Berkeley Packet Filter
BCC	BPF Compiler Collection
HTTP	Hypertext Transfer Protocol
OVF	Open Virtualization Format
CLI	Command Line Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ICMP	Internet Control Message Protocol

SINOPSIS

Având în vedere dezvoltarea continuă în aria rețelelor, se remarcă necesitatea dezvoltării unor tool-uri din ce în ce mai performante care să analizeze buna funcționare a canalului de comunicație dintre două endpoint-uri (noduri din rețea). Prin intermediul unor astfel de programe de monitorizare a traficului dintre sisteme care oferă statistici cu privire la performanța canalului de comunicație se pot observa diferite probleme sau comportamente nedorite ale acestuia. Așadar, avem nevoie să analizăm rețeaua dezvoltată în cât mai multe cazuri de utilizare pentru a facilita procesul de îmbunătățire al implementării și dezvoltarea rețelei.

ABSTRACT

Given the continuous development in the area of networks, there is a need to develop more and more powerful tools to analyze the proper functioning of the communication channel between two endpoints (network nodes). Using such traffic monitoring programs between systems that provide statistics on the performance of the communication channel, various problems or unwanted behaviors can be observed. Therefore, we need to analyze the developed network in as many use cases as possible in order to facilitate the process of improving the implementation and development of the network.

Aș vrea să transmit sincere aprecieri coordonatorului meu științific, Conf. Dr. Ing. Răzvan Deaconescu, pentru implicare, îndrumare și feedback-ul periodic în realizarea acestei lucrări de diplomă.

1 INTRODUCERE

Rețelistica și telecomunicațiile sunt domenii foarte complexe care au un impact major asupra susținerii și dezvoltării societății actuale. Fără acestea nu ar fi posibil schimbul de informații pe distanțe mari aproape instantaneu.

În acest context, canalul de comunicație este o noțiune de bază. Prin canal de comunicație ne referim la un mediu de transmisie fizică sau prin semnale electromagnetice. Ne dorim, astfel, ca acest canal să prezinte performanțe cât mai bune. Din perspectiva specialiștilor în domeniu, acest lucru presupune, în principal, o întârziere de răspuns cât mai mică, rata de transmisie a datelor cât mai mare, număr de pachete pierdute cât mai mic (care tinde spre zero) și fiabilitate.

Ținând cont de cele menționate anterior, prin actualul proiect se dorește realizarea unui tool de monitorizare a traficului care să prezinte capabilitatea de a măsura în timp real diverși parametrii ai comunicării în interiorul unei rețele.

2 MOTIVAȚIE

2.1 De ce vTapping?

Având în vedere migrarea continuă către mediile virtuale de calcul, este esențial ca acestea să rămână sigure și să își păstreze buna funcționare. Pentru a asigura acest lucru avem nevoie de soluții capabile să monitorizeze și să analizeze sistemele în vederea surprinderii diverselor probleme ce pot apărea. În acest context, putem vorbi despre vTapping.

Virtual Tap-ul (vTap-ul) este o soluție software care capturează o copie a pachetelor care circulă între două mașini virtuale (VMs). Prin intermediul unui vTap se poate analiza atât traficul dintre două mașini virtuale cât și traficul din interiorul unei singure mașini. vTap-urile sunt capabile să filtreze traficul capturat și să îl trimită mai departe tool-urilor fizice sau virtuale de monitorizare. Acestea, la rândul lor, analizează traficul primit și pot estima performanța canalului de comunicație dintre mașini.

Un virtual Tap aduce beneficiul vizibilității în rețea prin posibilitatea capturării traficului critic în punctele considerate "oarbe" și transmiterea acestuia către instrumentele de securitate sau de analiză a performanțelor.

Pe cazuri reale, Tap-urile virtuale sunt benefice, în principal, pentru reducerea problemelor de performanță (prin observarea continuă a datelor din interiorul sistemului) și consolidarea securității (reducând riscul apariției unor atacuri cibernetice costisitoare). [1]

2.2 Soluții existente

În prezent, există numeroase utilitare/programe de capturare/analizare a traficului atât în Windows, cât și Linux. Câteva dintre acestea, destul de cunoscute și utilizate, sunt Wireshark, tcpdump, netsniff-ng, iftop, nmap și bmon, care fie doar capturează pachetele și le afișează utilizatorului, fie realizează anumite măsurători, cum ar fi bandwidth-ul sau erori ale CRC (Cyclic Redundancy Check), în cazul bmon. Aceste utilitare folosesc în implementare biblioteca libpcap care a fost implementată de dezvoltatorii TCPDUMP-ului.

Aceste soluții deja existente prezintă câteva probleme sau inconveniente precum: overhead-ul mare cauzat de tranziția user space - kernel space sau faptul că prezintă output-ul într-un mod mai puțin intuitiv și mai dificil de urmărit sau înțeles de către utilizatori. O altă

problemă este și faptul că unele dintre aceste utilitare măsoară doar anumite metrici și astfel nu pot acorda utilizatorului o imagine de ansamblu asupra statusului și performanței rețelei. Pentru analiza cât mai concretă a bunei funcționări a rețelei dezvoltate avem nevoie să știm simultan valorile mai multor metrici de performanță pentru a estima cât mai concis existența unor probleme sau tipul acestora.

2.3 Problema

Problema principală adresată prin intermediul proiectului curent este legată de rapiditatea rulării unui astfel de program de analiză a traficului de pachete, în contextul dezvoltării continue din domeniul telecomunicațiilor/rețelelor. Este nevoie de programe de analiză a traficului cât mai rapide care să aibă relevanță cât mai mare în ajutorul dezvoltatorilor din acest domeniu.

Deci, focusul se mută pe găsirea unei modalități de preluare a datelor din pachete și prelucrarea acestora cu un overhead de timp cât mai mic posibil.

2.4 Soluția propusă

O tehnologie relativ nouă din același domeniu al problematicii în discuție este eBPF. Berkeley Packet Filter (BPF) este o mașină virtuală care rulează just-in-time în kernelul Linux. Acest lucru este favorabil având în vedere confortul și siguranța pe care le oferă. [2]

Inițial, BPF-ul era folosit pentru filtrarea pachetelor de HTTP în sistemul de operare BSD (Berkeley Software Distribution). Apoi, această tehnologie a fost extinsă pentru software-defined networking (SDN) și, de asemenea, pentru alte scopuri în afara rețelisticii, având denumirea de eBPF (extended Berkeley Packet Filter) care a fost introdusă în noile versiuni de kernel Linux [3].

eBPF-ul este o tehnologie promițătoare care face posibilă rularea de programe direct in kernelul Linux fără a fi nevoie de încărcarea altor module în kernel sau modificarea codului sursă al acestuia.

Siguranța pe care o oferă eBPF-ul este dată de câteva constrângeri cum ar fi memoria, numărul limitat de instrucțiuni alocate per program sau imposibilitatea compilării codului care conține bucle interminabile, lucruri care evită crash-ul sistemului sau alte probleme specifice. Cu alte cuvinte, eBPF-ul asigură terminarea programelor, fără să se agațe.

Acest proiect folosește framework-ul BCC, un kit de compilare bazat pe LLVM, prin intermediul căruia se poate interacționa din Python cu eBPF-ul.

Următoarea diagramă arată care sunt pașii prin care trece un program eBPF pentru a

ajunge să fie executat.

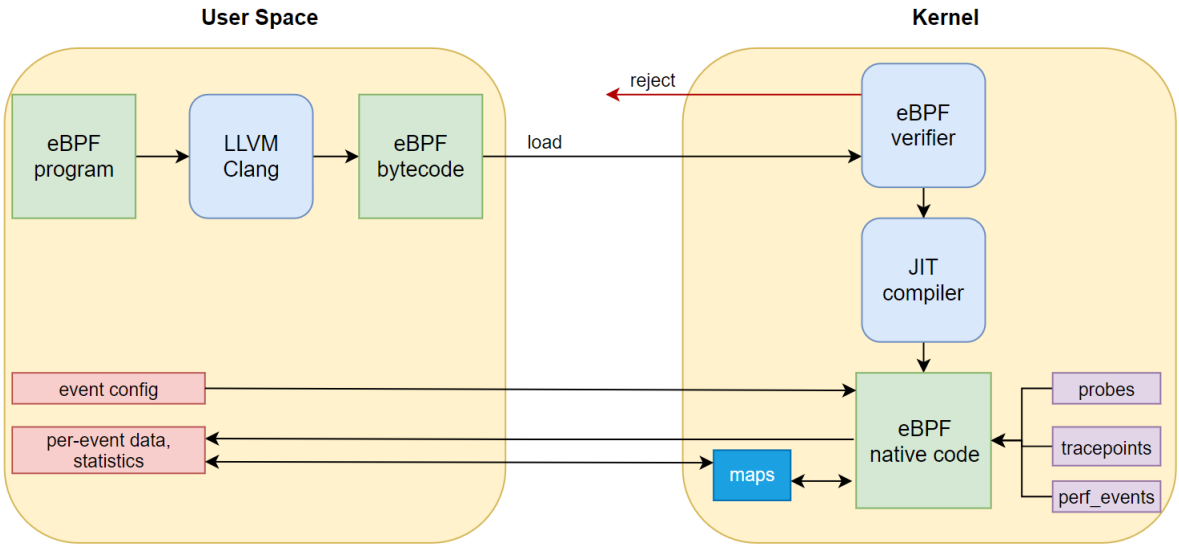


Figura 1: Arhitectură generică program BCC-eBPF

Un program eBPF rulează de fiecare dată când are loc un eveniment predefinit specificat. Prin evenimente înțelegem apeluri de sistem, evenimente în rețea, etc. Dacă avem nevoie de un tip de eveniment care nu există predefinit, putem crea un kprobe (kernel probe) sau uprobe (user probe) pentru a atașa programul oriunde în aplicații.

Pentru început, programul eBPF este compilat cu LLVM, rezultând bytecode care este încărcat în kernel. Odată ajuns în kernel, acesta trece prin două etape importante înainte de a fi executat:

- *verificare*. La acest pas se verifică dacă programul este sigur pentru rulare prin îndeplinirea unor condiții precum:
  - programul își va termina cu siguranță execuția, adică, de exemplu, nu ciclează la infinit într-o buclă. Buclele sunt permise în versiunile mai noi de kernel atât timp cât conțin o condiție de ieșire pe care se va ajunge sigur.
  - programul nu va provoca un crash al sistemului (exemplu: memory out of bounds)
  - nu există variabile neinițializate
  - dimensiunea programului se încadrează în limita impusă de sistem
- *compilare Just-in-Time (JIT)*. În această etapă, bytecode-ul rezultat din compilarea cu LLVM este tradus în cod mașină pentru o performanță mai bună.

Pentru stocarea datelor, dar și pentru schimbul de informații dintre programul eBPF și aplicația din user space se folosesc map-urile. Acestea sunt structuri în care pot fi reținute tipuri diferite de date și fiecare tip de map are următoarele caracteristici: numărul maxim de elemente, tipul map-ului, mărimea cheii [bytes], mărimea valorii [bytes].

3 OBIECTIVE

Scopul proiectului este realizarea unui tool de monitorizare a traficului care să măsoare performanța rețelei folosind eBPF-ul, care promite a fi o soluție optimă la momentul actual [4], și să rezolve neajunsurile tool-urilor deja existente.

Obiectivele proiectului sunt bine redate prin următoarele caracteristici:

- rapiditate: asigurată prin utilizarea BPF-ului
- ușurință în utilizare
- măsurători în timp real
- afișarea rezultatelor într-un mod intuitiv (grafice)
- ușurința deployment-ului

Tool-ul este unul asemănător, ca și interfață cu utilizatorul, cu alte utilitare din Linux și dispune, de asemenea, de documentație, ceea ce facilitează utilizarea acestuia.

Vorbind despre analizarea unui trafic continuu de date, avem nevoie ca tool-ul să măsoare performanța rețelei și să afișeze în timp real rezultatele pentru a putea corela fiecare eroare sau comportament nedorit al rețelei cu input-ul primit la momentul respectiv.

Având în vedere faptul că rețelele monitorizate nu sunt medii ideale de transmisie, nu putem diagnostica performanța acestora bazându-ne doar pe măsurători punctuale, rezultate obținute la un moment dat în timpul comunicării. În acest sens, vizualizarea istoricului măsurătorilor este o soluție bună prin intermediul căreia se pot observa cu mai mare ușurință instabilitățile din rețea. Astfel, afișarea rezultatelor sub formă de grafice ne permite urmărirea istoricului acestora și concluzionarea asupra eventualelor erori de comunicație.

Un ultim obiectiv al proiectului este ușurința deployment-ului soluției. Fiind un tool de monitorizare pentru rețele, ne dorim ca acesta să fie portabil, adică să poată fi rulat de pe mașini diferite.

4 CAZURI DE UTILIZARE

Proiectul a fost conceput pentru utilizarea de către echipa de testare din cadrul companiei Keysight în scopul dezvoltării proiectului de 5G (a cincea generație de tehnologie pentru rețelele celulare fără fir).

Tool-ul poate fi folosit pentru capturarea și monitorizarea traficului dintre două sisteme, în vederea testării și observării unui eventual comportament eronat al canalului de comunicație.

Cum soluțiile existente menționate în capitolul Motivație prezintă diferite neajunsuri precum overhead cauzat de tranziția user space - kernel space sau afișarea rezultatelor într-o formă mai puțin intuitivă sau măsurarea doar a anumitor metrici, proiectul actual vine cu rezolvările acestora.

Scopul tool-ului creat este de a ușura munca celor care testează o rețea prin măsurarea mai multor parametrii simultan și colectarea datelor, fără a cauza overhead, într-o bază de date persistentă, lucru care favorizează vizualizarea ulterioară a istoricului măsurătorilor și, deci, observarea cu ușurință a instabilităților.

5 ARHITECTURA PROIECTULUI

Așa cum am specificat în capitolul Obiective, pentru ușurința deployment-ului tool-ului, acesta este livrat sub formă de mașină virtuală care poate fi încărcată pe orice sistem de operare care permite rularea unei soluții de virtualizare. Această mașină virtuală poate comunica în exterior prin intermediul unei interfețe expuse de către hypervisor-ul folosit pe mașina host. Astfel, traficul care se dorește analizat trebuie redirectat către această interfață. Pentru ca acest lucru să fie posibil, este nevoie de un switch care să fie configurat să trimită o copie a pachetelor și către echipamentul care se ocupă de monitorizare.

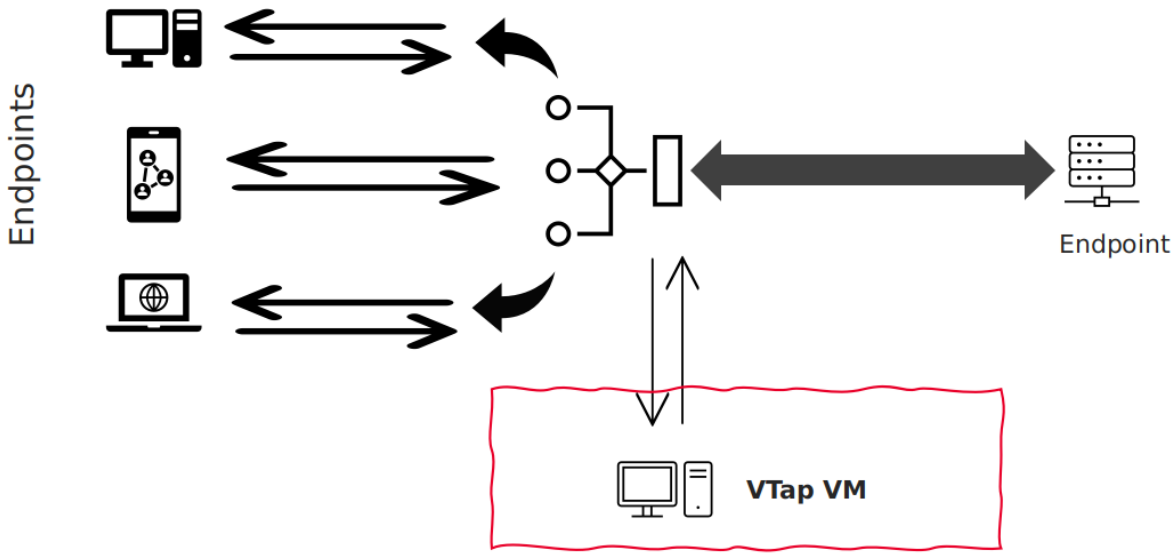


Figura 2: Caz de utilizare

5.1 Tehnologii folosite

Pentru a îndeplini obiectivele menționate anterior, am ales să folosesc tehnologii open-source precum:

- eBPF
- Python3
- BCC
- Prometheus
- Grafana

- Docker
- Ubuntu 20.04 LTS (versiune kernel 5.4.0)

eBPF-ul este tehnologia principală folosită în cadrul acestui proiect din motive de performanță, dar și pentru a experimenta cu aceasta, fiind introdusă relativ recent în kernelul Linux.

BCC este un set de instrumente pentru interfațarea cu eBPF-ul folosind Python. Este o alternativă ușor de folosit în detrimentul scrierii de cod C pentru programe eBPF.

Prometheus este o aplicație software de monitorizare a evenimentelor și alertare, fiind adesea folosită datorită ușurinței în utilizare, opțiunilor numeroase de integrare și versatilității sale. Prometheus a fost conceput pentru a satisface cerințe precum un model de date multi-dimensional, colectare scalabilă de date, simplicitate operațională și un limbaj de interogare puternic. [5]

Componente esențiale ale Prometheus sunt:

- *serverul Prometheus*: partea cea mai importantă care colectează metricile (proces numit și scraping) și le stochează. Serverul Prometheus preia în mod activ datele de la aplicații la un anumit interval de timp (numit și scrape interval), ceea ce reprezintă un avantaj în comparație cu alte tool-uri asemănătoare care așteaptă ca aplicațiile să trimită metricile. În cazul Prometheus-ului, este de ajuns ca aplicația să își expună datele printr-un server HTTP.
- *clientul* este cel care trebuie să expună datele ce vor fi colectate într-un format pe care Prometheus să îl înțeleagă. Acest lucru poate fi realizat în două moduri, fie prin librării specifice clientului Prometheus, fie prin exporters.
- *push gateway* se comportă ca un fel de cache care este util în cazurile în care vrem să păstrăm datele unor job-uri a căror durată de viață este mai scurtă decât intervalul de scraping.
- *managerul de alerte* care oferă posibilitatea definirii de alerte de către utilizator pentru a primii notificări în cazul în care sunt detectate date care nu respectă anumite limite sau orice alt comportament considerat anormal.
- *vizualizarea*: datele colectate pot fi vizualizate în formă de grafice sau tabele direct în Prometheus sau chiar trimise mai departe pentru vizualizarea externă prin intermediul API-ului HTTP. De obicei, împreună cu Prometheus, pentru vizualizare, se folosește Grafana.

Grafana este o aplicație web care permite analizarea și vizualizarea datelor din mai multe surse. Aceasta prezintă avantajul ușurinței în utilizare, fiind intuitivă de folosit, și custo-

mizabilității în ceea ce privește modul de vizualizare al datelor. Grafana este unul dintre cele mai folosite software-uri de vizualizare, oferind următoarele feature-uri: [6]

- *vizualizarea*. Există multe opțiuni de vizualizare care contribuie la crearea unui dashboard Grafana.
- *alerte*. La fel ca în cazul Prometheus, Grafana pune la dispoziție un sistem de alerte pentru a putea primii notificări când există un comportament anormal în parcursul datelor.
- *adnotări*. Grafana permite scrierea de notițe direct pe grafice, oferind posibilitatea marcării unor evenimente importante "la vedere".

Docker este cel mai popular și des folosit tip de containere. Prin container ne referim la un software care izolează o aplicație cu toate dependențele sale astfel încât să poată fi rulată independent de mediul gazdă. Containerul poate fi pus pe orice sistem de operare pe care rulează Docker. Într-un container putem pune atât microservicii sau părți independente ale unei aplicații, cât și întreaga aplicație. [7]

5.2 Componentele software ale proiectului

Din punct de vedere software, proiectul cuprinde următoarele componente principale:

- script de monitorizare Python
- controller pentru tool-ul de monitorizare
- container Docker cu microserviciul Prometheus
- container Docker cu microserviciul Grafana
- server de scraping pentru Prometheus
- teste de acceptanță pentru măsurătorile realizate

Cea mai importantă componentă a proiectului este un script Python care folosește tehnologia eBPF prin intermediul framework-ului BCC pentru a monitoriza traficul dorit de utilizator și a măsura metricile de performanță corespunzătoare cerințelor.

Script-ul Python trimite către compilare, prin intermediul BCC-ului, cod C pentru eBPF, iar măsurătorile sunt colectate în eBPF Maps si preluate în Python. Mai departe, acestea sunt prelucrate în script și trimise către serverul Prometheus.

Controller-ul este componenta care se ocupă de manipularea programului de monitorizare aducând câteva funcționalități utile în vederea facilitării utilizării acestuia.

Colectarea datelor măsurate de programul de monitorizare se realizează de către un server Prometheus și sunt afișate într-o pagină web cu ajutorul unui server Grafana. Atât serverul Prometheus, cât și cel de Grafana rulează în două containere docker diferite, interconectate.

Pentru a pune la dispoziție Prometheus-ului metricile măsurate, am folosit un server simplu HTTP pentru expunerea datelor.

Pentru validarea rezultatelor obținute de către program am creat câteva teste de acceptanță pentru verificarea bunei funcționări a logicii programului în diferite cazuri de utilizare.

În imaginea următoare, am prezentat schema minimalistă a interconectării componentelor discutate.

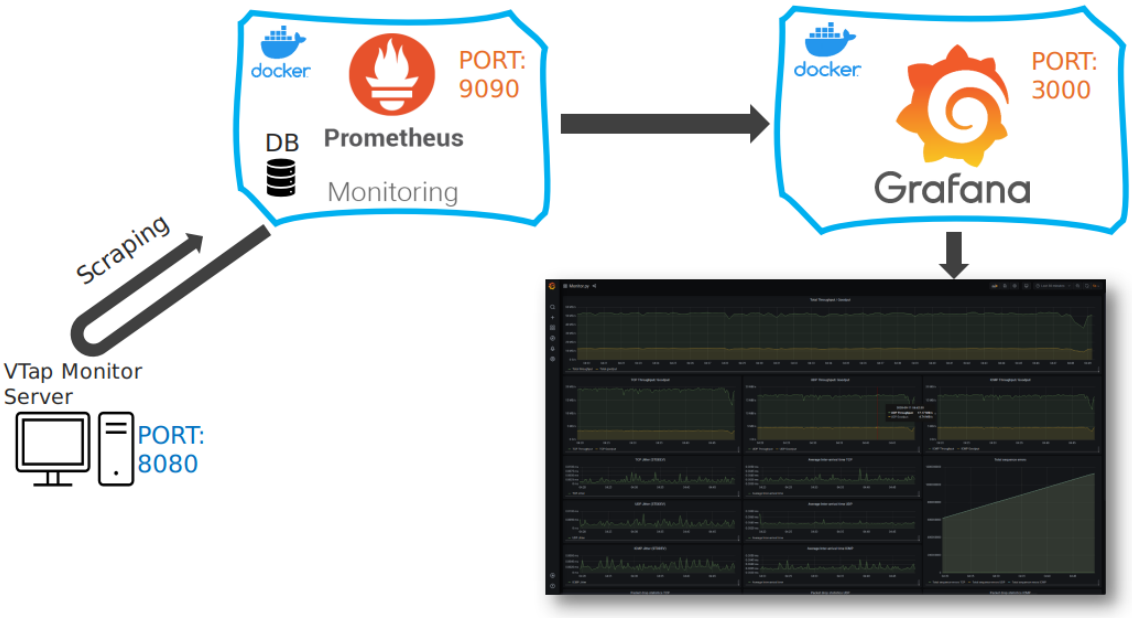


Figura 3: Componente integrate

6 FUNCȚIONALITATE CURENTĂ

Tool-ul respectă toate obiectivele propuse inițial. Produsul final este un tool de monitorizare al rețelei, capabil să selecteze stream-ul de pachete în funcție de dorințele utilizatorului. Acesta prezintă rezultatele intuitiv utilizatorului și este livrat sub forma unei mașini virtuale pregătită să fie încărcată pe sistemul unde se dorește monitorizarea.

Îmbunătățirile aduse de acest proiect față de soluțiile deja existente sunt prezentate în subcapitolele ce urmează.

6.1 Măsurători realizate

Pentru a estima performanța unei rețele, acest proiect realizează următoarele măsurători:

- Throughput
- Goodput
- Jitter
- Erori de secvență

Throughput-ul reprezintă rata de transferare a pachetelor în rețea în unitatea de timp. Acesta poate fi măsurat în biți/Bytes/pachete per secundă.

Goodput-ul reprezintă, spre deosebire de throughput, rata de transfer doar a conținutului util efectiv al pachetelor.

Jitter-ul reprezintă deviația timpului de sosire dintre pachete. Presupunând că avem un flux de pachete trimise la același interval de timp, acestea pot ajunge totuși decalate din cauza congestionării rețelei sau a altor instabilități, ținând cont și de faptul că mediile de transmisie nu sunt ideale. Este esențial ca valoarea jitter-ului să fie una cât mai mică în special când vine vorba de transmisii video sau audio.

Erorile de secvență se referă la inordinea sosirii pachetelor la destinație, care poate fi cauzată, atât de reordonări ale frame-urilor cât și de pierderea anumitor pachete.

6.2 Vizualizare rezultate

O funcționalitate importantă a proiectului pentru logica acestuia este vizualizarea rezultatelor. Așa cum am menționat anterior, nu este de ajuns să știm doar rezultatele la un anumit moment de timp pentru a diagnostica rețeaua. Astfel, am decis că afișarea acestora sub formă de grafice individuale care se actualizează în timp real (la un interval de timp predefinit) este o soluție optimă pentru a putea urmări evoluția în timp a datelor. În acest sens, am creat un dashboard customizabil în Grafana.

6.3 Controller

Având în vedere că tool-ul de monitorizare creat ar putea avea mai multe instanțe pe același sistem, trebuie evitată folosirea aceluiași port pentru clientul care trimite datele către Prometheus pentru a nu întâmpina probleme de intercalare/suprapunere a datelor.

De asemenea, programul de monitorizare poate fi folosit și în modul detached (fără afișare la consolă), ceea ce poate duce la managementul ineficient al instanțelor. Pentru rezolvarea acestor tipuri de probleme am creat controllerul.

6.4 Deployment

Soluția implementată pentru asigurarea deployment-ului facil al tool-ului este încapsularea programului într-o mașină virtuală cu Ubuntu 20.04 (distribuție de Linux) ca sistem de operare, livrată sub format OVF. Această mașină conține environmentul cu toate dependențele eBPF-ului, ale BCC-ului și toate celelalte utilitare și configurații necesare rulării și va putea fi încărcată pe orice sistem de operare care permite virtualizare (VirtualBox/VMware).

7 IMPLEMENTARE

7.1 Program eBPF

Componenta principală a proiectului este reprezentată de un program scris în Python care interacționează cu eBPF-ul prin intermediul framework-ului BCC. Acest framework presupune încărcarea în mașina virtuală eBPF a unui cod C dat ca parametru sub formă de string, urmând să fie specificată funcția care să fie executată pentru un anumit tip de evenimente (de exemplu, funcția de numărare a pachetelor la fiecare pachet ajuns pe un socket). Stocarea codului C într-un string în Python oferă posibilitatea modificării dinamice a acestuia la runtime.

Mai jos se poate observa cum se realizează în cadrul proiectului integrarea eBPF-ului în Python și specificarea funcției ce trebuie declanșată la interceptarea unor evenimente în rețea.

```
bpf_text =
"""

        #include ...
        [...]
        int process_packets(struct __sk_buff *skb) {
            [...]
        }
        [...]
"""

[...]
bpf = BPF(text=bpf_text)
ffilter = bpf.load_func("process_packets", BPF.SOCKET_FILTER)
[...]
```

Listing 7.1: Integrare BPF în Python

Inițial, cu ajutorul constructorului *BPF* se creează un obiect de acest tip, prin intermediul căruia se definește programul BPF și se poate interacționa mai apoi cu outputul său. Programul BPF reținut în variabila *bpf_text* este scris în C sub forma unui string Python.

Odată creat obiectul BPF, prin *load_func()* specificăm funcția (implementată în codul

BPF) ce va fi executată pentru fiecare eveniment interceptat. Deoarece cazul de utilizare al tool-ului impune monitorizarea traficului care circulă prin anumite interfețe ale sistemului, trebuie specificat faptul că această funcție se va executa doar la declanșarea evenimentelor pe aceste interfețe. Acest lucru este posibil în eBPF prin intermediul funcției *attach_raw_socket*() apelată pentru fiecare dintre interfețele date ca parametru tool-ului.

Funcția declanșată la primirea unui pachet va executa calcule asupra acestuia, iar stocarea rezultatelor și valorilor temporare se face în map-uri BPF_ARRAY care se comportă asemenea tipului de date vector din C.

```
1  BPF_ARRAY(packet_count, u64, 4);
2  BPF_ARRAY(bytes_sent, u64, 4);
3  BPF_ARRAY(bytes_sent_no_headers, u64, 4);
4
5  BPF_ARRAY(prev_time_jitter_hash, u64, 4);
6  BPF_ARRAY(jitter_index_hash, u64, 4);
7
8  BPF_ARRAY(jitter_values_tcp, u64, JITTER_ARRAY_SIZE);
9  BPF_ARRAY(jitter_values_udp, u64, JITTER_ARRAY_SIZE);
10 BPF_ARRAY(jitter_values_icmp, u64, JITTER_ARRAY_SIZE);
11
12 BPF_ARRAY(seq_hash, struct seq_memory, 4);
```

Listing 7.2: Map-uri eBPF folosite

Am folosit array-uri pentru a stoca numărul total de pachete, numărul de bytes primiți, întârzierea dintre pachete, dar și numărul erorilor de secvență pentru fiecare protocol în parte. BPF suportă și alte tipuri de map-uri precum BPF_TABLE, BPF_HASH, dar am ales să folosesc BPF_ARRAY deoarece folosește indexare după întregi, iar căutarea este optimizată oferind o viteză de citire/scriere a elementelor mult mai rapidă decât celelalte tipuri de date.

Un alt aspect de menționat este faptul că funcția ce definește comportamentul la primirea unui pachet pe interfețele monitorizate, *process_packets()*, primește ca argument o structură fundamentală pentru rețelistică în Linux, *__sk_buff*. Această structură este o "clonă" accesibilă user-ului a structurii *sk_buff* care conține metadate despre fiecare pachet și un pointer către pachetul recepționat.

Pentru extragerea de informații din headerele unui pachet am folosit funcția *cursor_advance()* cu un cursor folosit ca iterator peste câmpul *data* al structuri *__sk_buff *skb*.

```
u8 *cursor = 0;
struct ethernet_t *ethernet = cursor_advance(cursor,
                                             sizeof(*ethernet));
struct ip_t *ip = cursor_advance(cursor, sizeof(*ip));
struct tcp_t *tcp;
struct udp_t *udp;
struct icmp_t *icmp;

int protocol = ip->nextp;

if (protocol == IPPROTO_UDP) {
    udp = cursor_advance(cursor, sizeof(*udp));
} else if (protocol == IPPROTO_TCP) {
    tcp = cursor_advance(cursor, sizeof(*tcp));
} else if (protocol == IPPROTO_ICMP) {
    icmp = cursor_advance(cursor, sizeof(*icmp));
}
```

Listing 7.3: Extragerea headerelor din pachet

Pentru a accesa datele din pachet am folosit funcția *load_word()* care întoarce 32 de biți de la offset-ul dat ca parametru față de începutul pachetului.

7.2 Opțiuni de monitorizare - fișier de configurație JSON

Programul dezvoltat este capabil să realizeze măsurătorile dorite doar pentru anumite pachete în funcție de preferințele utilizatorului (preferințe legate de IP-ul sursă, porturile sursă/destinație, protocol sau orice combinări ale acestora). Aceste preferințe alături de alegerea măsurătorilor ce se doresc a fi afișate utilizatorului și alte argumente necesare programului (care vor fi explicate mai târziu pe parcursul acestui capitol) se specifică prin intermediul unui fișier în format JSON a cărui cale trebuie dată ca parametru.

Un exemplu de fișier json necesar programului este următorul:

```
{
    "RULES": [
        //source_ip destination_ip source_port destination_port
    protocol
        "1.1.1.1 2.2.2.2 11 22 TCP",
        "3.3.3.3 - - - [TCP,UDP]"
    ],
```

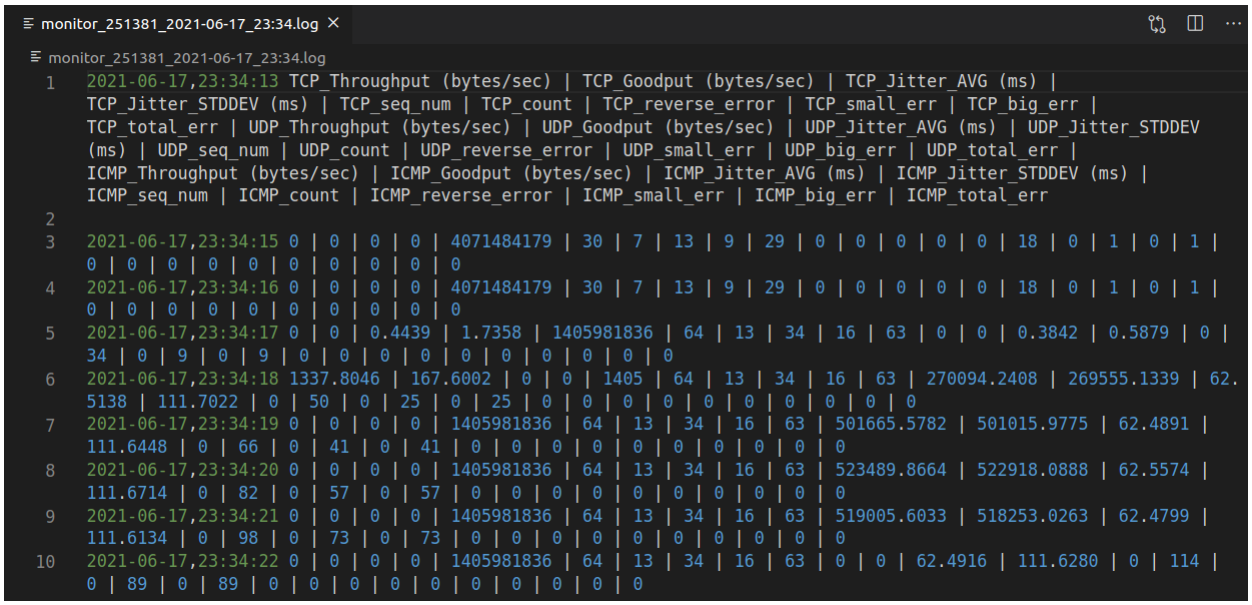
```
7
8       "DISPLAY": {
9           "Throughput": "True",
10          "Goodput": "True",
11          "Jitter": "True",
12          "SEQ_ERRORS": "True"
13      },
14
15      "PATTERN": "0xE7A5C318",
16      "OFFSET": 4,
17      "SEQUENCE_THRESHOLD": 2,
18      "MONITOR_SERVER": "True",
19      "LOG_DATA": "False"
20  }
```

Listing 7.4: Exemplu fișier json necesar programului

În lista *RULES* se specifică regulile pe care dorim să le îndeplinească pachetele pentru care vrem să măsurăm anumite metrici pe principiul IPTABLES. Pentru fiecare pachet ajuns pe una din interfețele monitorizate se verifică regulile până când se găsește una pe care o respectă sau până la final (caz în care este ignorat). Regulile sunt de forma *"ip_sursă ip_destinație port_sursă port_destinație protocol"*, unde fiecare dintre aceste elemente poate fi o listă. În cazul în care nu ne interesează valoarea unuia dintre elemente se poate pune "-". De exemplu, o regulă de forma *"1.1.1.1 2.2.2.2 - 22 [TCP,UDP]"* înseamnă că vor fi considerate toate pachetele TCP sau UDP care merg de la sursa 1.1.1.1 către 2.2.2.2, având portul destinație 22. În acest caz portul sursă nu contează.

În dicționarul *DISPLAY* se pot alege măsurătorile dorite prin specificarea "True" sau "False" în dreptul fiecărei metrici.

Tool-ul creat oferă și posibilitatea salvării output-ului în fișiere de logging. Pentru realizarea acestui lucru, utilizatorul trebuie să marcheze cu "True" intrarea specifică din fișierul de configurație JSON, *LOG_DATA*. Pentru fiecare instanță a tool-ului se creează câte un fișier de logging ce conține data, ora și minutul în denumire. Un exemplu de astfel de fișier este următorul:



monitor_251381_2021-06-17_23:34.log

```
2021-06-17,23:34:13 TCP_Throughput (bytes/sec) | TCP_Goodput (bytes/sec) | TCP_Jitter_AVG (ms) | TCP_Jitter_STDDEV (ms) | TCP_seq_num | TCP_count | TCP_reverse_error | TCP_small_err | TCP_big_err | TCP_total_err | UDP_Throughput (bytes/sec) | UDP_Goodput (bytes/sec) | UDP_Jitter_AVG (ms) | UDP_Jitter_STDDEV (ms) | UDP_seq_num | UDP_count | UDP_reverse_error | UDP_small_err | UDP_big_err | UDP_total_err | ICMP_Throughput (bytes/sec) | ICMP_Goodput (bytes/sec) | ICMP_Jitter_AVG (ms) | ICMP_Jitter_STDDEV (ms) | ICMP_seq_num | ICMP_count | ICMP_reverse_error | ICMP_small_err | ICMP_big_err | ICMP_total_err

2021-06-17,23:34:15 0 | 0 | 0 | 0 | 4071484179 | 30 | 7 | 13 | 9 | 29 | 0 | 0 | 0 | 0 | 0 | 18 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2021-06-17,23:34:16 0 | 0 | 0 | 0 | 4071484179 | 30 | 7 | 13 | 9 | 29 | 0 | 0 | 0 | 0 | 0 | 18 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2021-06-17,23:34:17 0 | 0 | 0.4439 | 1.7358 | 1405981836 | 64 | 13 | 34 | 16 | 63 | 0 | 0 | 0.3842 | 0.5879 | 0 | 34 | 0 | 9 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2021-06-17,23:34:18 1337.8046 | 167.6002 | 0 | 0 | 1405 | 64 | 13 | 34 | 16 | 63 | 270094.2408 | 269555.1339 | 62.5138 | 111.7022 | 0 | 50 | 0 | 25 | 0 | 25 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2021-06-17,23:34:19 0 | 0 | 0 | 0 | 1405981836 | 64 | 13 | 34 | 16 | 63 | 501665.5782 | 501015.9775 | 62.4891 | 111.6448 | 0 | 66 | 0 | 41 | 0 | 41 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2021-06-17,23:34:20 0 | 0 | 0 | 0 | 1405981836 | 64 | 13 | 34 | 16 | 63 | 523489.8664 | 522918.0888 | 62.5574 | 111.6714 | 0 | 82 | 0 | 57 | 0 | 57 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2021-06-17,23:34:21 0 | 0 | 0 | 0 | 1405981836 | 64 | 13 | 34 | 16 | 63 | 519005.6033 | 518253.0263 | 62.4799 | 111.6134 | 0 | 98 | 0 | 73 | 0 | 73 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
2021-06-17,23:34:22 0 | 0 | 0 | 0 | 1405981836 | 64 | 13 | 34 | 16 | 63 | 0 | 0 | 62.4916 | 111.6280 | 0 | 114 | 0 | 89 | 0 | 89 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0
```

Figura 4: Exemplu de fișier de logging

În ceea ce privește implementarea preferințelor utilizatorului legate de stream-ul de date ce trebuie analizat, acest lucru este realizat prin generare de cod dinamic din Python, așa cum am arătat mai jos.

```
bpf_text =
"""
        [...]
        FILTER_OUTCOMING
        FILTER_RULES
        [...]
"""

def f_filter_outcoming(bpf_text):
    FILTER_outcoming = ""
    [...]
    if "saddr" in FILTER_outcoming:
        FILTER_outcoming += " && "

    FILTER_outcoming = FILTER_outcoming + "(saddr == {})".format(int(IPv4Address(ip)))
    [...]
    return bpf_text.replace('FILTER_OUTCOMING', FILTER_outcoming)

bpf_text = f_filter_outcoming(bpf_text)
bpf_text = f_filter(bpf_text, rules_json)
bpf = BPF(text=bpf_text)
```

Listing 7.5: Generare cod dinamic eBPF

Prin declararea unor cuvinte cheie (precum "FILTER_RULES") în programul eBPF a fost posibilă substituirea dinamică a acestora cu string-uri construite ulterior pornirii programului în funcție de parametrii dați la rulare.

O alternativă la această variantă ar fi fost trimiterea valorilor alese de utilizator către programul eBPF prin intermediul map-urilor, însă această metodă ar fi generat overhead de timp și spațiu, în ciuda faptului că ar fi fost mai simplu de implementat.

Făcând abstracție de opțiunile ce pot fi completate în fișierul de configurație, programul mai primește și alte argumente ce se dau direct la rularea acestuia în linia de comandă precum:

- interfața/-ele pe care se dorește monitorizarea → parametrul **-i**. Default, sunt monitorizate toate interfețele sistemului.
- intervalul de timp pe care să fie calculat throughput-ul → parametrul **-t**. Default, este calculat la fiecare secundă scursă.
- intervalul de timp pentru calculul jitter-ului → parametrul **-j**. Default, este calculat la fiecare secundă scursă.
- portul pentru serverul de colectare a datelor și trimitere către Prometheus → parametrul **-server_port**. Default, acest server este deschis pe portul 8080.
- calea relativă către fișierul de configurație de format json în care sunt date alte detalii utile rulării programului → parametrul **-conf**. Default, este considerat fișierul "config.json" din folderul curent.
- posibilitatea afișării throughput-ului/goodput-ului în biți/secundă → parametrul **-b**. Default, acestea sunt măsurate în Bytes/secundă.

7.3 Controller

O altă componentă a proiectului este controller-ul pentru programul de monitorizare, un script care oferă posibilitatea pornirii "sigure" a mai multor instanțe ale tool-ului, fără să existe conflicte între acestea/suprapuneri de date în cadrul serverului de colectare a acestora.

Controller-ul permite pornirea, oprirea și afișarea statusului instanțelor. Parametrii de rulare ai controller-ului sunt:

- **start**: poate fi urmat de **--detached** care oferă posibilitatea rulării programului în background (fără afișarea output-ului la consolă); după **start** [**--detached**] ur-

mează parametrii programului de monitorizare așa cum i-am prezentat anterior acestei secțiuni
- **stop**: oprește instanțele în rulare ale tool-ului
- **status**: afișează instanțele (alături de pid-urile proceselor) pentru ușurința managementului acestora

7.4 Măsurători de performanță a rețelei

Throughput-ul/Goodput-ul sunt măsurate contorizând efectiv numărul de biți primiți per fiecare interval de timp consumat, în cazul în care intervalul este dat ca parametru de utilizator sau per fiecare secundă, în caz contrar. Pentru throughput am considerat dimensiunea totală a unui pachet (inclusiv headerul de ethernet), în timp ce pentru goodput am considerat doar dimensiunea payload-ului de la nivelul de transport (layer 4) din stiva OSI (Open Systems Interconnection) (partea efectivă de date, payload TCP/UDP/ICMP).

În ceea ce privește **jitter-ul**, l-am calculat ca fiind deviația standard pe baza timpilor consumați între fiecare două pachete consecutive primite. Am ales să afișez atât o medie a acestor timpi cât și deviația standard calculată după formula:

$$STDDEV = \sqrt{\frac{\sum \mid x - \overline{x} \mid^2}{n}}$$

unde:

- x este o valoare din setul de date, în acest caz, întârzierea dintre două pachete succesive
- $\overline{x}$ este media tuturor valorilor din setul de date, adică media întârzierilor dintre pachete
- n este numărul total de valori pe care se aplică formula

Un alt aspect important de menționat este faptul că pentru un timp mai mare de două secunde între două pachete consecutive, am considerat că există o întrerupere a flow-ului de pachete. În acest caz, este resetată valoarea jitter-ului astfel încât să fie luate în considerare doar pachetele ulterioare acelei pauze. Am făcut acest lucru pentru a nu creea falsa impresie a unei probleme în rețea prin afișarea unui jitter foarte mare. La fel ca în cazul throughput-ului, am implementat posibilitatea alegerii de către utilizator a intervalului de timp pe care să fie realizat calculul.

Legat de **packet loss**, programul afișează câteva statistici cu referire la acesta, și anume:

- numărul curent de secvență
- numărul total de pachete primite de la începutul monitorizării
- numărul de erori "de inversare": numărul de pachete care îndeplinesc la momentul primirii condiția conform căreia numărul de secvență este mai mic decât cel al pachetului primit anterior
- erori "mici" și "mari": numărul de pachete pentru care diferența dintre numărul lor de secvență și cel anterior este mai mică, respectiv mai mare, decât o limită aleasă de utilizator. Această limită poate fi introdusă în câmpul *SEQUENCE_THRESHOLD* în fișierul de configurație în format JSON despre care am vorbit și într-o secțiune anterioară.

Așa cum știm deja, în cadrul pachetelor de tip UDP sau ICMP nu există un câmp dedicat numărului de secvență așa cum avem în cazul pachetelor TCP. Astfel, am considerat posibilitatea adăugării de către utilizator a numărului de secvență în payload-ul pachetului. Pentru a face posibil acest caz, există două variante:

- specificarea unui offset relativ la începutul payload-ului
  SAU
- specificarea unui pattern și a unui offset relativ la începutul pattern-ului, loc în care se va găsi numărul de secvență.

Cea de-a doua variantă (cea cu specificarea unui pattern) este utilă în cazul în care alte encapsulări sunt sau pot fi adăugate pachetului. Atât pattern-ul cât și offset-ul sunt pe 32 de biți și trebuie specificate, de asemenea, în cadrul fișierului JSON, în câmpurile corespunzătoare, *PATTERN* și *OFFSET*. Dacă nu se dorește măsurarea statisticilor referitoare la packet loss atunci câmpurile referitoare la pattern și offset vor fi completate cu 0xFFFFFFFF și, respectiv, -1.

```
if (protocol == PROTO_TCP) {
    seq_num = tcp->seq_num;
} else if (protocol == PROTO_UDP || protocol == PROTO_ICMP) {
    u32 payload_offset = ETH_HLEN + ip_header_length + 8;

    if (pattern != -1) {
        u32 test_pattern;
        // search for given pattern in first 5000 bytes of data
        for (int i = 0; i < 5000; i++) {
            test_pattern = load_word(skb, payload_offset + i);

```

```
12              if (test_pattern == pattern) {
13                  seq_num = load_word(skb, payload_offset +
14                                                  i + offset);
15                  break;
16              }
17          }
18      } else {
19          seq_num = load_word(skb, payload_offset + offset);
20      }
21  }
```

Listing 7.6: Extragere număr de secvență din pachete

Anterior am arătat cum am obținut numărul de secvență în programul eBPF în funcție de protocol ținând cont de variantele pe care tocmai le-am menționat.

În cazul pachetelor de tip TCP, am extras numărul de secvență direct din structura *tcp* preluată așa cum am prezentat în 7.3. În cazul pachetelor UDP sau ICMP, pentru a ajunge la acest număr a trebuit fie să accesez direct offset-ul la care se găsește în payload, fie să caut pattern-ul care indică cu ajutorul unui offset poziția acestuia. Pentru partea de pattern matching am utilizat funcția *load_word()* în eBPF pentru a căuta în primii 5000 de bytes ai payload-ului pachetului, constrângere impusă de limitările eBPF-ului legate de spațiul ocupat de program.

7.5 Prometheus și Grafana

Vizualizarea, ca funcționalitate a tool-ului dezvoltat, este realizată prin intermediul Prometheus și Grafana. Din punct de vedere al implementării, am folosit un client Prometheus în Python care găzduiește un server HTTP cu ajutorul căruia datele programului pot fi colectate (scrap-uite) de serverul Prometheus.

Pentru a putea fi colectate de Prometheus, datele trebuie să fie într-un format pe care acesta să îl înțeleagă. Există 4 tipuri de metrici ce pot fi folosite: Counter, Gauge, Summary și Histogram. Eu am folosit tipul Gauge deoarece permite rezultate care pot atât să crească cât și să scadă.

Clientul Prometheus este pornit la lansarea în execuție a programului de monitorizare folosind librăria *prometheus_client* din Python cu funcția *start_http_server()* apelată pe portul dat ca parametru în linia de comandă la rularea programului. Acest lucru este posibil doar dacă utilizatorul marchează cu True câmpul *MONITOR_SERVER* din fișierul de configurație pentru a-și exprima preferința legată de vizualizarea datelor.

Rezultatele sunt afișate într-o pagină web prin intermediul unui server Grafana. Acest server se conectează la cel de Prometheus, extrage datele și le afișează folosind dashboard-ul creat care poate fi customizat de către utilizator.

Cele două servere, Prometheus și Grafana, rulează independent, fiecare într-un container Docker după configurațiile următoare:

```
version: "3"
services:
    prometheus:
    container_name: prometheus-svc
    image: prom/prometheus
    network_mode: host
    ports:
    - "9090:9090"
    command: --config.file=/etc/prometheus/prometheus.yaml --web.
   enable-admin-api --web.listen-address=:9090
    volumes:
    - ./prometheus.yaml:/etc/prometheus/prometheus.yaml
    grafana:
    container_name: grafana
    depends_on: [prometheus]
    network_mode: host
    image: grafana/grafana:7.1.5
    ports:
    - "3000:3000"
    environment:
    - GF_AUTH_DISABLE_LOGIN_FORM=true
    - GF_AUTH_ANONYMOUS_ENABLED=true
    - GF_AUTH_ANONYMOUS_ORG_NAME=Main Org.
    - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
    - GF_USERS_ALLOW_SIGN_UP=false
    - GF_SERVER_HTTP_PORT=3000

```

Listing 7.7: Fișier de configurație Docker

7.6 Provocări și probleme întâlnite

De-a lungul dezvoltării acestui proiect am întâmpinat destul de multe provocări având în vedere tehnologia nouă folosită, eBPF-ul.

O primă provocare, și probabil cea mai însemnată, a fost cercetarea în vederea stabilirii potențialului acestei tehnologii pentru scopul propus pentru proiect, având în vedere faptul

că este o tehnologie relativ nouă și nu foarte utilizată. Documentația este una destul de sărăcăcioasă din care lipsesc destul de multe explicații importante. Există în schimb câteva mici exemple de programe, cu ajutorul cărora poți deduce sau intui utilitatea și modul de folosire al anumitor funcții.

Odată stabilite potențialul și câteva detalii generale legate de această tehnologie, urmează scrierea unui prim program funcțional eBPF. Pentru acest lucru trebuie realizate câteva configurații de environment, flag-uri specifice kernelului Linux ce trebuie să fie specificate pentru a putea rula un astfel de program [1]. Primul program pe care l-am scris în eBPF era capabil să numere toate pachetele venite pe rețea.

În ceea ce privește partea de implementare, un exemplu de problemă întâlnită a fost în momentul în care am ajuns la partea de implementare a "filtrării" pachetelor astfel încât măsurătorile să fie realizate doar pentru anumite flow-uri de date după preferințele utilizatorului. Soluția inițială ar fi fost folosirea map-urilor pentru trimiterea valorilor către programul eBPF însă această metodă ar fi generat overhead de timp ceea ce nu era în concordanță cu scopul final principal al tool-ului (cel de rapiditate). Astfel, am descoperit posibilitatea generării de cod dinamic, o soluție mai dificilă de implementat și mai greu de înțeles ulterior de alți eventuali dezvoltatori, dar optimă.

Pe parcursul dezvoltării acestui proiect am înțeles că eBPF-ul impune destul de multe constrângeri ca urmare a faptului că programele rulează direct în kernel. Astfel, înainte de a fi încărcat pentru executare, programul trece printr-o etapă de validare a mai multor aspecte precum: toate variabilele trebuie să fie inițializate, nici o buclă infintă nu trebuie să existe, accesarea pointer-ilor nu trebuie să creeze probleme, etc. Versiunea de kernel Linux folosită în acest proiect este 5.4 pentru a avea constrângeri mai relaxate, și anume suport pentru bucle finite (verifier-ul se ocupă de validarea faptului că există condiție de ieșire din buclă și se va ajunge la aceasta indiferent de parcursul programului).

De asemenea, pe lângă constrângerea legată de versiunea de kernel, mai există o constrângere impusă tool-ului la partea de pattern matching pentru calculul erorilor de secvență. Având în vedere că un program eBPF impune o limită de spațiu ocupat, tool-ul vine cu constrângerea introducerii numărului de secvență în cazul pachetelor UDP sau ICMP în primii 5000 bytes ai payload-ului.

O altă provocare a fost partea de deployment. Pentru acest lucru există două soluții: container docker sau VM. Considerând partea de rapiditate, docker este o soluție mai bună, însă există două motive principale pentru care am ales să înglobez tool-ul într-o mașină virtuală:

- o mașină virtuală oferă o mai mare izolare. În cazul tool-ului creat, mașina virtuală a fost singura soluție deoarece aveam nevoie să păstrez configurațiile sistemului pentru

[1] `https://github.com/iovisor/bcc/blob/master/INSTALL.md#kernel-configuration`.

a putea rula eBPF-ul.
- integrarea aplicației în Docker nu ar fi permis deployment-ul pe un al sistem de operare exceptând Linux. În cazul mașinii virtuale, totul este integrat și nu depinde de sistemul de operare al host-ului.

Un ultim lucru de menționat pe partea de provocări întâlnite este dificultatea în procesul de debugging și faptul că, fiind o tehnologie nefolosită încă pe scară largă, de multe ori nu se găsesc erori rezolvate.

8 PREZENTAREA REZULTATELOR

Așa cum am specificat și în capitolele anterioare, tool-ul oferă posibilitatea redării rezultatelor sub formă de metrici punctuale, afișate periodic în consolă, dar și afișarea istoricului măsurătorilor sub formă de grafice.

8.1 CLI

Afișarea în terminal este de forma prezentată în imaginea de mai jos.

```
Jitter[STDDEV] TCP: 0.0026 ms
Average TCP: 0.0054 ms

Jitter[STDDEV] UDP: 0.0055 ms
Average UDP: 0.0062 ms

Jitter[STDDEV] ICMP: 0.0043 ms
Average ICMP: 0.0059 ms


THROUGHPUT measurement on veth1 each second:
TCP bitrate [MBytes/sec]:  11.5944
UDP bitrate [MBytes/sec]:  10.1840
ICMP bitrate [MBytes/sec]:  10.1840

GOODPUT measurement on veth1 each second:
TCP bitrate [MBytes/sec]:  2.1080
UDP bitrate [MBytes/sec]:  2.8094
ICMP bitrate [MBytes/sec]:  2.8093
ICMP
        seq_num: 181
        count: 3953518
        reverse_err: 33935
        small_err: 3919582
        big_err: 0
        total_sequence_errors: 3953517
TCP
        seq_num: 570
        count: 3955242
        reverse_err: 5668
        small_err: 4
        big_err: 1718
        total_sequence_errors: 7390
UDP
        seq_num: 141
        count: 3953538
        reverse_err: 67887
        small_err: 0
        big_err: 3885649
        total_sequence_errors: 3953536
```

Figura 5: Exemplu output în consolă

Rezultatele sunt actualizate la consumarea intervalului de timp ales (dacă este ales un interval pentru care să fie realizate măsurătorile, în cazul throughput-ului/goodput-ului și jitter-ului) sau la trecerea fiecărei secunde (în mod default).

8.2 Prometheus

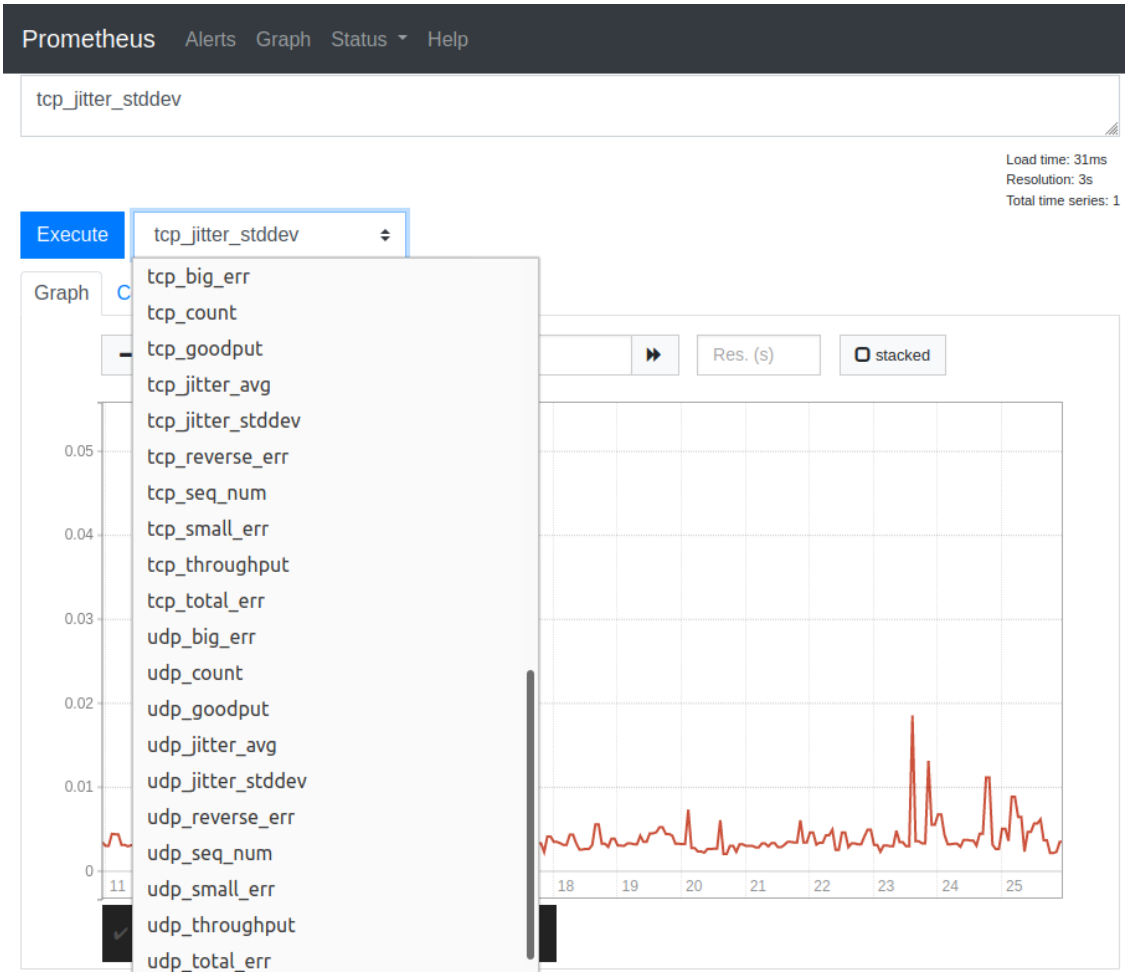


Figura 6: Vizualizare Prometheus

În Prometheus pot fi vizualizate grafice minimale pentru toate metricile măsurate. Datele sunt scrap-uite la fiecare 5 secunde din clientul creat în programul de monitorizare.

8.3 Grafana

În imaginile de mai jos este prezentat dashboard-ul Grafana creat.

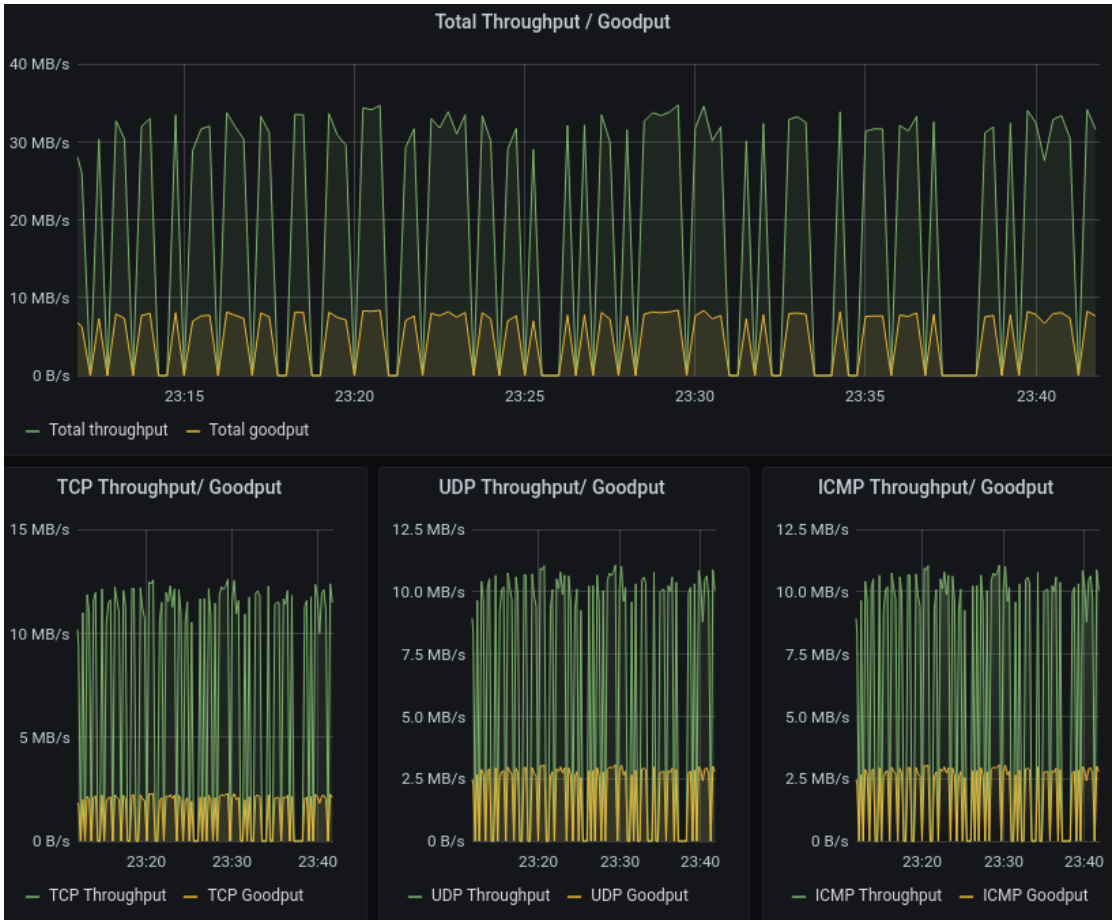


Figura 7: Vizualizare throughput Grafana

Throughput-ul și goodput-ul sunt afișate pe aceleași grafice. Unul dintre grafice reprezintă throughput-ul și goodput-ul total, iar celelalte 3 sunt măsurătorile per fiecare protocol în parte.

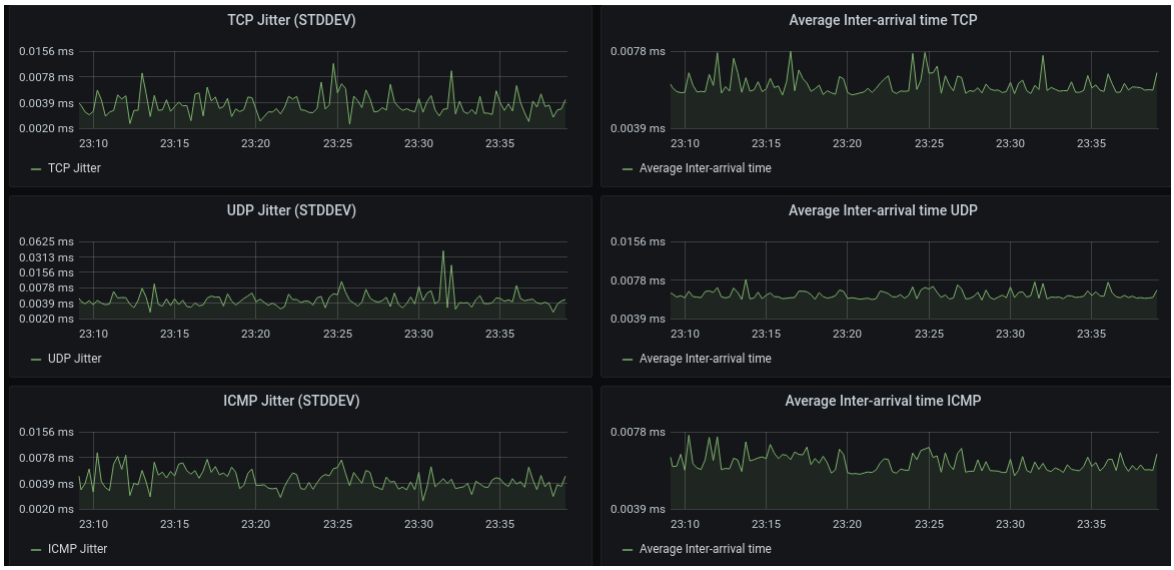


Figura 8: Vizualizare jitter Grafana

Asemenea throughput-ului, și în cazul jitter-ului, sunt afișate grafice cu deviația standard și media întârzierilor pentru fiecare protocol.

9 TESTARE

Testarea tool-ului se rezumă la testarea corectitudinii realizării măsurătorilor de performanță pe care le realizează programul de monitorizare. În acest sens am creat câte un script de testare pentru fiecare dintre cele 3 funcționalități de măsurători realizate. Pașii urmați în script-urile de testare sunt, în principal:

- generare trafic
- pornire program de monitorizare
- pornire tool-uri existente pentru măsurare în paralel a rezultatelor
- comparare rezultate

Înainte de a genera trafic pentru analizare, am creat un namespace "test" și o pereche de interfețe virtuale, veth0 și veth1, pentru a izola traficul, fără influențe exterioare.

Pentru generarea traficului am folosit tool-uri deja existente, precum *iperf3* și *trafgen*, iar pentru măsurători sau monitorizare am folosit *bmon* și *tcpdump*.

9.1 Testare throughput

```
configs = [
    {
        "RULES": [
            "- - - - [TCP,UDP]"
        ],
        "PATTERN": "0xFFFFFFFF",
        "OFFSET": -1,
        "SEQUENCE_THRESHOLD": 2
    },
    {
        "RULES": [
            "192.168.51.2 - 4500 - -"
        ],
        "PATTERN": "0xFFFFFFFF",
        "OFFSET": -1,
        "SEQUENCE_THRESHOLD": 2
    }
    [...]
]
```

Listing 9.1: Listă configurații pentru testare

Pentru testarea throughput-ului am creat o listă care conține mai multe tipuri de configurații pentru a măsura rezultatele pe mai multe seturi de reguli. Fiecare configurație se scrie într-un fișier JSON și este dat ca parametru programului la lansare.

Utilitarul *bmon* este pornit în paralel cu scriptul de monitorizare, apoi se generează trafic folosind *trafgen*. Output-urile celor două sunt redirecționate prin două pipe-uri separate pentru ca rezultatele să poată fi preluate în script-ul de testare în felul următor:

```
# Create communication pipes
try:
    os.mkfifo(monitor_pipe)
    os.mkfifo(bmon_pipe)
except OSError as oe:
    if oe.errno != errno.EEXIST:
        raise

# start bmon and MONITOR script in parallel
proc_bmon = subprocess.Popen("bmon -o ascii -p veth1 > bmon_pipe &",
                             shell=True)
proc_monitor = subprocess.Popen("python3 monitor.py -i veth1 " +
                             "-test_throughput > monitor_pipe &",
                             shell=True)

# start TRAFGEN
proc_trafgen = subprocess.Popen("ip netns exec test trafgen " +
                             "--dev veth0 --rate 600Mbit " +
                             "--conf config.cfg -q > /dev/null",
                             shell=True)

# open pipes
open_monitor_pipe = open(monitor_pipe, 'r')
open_bmon_pipe = open(bmon_pipe, 'r')
```

Listing 9.2: Creare procese pentru testare throughput

Un exemplu de fișier de configurație a pachetelor folosit pentru trafgen este:

```
# tcp packet
{   #ethernet header
    eth(da=00:1b:21:3c:9d:f8, da=90:e2:ba:0a:56:b4)
    #ip header
    ipv4(id=drnd(), mf, ttl=64, sa=10.1.0.8, da=10.1.0.9)
    tcp(sp=6000, seq=dinc(0,1000,1))        # tcp header
```

```
    'A', fill(0x41, 11)                         # payload
}
# udp packet
{   #ethernet header
    eth(da=00:1b:21:3c:9d:f8, da=90:e2:ba:0a:56:b4)
    #ip header
    ipv4(id=drnd(), mf, ttl=64, sa=10.1.0.7, da=10.1.0.9)
    udp(sp=4500)                                # udp header
    # payload
    0xA5, 0x30, 0x0F, 0x10,
    0xE7, 0xA5, 0xC3, 0x18,                     # pattern
    0x00, 0x00, 0x00, dinc(0, 1000, 4),         # seq number
    0xA5, 0x30, 0x0F, 0x10                      # rest of payload
}
# icmp packet
{   #ethernet header
    eth(da=00:1b:21:3c:9d:f8, da=90:e2:ba:0a:56:b4)
    #ip header
    ipv4(id=drnd(), mf, ttl=64, sa=10.1.0.6, da=10.1.0.9)
    icmpv4(echorequest)                         # icmp header
    # payload
    0xA5, 0x30, 0x0F, 0x10,
    0xE7, 0xA5, 0xC3, 0x18,                     # pattern
    0x00, 0x00, 0x00, dinc(0, 1000, 2),         # seq number
    0xA5, 0x30, 0x0F, 0x10                      # rest of payload
}
```

Listing 9.3: Fișier de configurare pentru trafgen pentru testarea throughput-ului

```
while True:
    line_bmon = open_bmon_pipe.readline().split()
    bmon_throughput = line_bmon[1][:-3]
    for j in range(5):
        line_monitor = open_monitor_pipe.readline().split()
        if len(line_monitor) > 1:
            if j > 1:
                if j == 2:
                    tcp_throughput = float(line_monitor[3])
                elif j == 3:
                    udp_throughput = float(line_monitor[3])
                else:
                    icmp_throughput = float(line_monitor[3])

    monitor_throughput += tcp_throughput + udp_throughput + \
                          icmp_throughput
    if monitor_throughput >= 0.95*bmon_throughput and \
        monitor_throughput <= 1.05*bmon_throughput:
        print('CORRECT MEASUREMENT!!!')
```

```
20      else:
21          print('WRONG MEASUREMENT!!!')
```

Listing 9.4: Parsare output bmon și script de monitorizare

După parsarea output-urilor celor două, bmon și scriptul creat de monitorizare, se compară rezultatele cu o anumită toleranță.

În cazul tuturor configurațiilor de test, rezultatele tool-ului creat și cele ale bmon-ului au fost în concordanță.

9.2 Testare jitter

Scriptul de test corespunzător jitter-ului se bazează pe utilitarul *tcpdump* care capturează pachetele venite pe rețea și redă timestampul capturării fiecăruia.

```
1  try:
2      os.mkfifo(tcpdump_pipe)
3      os.mkfifo(monitor_pipe)
4  except OSError as oe:
5      pass
6
7  # start monitor
8  proc_monitor = subprocess.Popen("python3 monitor.py -i veth1" + \
9                                  " -test_jitter > monitor_pipe &",
10                                 shell=True)
11
12 # start tcpdump
13 proc_tcpdump = subprocess.Popen("tcpdump -i veth1 " + \
14                                 "--immediate-mode -l -n " + \
15                                 "--time-stamp-precision=nano " +\
16                                 "-ttt > tcpdump_pipe &", shell=True)
17
18 open_monitor_pipe = open(monitor_pipe, 'r')
19 open_tcpdump_pipe = open(tcpdump_pipe, 'r')
```

Listing 9.5: Creare procese pentru testare jitter

La fel ca în cazul throughput-ului, am folosit pipe-uri pentru redirectarea output-ului în vederea prelucrării.

Acest script se bazează pe captura tcpdump-ului care este pornit în modul "immediate"

pentru a afișa pachetele la momentul sosirii lor din rețea, fără a fi salvate în fișiere separate pe motive de eficiență.

```
test_number = 0
trafgen_gaps = [10, 20, 50, 100, 500]

os.chdir("../")
print("Starting trafgen test {0} with {1} ms gap between packets"\
      .format(test_number, trafgen_gaps[test_number]))
time.sleep(1)

proc_trafgen = subprocess.Popen("ip netns exec test trafgen " + \
                                "--dev veth0 " + \
                                "--conf config_jitter.cfg " +\
                                "-t {}ms > /dev/null"
                                .format(trafgen_gaps[test_number]),
                                shell=True)
```

Listing 9.6: Introducere pauze între pachete

Am creat 5 teste pentru validarea jitter-ului, fiecare corespunzător unei anumite întârzieri între pachete de ordinul milisecundelor. Am reușit să introduc aceste întârzieri prin intermediul unei opțiuni valabile pentru trafgen (-t). Pentru fișierul de configurație dat generatorului de trafic am ales pachete simple TCP, protocolul neavând un rol important în cazul jitter-ului:

```
# tcp packet
{
  # ethernet header
  eth(da=00:1b:21:3c:9d:f8, da=90:e2:ba:0a:56:b4)
  # ip header
  ipv4(id=drnd(), mf, ttl=64, sa=192.168.51.2, da=10.1.0.2)
  tcp(sp=6000)                # tcp header
  'A', fill(0x41, 11),        # payload
}
```

Listing 9.7: Fișier de configurare pentru trafgen pentru testarea jitter-ului

Pentru a putea verifica valorile jitter-ului, am aplicat funcțiile de medie și deviație standard valabile prin biblioteca *statistics* din Python pe vectorul de timestampuri reținute pe baza tcpdump-ului.

```
nanos = []
old_time = time.time()
while True:
    line_tcpdump = open_tcpdump_pipe.readline()
    line_monitor = open_monitor_pipe.readlines()
    if len(line_tcpdump) > 5:
        data_tcpdump = line_tcpdump.split()[0]
        ns = int(str_to_ns(data_tcpdump))
        [...]
        nanos.append(ns)
    for line in line_monitor:
        data_monitor = line.split()
        [...]
        values_monitor["Average"] = float(data_monitor[2])
        [...]
        values_monitor["Jitter"] = float(data_monitor[2])
    current_time = time.time()
    if current_time - old_time >= 1:
        [...]
        values_tcpdump["Average"] = round(statistics.mean(nanos) /
    10**6, 4)
        values_tcpdump["Jitter"] = round(statistics.stdev(nanos) /
    10**6, 4)
        print(tabulate([
            ['Avg', values_tcpdump["Average"],
            values_monitor["Average"], "ms"],
            ['Jitter', values_tcpdump["Jitter"],
            values_monitor["Jitter"], "ms"]],
            headers=[' ','TCPDUMP', 'MONITOR.PY', "UNIT"]))
        [...]
```

Listing 9.8: Parsare output tcpdump și script de monitorizare

9.3 Testare erori de secvență

În lipsa unui tool existent care să realizeze aceste contorizări, pentru testarea erorilor de secvență am creat un script de testare bazat pe observarea comportamentului măsurătorilor în funcție de traficul generat și comportamentul așteptat. De exemplu, am generat pachete cu numere de secvență care merg din 3 în 3 și se resetează la un anumit număr pentru a verifica corectitudinea modificării erorilor mari și a celor de inversare. De asemenea, un alt exemplu este generarea de pachete UDP cu număr de secvență inserat după un anumit pattern în payload pentru a valida buna funcționare a pattern matching-ului și aflarea numărului de secvență introdus.

10 CONCLUZII ȘI FUNCȚIONALITĂȚI POSIBILE ULTERIOARE

Pe baza tuturor testelor prezentate în capitolul Testare realizate asupra tool-ului și a celorlalte aspecte descrise pe larg pe parcursul lucrării, se poate concluziona asupra faptului că proiectul dezvoltat îndeplinește toate obiectivele propuse inițial, și anume:

- rapiditate: asigurată prin utilizarea BPF-ului
- ușurință în utilizare
- măsurători în timp real
- afișarea rezultatelor într-un mod intuitiv (grafice)
- ușurința deployment-ului

Deși complex și într-un stadiu pregătit de utilizare, proiectul poate fi îmbunătățit și alte funcționalități utile îi pot fi aduse. Câteva dintre acestea sunt:

- măsurarea latenței
- interfață grafică (alternativă la editare fișier de configurație și utilizare CLI)
- posibilitate modificare sau aruncare pachete

În concluzie, prin intermediul acestei teze am prezentat utilitatea și capabilitățile tool-ului creat. Acesta este un tool de monitorizare și analizare al traficului care poate fi folosit pentru partea de testare a unei rețele în dezvoltare.

BIBLIOGRAFIE

[1] Jason Lackey. Virtual taps - the abcs of network visibility. https://blogs.keysight.com/blogs/tech/nwvs.entry.html/2020/04/16/virtual_taps_-_thea-3sSE.html. Last accessed: 8 June 2021.

[2] Adrian Ratiu. An ebpf overview. https://www.collabora.com/news-and-blog/blog/2019/04/05/an-ebpf-overview-part-1-introduction/. Last accessed: 29 March 2021.

[3] Joab Jackson. Linux technology for the new year: ebpf. https://thenewstack.io/linux-technology-for-the-new-year-ebpf/. Last accessed: 29 March 2021.

[4] Lucas Severo Alves. The top reasons why you should give ebpf a chance. https://blog.container-solutions.com/the-top-reasons-why-you-should-give-ebpf-a-chance. Last accessed: 29 March 2021.

[5] Ryan Tendonge. What is prometheus? https://www.metricfire.com/blog/what-is-prometheus/, . Last accessed: 14 June 2021.

[6] Ryan Tendonge. What is grafana? https://www.metricfire.com/blog/what-is-grafana/, . Last accessed: 15 June 2021.

[7] Ryan Tendonge. What is docker monitoring? https://www.metricfire.com/blog/what-is-docker-monitoring/, . Last accessed: 15 June 2021.