Università di Pisa
Dipartimento di Informatica

UNIVERSITÀ DI PISA

Disegno ed implementazione di un IDS basato su DPI

Tesi di Tirocinio

Candidato: Matteo Biscosi
Supervisore: Luca Deri

A.A. 2019-20

Indice

1 Introduzione

Il progetto di studio consiste nello sviluppo di un Network-based IDS (Network-based Intrusion Detection System), ossia di un software che attraverso il monitoraggio del traffico di rete cerca di individuare la presenza di attività sospette. Tale software a differenza dei vari IDS Open Source disponibili sul mercato, si basa sulla tecnica di DPI (Deep Packet Inspection), che viene utilizzata per l'analisi dei contenuti del traffico di rete fino al livello 7 del modello ISO OSI (modello standard per la rappresentazione della struttura dei pacchetti del traffico di rete).

La finalità del software sviluppato è quella di migliorare gli attuali IDS col supporto di DPI in quanto esso aiuta a superare le limitazioni di questi ultimi, permettendo estrazione di maggiori metadati dai flussi (collezione di traffico di rete con caratteristiche specifiche in comune). Questo è reso possibile grazie alla ampia quantità di protocolli supportati da DPI. Inoltre essa aiuta ad incrementare la difesa del sistema in quanto, per esempio, un normale IDS studiando solamente una parte dei contenuti del traffico (fino al livello 4 del modello ISO OSI) non riesce a riconoscere il traffico di rete che circola su porte non standard.

Tuttavia l'implementazione di questo software potrebbe causare problemi di prestazioni in quanto dovendo analizzare tutto il traffico di rete, il carico di lavoro aumenta fortemente; per contrastare tale rischio ho utilizzato l'accelerazione al processamento dei dati e dei

flussi fornita da una scheda di rete Napatech, la quale fornisce hardware offloading che consiste nella possibilità di eseguire operazioni utilizzando direttamente l'hardware al posto di un processo o di una funzione/metodo software.

Il progetto è divisibile in due fasi: nella prima fase ho sviluppato un software IDS funzionante su tutti i dispositivi Linux indipendentemente dalla scheda di rete, in quanto ho utilizzato una libreria apposita (libpcap) per la cattura dei pacchetti. Nella seconda fase ho migliorato il software sviluppato, utilizzando il supporto Hardware fornitomi dalla scheda di rete Napatech, principalmente per la cattura dei pacchetti e per la gestione dei flussi.

1.1 Struttura della tesi

Nel capitolo 2 verrà trattato l'obiettivo del progetto ed i vari argomenti affrontati, terminando con un approfondimento del problema e delle lacune degli attuali sistemi di IDS.
Successivamente verrano descritte le varie librerie e compenenti hardware utilizzate specificando il perchè di tali scelte; seguirà un breve focus sull'implementazione specifica dell'architettura del sistema.
Infine concluderò l'elaborato con i capitoli dedicati alla fase di testing e risultati ottenuti da essi, alle possibili migliorie apportabili in futuro e alle conclusioni del progetto.

2 Obiettivi e Stato dell'Arte

Come già introdotto precedentemente, l'obiettivo del progetto è quello di sviluppare un IDS (Intrusion detection system) ossia un software che monitora il traffico di rete cercando di individuare possibili intrusioni o attività sospette nel sistema e basandolo sulla tecnica della DPI (Deep Packet Inspection). Tale tecnica si fonda sul concetto di analizzare in modo approfondito i pacchetti che circolano nella rete al fine di ottenere informazioni maggiori e più accurate sul traffico che attraversa il sistema.
Questo obiettivo è dettato dal fatto che gli attuali sistemi di IDS utilizzanti DPI, tra i più conosciuti ed efficienti abbiamo snort, zeek e suricata (tali sistemi verranno presentati in seguito in questo capitolo), supportano solamente pochi Gigabit al secondo di pacchetti [48] (considerando una media di 65 bytes a pacchetto, 1 Gbps si tratta all'incirca di 1,5 milioni di pacchetti al secondo), ad esempio nel caso di snort e suricata non riescono a superare i 2 Gbps senza iniziare a perdere pacchetti [48].
Ciò comporta che in caso di alta affluenza di pacchetti questi software potrebbero diventare dei colli di bottiglia del sistema con il rischio di rallentamento di tutta la rete, senza contare la non affidabilità sulla sicurezza del sistema in quanto, in seguito alla perdita di pacchetti e quindi delle relative informazioni che essi trasportavano, potrebbero fornire resoconti non accurati e, come sottolineato anche da Axellson [27], spesso anche errati.

Tuttavia queste lacune di efficienza sono dovute a vari problemi

che questi tipi di Intrusion Detection Systems si portano con sè: l'efficienza della DPI e la gestione delle informazioni dei pacchetti.

Per quanto riguarda la Deep Packet Inspection si riscontrano due problemi fondamentali. Il primo è dovuto all'analisi delle informazioni portate dal pacchetto in quanto richiede un'analisi completa delle informazioni (a differenza dei normali sistemi di intrusione che ne richiedono soltanto un'analisi parziale). Il secondo è invece dovuto alla complessità dell'algoritmo di ricerca utilizzato per individuare il tipo di informazioni trasportate dai pacchetti: più aumenta lo spazio in cui cercare il tipo di informazioni contenute più l'algoritmo richiederà tempo.

Il problema che riguarda la gestione dei pacchetti è invece dovuto al fatto di capire se un pacchetto arrivato, contiene informazioni appartenenti ad un nuovo flusso o meno e nel caso aggiungerle al sistema. Per flusso, come descritto nell'rfc 7011 [15], si intende una sequenza di pacchetti che condividono una serie di informazioni specifiche uguali (per esempio da chi è stato inviato e chi l'ha ricevuto). Nel caso queste siano di un flusso già esistente, verranno aggiornate le informazioni presenti nel sistema con quelle nuove, mentre nel caso in cui siano di un nuovo flusso dovranno essere aggiunte nel sistema.
Inoltre, per mantenere le informazioni consistenti bisogna controllare continuamente se questi flussi di informazioni sono sempre attivi (per cui continuano ad arrivare pacchetti appartenenti al flusso) oppure se sono terminati (se è arrivata una specifica richiesta di terminazione della comunicazione) o se sono diventati inattivi (ossia

non si hanno più informazioni su quel flusso da un certo periodo di tempo) ed in questi due ultimi casi bisogna provvedere ad eliminare le informazioni per evitare un sovraccarico del sistema.

Per ovviare a tali problemi di efficienza ho deciso di utilizzare una scheda di rete (componente che consente la connessione del dispositivo ad una rete internet e la conseguente trasmissione e ricezione di dati [51]) che mi fornisca la cosiddetta accelerazione hardware, ossia la possibilità di eseguire funzioni utilizzando direttamente l'hardware invece che di un processo o di una funziona software, permettendomi così di migliorare le prestazioni in modo esponenziale.

Per cui l'obiettivo del progetto è quello di colmare queste lacune di prestazioni, riuscendo così a migliorare i sistemi di IDS iniziali [27], rendendoli sistemi ad alta precisione ed efficacia [11, 41] senza che siano un collo di bottiglia del sistema.
Per raggiungere questo obiettivo il risultato ottimale sarebbe quello di ottenere un IDS che riesca a gestire anche reti con 100 Gbps di connessione, tuttavia quello atteso (per i problemi di prestazioni discussi finora) è di un software che riesca ad avere un buon risultato fino a 30 Gbps ed inizi a perdere una quantità ragionevole di pacchetti nel range che va dai 30 ai 50 Gbps, così da poter essere un buon software fino alle reti da 10 Gbps ed accettabile in quelle superiori.
Per verificare il raggiungimento dell'obiettivo a fine elaborato confronterò i risultati ottenuti con i risultati attesi appena descritti oltre a confrontare il software sviluppato con altri software IDS con e senza hardware offloading (accelerazione hardware) in modo da

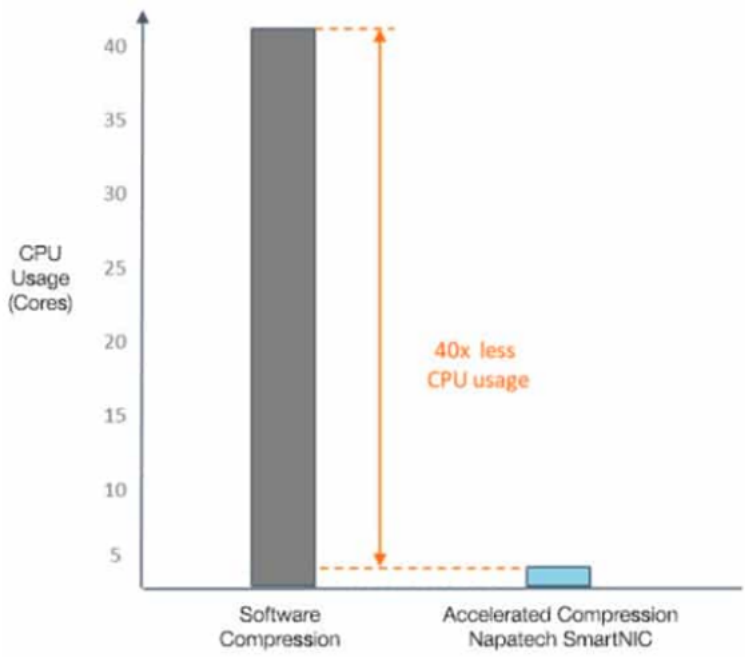
poter paragonare l'efficienza con i prodotti distribuiti sul mercato quali Cento, Zeek ed altri due software, FIXIDS e General IDS [63, 64], che utilizzano hardware offloading.

Nel seguito del capitolo introdurrò in maniera approfondita e tecnica gli argomenti trattati in questo software, per la precisione in ordine: breve introduzione all'hardware offloading ed ai flussi, Intrusion Detection System, Deep Packet Inspection, Sistemi IPS/IDS con supporto DPI ed infine farò una breve presentazione dei migliori software presenti sul mercato che svolgono questo stesso lavoro e delle prestazioni che hanno.

2.1 Hardware Offloading

Per hardware offloading si intende la possibilità di eseguire funzioni utilizzando direttamente l'hardware invece che di un processo o di una funziona software[46].
Il beneficio di utilizzare l'offloading, in genere, è l'aumento delle performance e throughput senza dipendere dalle decisioni della CPU [43, 45, 52].
Come descritto nell'articolo scritto da Napatech[12], l'hardware offloading permette ad una qualunque funzione di terze parti (Intellectual Property, IP) di essere aggiunto all'FPGA ("Field Programmable Gate Array" è un dispositivo logico programmabile ovvero genericamente un dispositivo hardware elettronico formato da un circuito integrato le cui funzionalità logiche di elaborazione sono appositamente programmabili [53]) della Napatech SmartNIC.

Per osservare il miglioramento di prestazioni fornito dalla scheda, sono stati fatti dei test di 40GB di compressione di files sviluppati da Nokia[13].



La figura mostra le performance ottenute facendo utilizzo di hardware offloading, come l'FPGAs

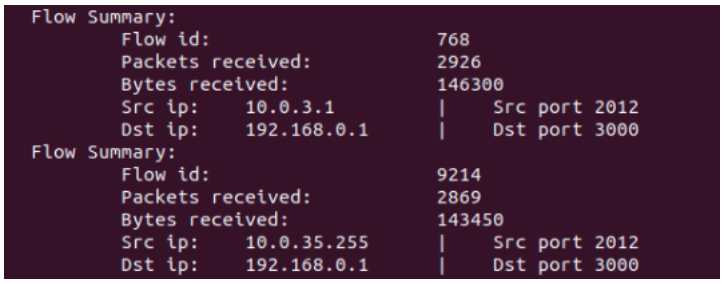
Infatti in questo particolare caso è stato possibile effettuare la compressione del file ad una velocità di 40 Gbps utilizzando un solo core della CPU per poter gestire l'offloading, fornendo quindi un miglioramento di circa 40 volte.

Uno dei principali esempi è l'IPv4 traffic forwarding. Senza l'offloading, il traffico IPv4 verrebbe ruotato attraverso la CPU rendendolo per cui limitato e aumentando anche il carico su di essa, cosa che altrimenti non avverrebbe in caso di offloading [47].

2.2 Flussi

Come definito nella versione standard di Cisco NetFlow[14, 15], un flusso è una sequenza unidirezionale (o bidirezionale) di pacchetti che condividono una serie di informazioni uguali, decise prima della creazione del flusso; queste informazioni in genere sono:

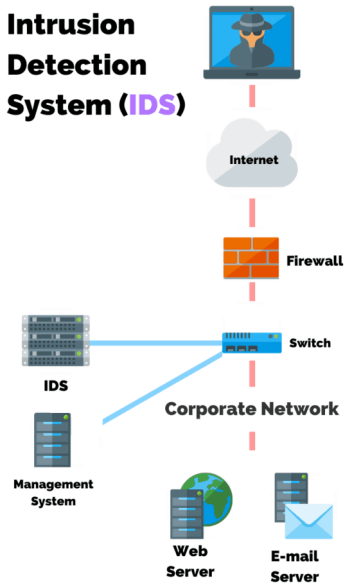
- Interfaccia di ingresso;
- Indirizzo IP sorgente;
- Indirizzo IP destinatario;
- Protocollo di livello IP;
- Porta sorgente per UDP o TCP, 0 per gli altri protocolli;
- Porta destinataria per UDP o TCP, 0 per gli altri protocolli;
- IP Type of Service;
- Protocollo di livello Trasporto (più raramente);



Esempio di flusso bidirezionale basato su indirizzo IP e porta sorgente e destinatario

2.3 Intrusion Detection System (IDS)

Un IDS è un software che monitora il traffico di rete cercando di individuare qualunque attività sospetta. Una volta che le minacce sono state individuate, l'IDS genera notifiche di allarme per avvertire del problema. Le ultime versioni di IDS tendono anche ad analizzare ed identificare patterns tipici di attacchi informatici in quanto hanno probabilità minore di generare falsi allarmi [31].



Struttura tradizionale di un IDS[7]

Come specificato nell'articolo di B. Santos Kumar[1], esistono tre tipi di Intrusione Detection System[26]: Network-Based Intrusion Detection System (NIDS), Host-Based Intrusion Detection System (HIDS), Wireless local area network IDS (WIDS).
In più possiamo aggiungerci anche un quarto tipo, l'Hybrid Intrusion Detection System.

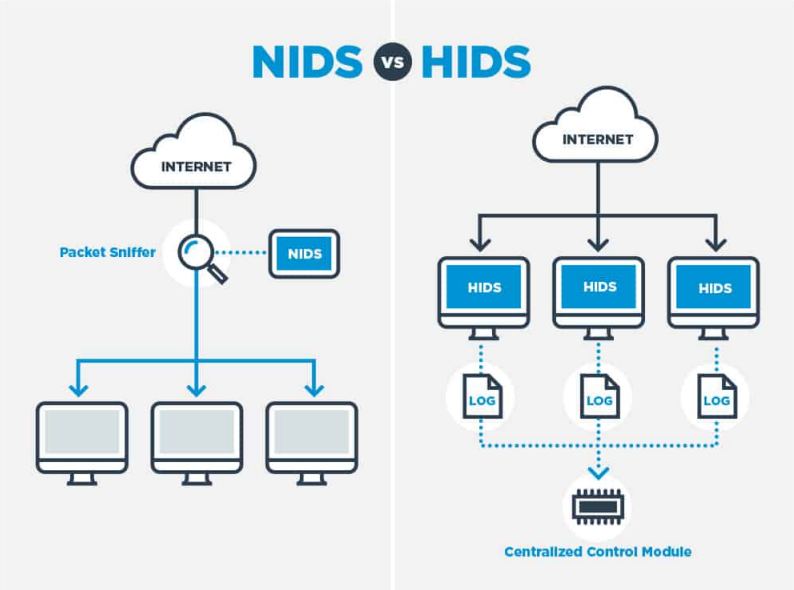
Network-Based Intrusion Detection System (NIDS)

I Network-Based Intrusion Detection System (NIDS) [28] sono dediti ad analizzare il traffico di uno o più segmenti di una LAN al fine di individuare anomalie nei flussi o probabili intrusioni informatiche [58]. I più comuni NIDS sono composti da una o più sonde (sensori) dislocate sulla rete le quali comunicano con un server (motore) che si appoggia ad un database. Fra le attività anomale che possono presentarsi e venire rilevate da un NIDS vi sono: accessi non autorizzati, propagazione di software malevolo, acquisizione abusiva di privilegi appartenenti a soggetti autorizzati, intercettazione del traffico (sniffing), negazioni di servizio (DDoS) [56, 57]. Un NIDS analizza il traffico in entrata ed in uscita che il Firewall non blocca classificandolo come "affidabile" (es. impostazione di regole non troppo restrittive sui pacchetti); inoltre sopperisce, se configurato opportunamente, ad errori di configurazione del Firewall stesso, creando file di log da cui il sistemista può trarre delle indicazioni utili alla configurazione.

Host-based intrusion detection systems (HIDS)

Gli Host-based intrusion detection systems (HIDS) sono applicazioni che operano su informazioni collezionate dagli individuali sistemi, fornendo il vantaggio di analizzare attività sull'host che monitora ad un alto livello di dettaglio, potendo spesso determinare quale processo e/o utente è coinvolto in attività malevoli [32, 33].

Inoltre gli HIDS sono a conoscenza del risultato di un tentativo di attacco in quanto possono accedere direttamente e monitorare i files ed i processi colpiti da essi.



Struttura a confronto di NIDS e HIDS[7]

In base al tipo di dati esaminati, gli HIDS sono divisibili in due categorie:

- HIDS Based Application: sono gli IDS che ricevono informazioni dalle applicazioni (per esempio i logs generati dal server web o dal firewall);
- HIDS Based Host: sono gli IDS che ricevono informazioni dalle attività di sistema;

Gli HIDS devono essere installati su tutte le macchine e richiedono configurazioni specifiche per quel sistema operativo e software.

Benefit	Host	Network
Deterrence	Strong deterrence for insiders.	Strong deterrence for outsiders.
Detection	Strong insider detection. Weak outsider detection.	Strong outsider detection. Weak insider detection.
Response	Weak real-time response. Good for long-term attacks.	Strong response against outsider attacks.
Damage Assessment	Excellent for determining extent of compromise.	Very weak damage assessment capabilities.
Attack Anticipation	Good at trending and detecting suspicious behavior patterns.	None.
Prosecution Support	Strong prosecution support capabilities.	Very weak because there is no data source integrity.

Confronto tra host-based e network-based IDS [34]

Wireless local area network IDS (WIDS)

Il Wireless local area network IDS (WIDS) è simile al NIDS, tuttavia analizza anche traffico wireless. La tecnologia wireless è una tecnologia relativamente nuova, per cui sono stati sviluppati fino ad ora solo pochi WIDS [1].

Hybrid Intrusion Detection System

L'Hybrid Intrusion Detection System è un IDS generato dalla combinazione di due o più approcci sopra descritti. Generalmente in un hybrid IDS i dati di sistema sono combinati con informazioni sulla rete per sviluppare una visione completa del sistema [35, 56] (gli hybrid IDS sono più efficaci rispetto agli altri IDS).

Gli IDS utilizzano in genere tre metodologie per individuare intrusioni nel sistema (la maggior parte di IDS utilizzano una combinazione delle prime due tecniche[35]): signature-based, anomaly-based e stateful protocol.

Signature-Based detection

L'approccio signature-based si incentra sull'identificare una "firma" di un'intrusione [31] (per esempio un pattern conosciuto di intrusione). Affinchè questa tecnica funzioni, c'è bisogno che sia aggiornata regolarmente così da poter riconoscere le firme comuni in quanto esse sono sempre in evoluzione [1]. In altre parole se l'attaccante cambia dettagli di un attacco conosciuto, potrebbe evadere il signature-based IDS, così come se utilizza pattern non presenti nel database dell'IDS.

Anomaly-Based detection

L'approccio anomaly-based si incentra sull'identificare pattern inusuali o inaspettati delle attività [36] (questo metodo va a compensare per tutti gli attacchi che riescono a passare il metodo signature-based). Tuttavia alcuni comportamenti validi fino ad ora sconosciuti potrebbero erroneamente essere flaggati come pericolosi [1]. Gli anomaly-based IDS sono ottimi nel fornire indicazioni di attacchi imminenti (un esempio di anomalia è il fallimento di multipli tentativi di login su una porta non usuale).

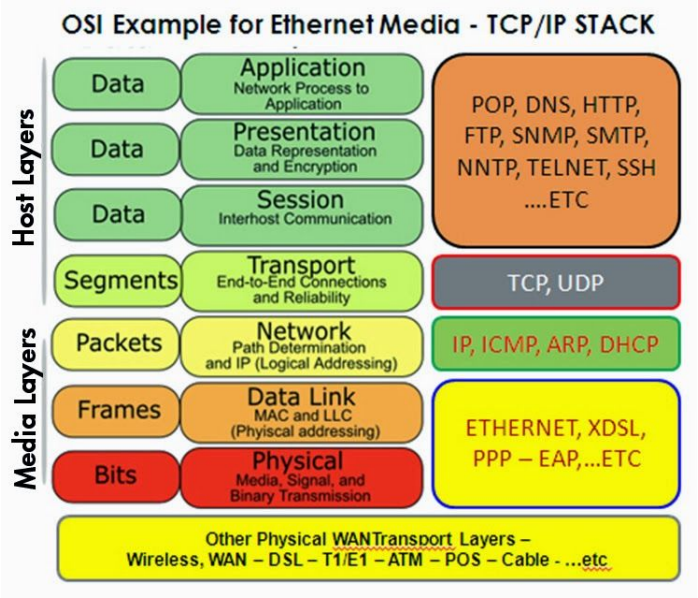
Stateful Protocol Inspection

Stateful Protocol Inspection è molto simile all'appoccio anomaly-based ma, a differenza di quest'ultimo, può anche analizzare il traf-

fico a livello rete e trasporto ed alcuni traffici specifici a livello applicativo [1].

2.4 Deep Packet Inspection (DPI)

La Deep Packet Inspection è un metodo utilizzato per analizzare i contenuti del traffico di rete, permettendo di filtrare pacchetti basati sull'analisi in profondità di tutti i livelli del modello ISO OSI [37]. Tipicamente, infatti, l'analisi si ferma solamente all'header del livello IP del pacchetto, per esempio nel caso di un firewall stateless, conosciuto anche come ACL (Access Control List)[8], controllerebbe le connessioni basandosi unicamente su indirizzo IP sorgente e destinatario, così come un firewall stateful [37] si fermerebbe a guardare fino al livello di trasporto. La DPI invece apre il pacchetto e guarda dal livello 2 al livello 7 del modello ISO OSI (Payload del messaggio compreso).



Modello ISO OSI

Esistono due tipi di analisi dei pacchetti basata su DPI[9]: pattern matching ed event-based analysis.

Pattern Matching

E' un metodo DPI che prevede la ricerca all'interno della rete per sequenze conosciute di bytes o di espressioni regolari [10] (La ricerca può essere anche limitata a specifiche parti del pacchetto). La relativa semplicità di questo approccio è un motivo per il quale è uno dei metodi più popolari dei DPI, tuttavia questa semplicità diventa un problema quando non è possibile descrivere i patterns con espressioni regolari semplici (per esempio nel caso in cui bisogni cercare se un certificato specifico appartiene ad una lista, potrebbe diventare una espressione regolare molto complicata).

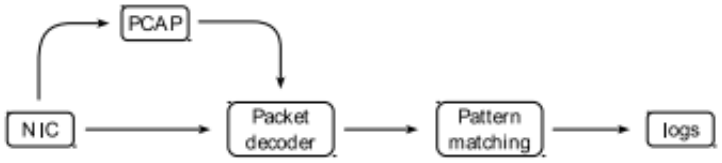


Figure. 7. DPI – Pattern matching method.

[10]

Oggi le reti che utilizzano il metodo di pattern matching, in genere, decodificano i protocolli più utilizzati.

Event-based Analysis

In questa tecnica i pacchetti sono processati in eventi e successivamente processati a turno da scripts [10] (i quali possono implementare complessi algoritmi di processamento e aggiungere nuove funzionalità legate a DPI).
Questa architettura rimpiazza la parte di pattern matching con algoritmi implementati con programmi.

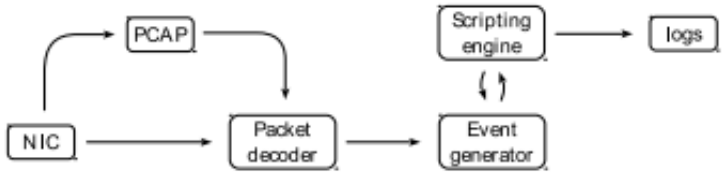


Figure. 8. DPI with event-based analysis.

[10]

Gli algoritmi possono essere stateless, ossia reazioni immediate o catene di reazioni a specifici eventi, o stateful, ossia algoritmi che

possono utilizzare vari programmi per ricordarsi lo stato tra eventi.

DPI è per cui molto importante in quanto può migliorare la sicurezza del traffico di una rete anche se non tutti i router possono utilizzare la deep packet inspection, in quanto, se non configurata correttamente, potrebbe essere un collo di bottiglia della rete [38]. Un router che utilizza DPI dovrebbe così essere abbastanza potente da poter aprire ogni pacchetto, ispiezionarlo e poi inviarlo nuovamente.

2.5 Sistemi IPS/IDS che utilizzano DPI

Un IDS è capace di trovare le intrusioni ma non di bloccare l'attacco, per cui può utilizzare DPI per [8, 39]:

- Collezionare più informazioni sull'attacco;
- Indentificare firme e pattern di alcuni attacchi;
- Controllare il traffico della rete, per esempio Telnet, FTP root access o contenuti specifici HTTP;

Con questa tecnica è possibile velocizzare gli IDS permettendogli di identificare gli attacchi più rapidamente e persino controllarli.

Tuttavia quando si utilizza DPI per scovare intrusioni, ci sono diverse sfide a cui si va incontro[11]:

- La complessità dell'algoritmo di ricerca: la complessità degli algoritmi e delle operazioni di paragone per rilevare il tipo di firma, diminuiscono il throughput del sistema;
- L'aumento delle firme di intrusione: l'ampio numero di firme rende il lavoro dell'IDS molto più difficile;
- La sovrapposizione delle firme: in genere gli attacchi possono essere categorizzati in gruppi in base alle proprietà in comune (per esempio il tipo di protocollo utilizzato). Per cui è necessario processare il pacchetto prima di effettuare l'operazione di matching di una firma;
- Locazione della firma sconosciuta: a causa della vasta varietà di tipi di attacchi differenti, la firma di un attacco non è localizzata in un posto specifico del pacchetto, questo vuol dire che l'IDS deve ispezionare tutto il payload per poterla trovare;

A causa di questi problemi, in genere, per essere utilizzato da un IDS il protocollo DPI deve soddisfare dei requisiti, quali [11, 40]:

- Performance deterministica: l'architettura deve operare e processare il traffico indipendentemente dalle caratteristiche delle firme o del traffico (quindi il sistema deve essere in grado di gestire il traffico nel caso peggiore di software ed hardware);
- Memory efficiency: il tempo di accesso in memoria è uno dei principali colli di bottiglia di un sistema DPI;
- Aggiornamenti dinamici: fornire la possibilità di aggiungere e rimuovere firme al sistema senza poter affliggere le operazioni di sistema;

Come vediamo in alcune classifiche[2], tra i software IDS che utilizzano DPI più conosciuti ed usati troviamo Zeek(Bro), Snort e Suricata.

Zeek (Bro)

Zeek[3] è un analizzatore passivo del traffico di rete open source. Utilizzato principalmente come sistema di monitoraggio per ispezionare tutto il traffico su un link al fine di individuare attività sospette. Fornisce un set molto ampio di files di log che registrano le attività di rete; questi log non includono solamente records di ogni connessione vista ma anche trascrizioni di livello applicativo, per esempio richieste DNS con relative risposte oppure tutte le sessioni HTTP con i corrispettivi URIs richiesti, headers, MIME types e risposte del server.
Inoltre Zeek fornisce un linguaggio Turing-completo così da poter effettuare qualunque attività di analisi l'utente voglia.

Snort

Snort[4] è un software per il rilevamento di intrusioni in rete (NIDS) open source basato sulla libreria libpcap. Si tratta di un software molto efficace, con capacità di eseguire in tempo reale l'analisi del traffico e un packet logging (utile per debuggare il traffico della rete) su IP networks. Esso può compiere analisi dei protocolli, può individuare una varietà di attacchi e probes, così come buffer overflow, port scanning, attacchi CGI e probes SMB. Guardando i file di

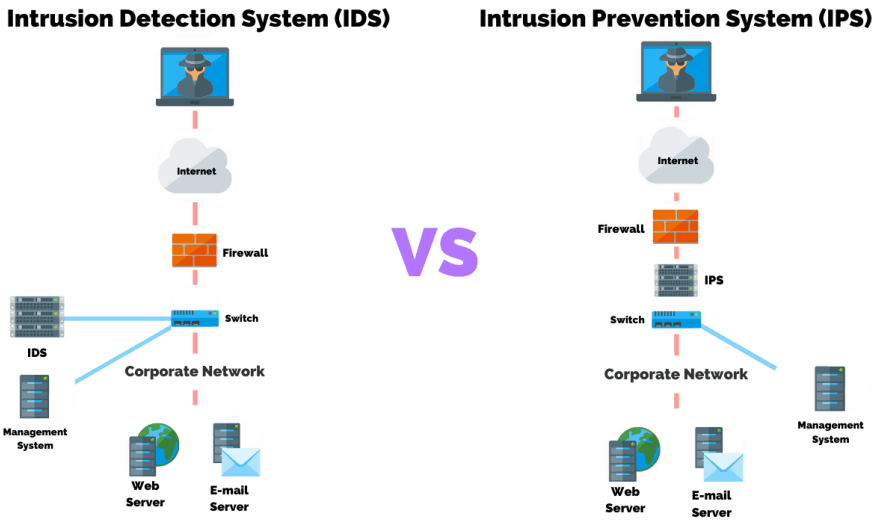
log si riesce a determinare a quale tipo di attacco si è stati soggetti. Un uso efficace di meccanismi di logging è fondamentale per la sicurezza del sistema. E' possibile usare programmi di filtro dedicati, che periodicamente effettuano uno scanning sui principali file di log di sistema, verificando eventuali "anomalie" come parecchie connessioni dallo stesso indirizzo IP in un brevissimo intervallo di tempo. Si possono anche scrivere programmi di logging dedicati per difendersi. Snort può lavorare in diversi modi:

- Packet Sniffer: è capace di ispezionare il carico dei pacchetti sulla rete, decodificando il livello di applicazione di un pacchetto e catalogando il traffico basato su un certo contenuto di dati.
- Packet Logger: può anche effettuare il log dei pacchetti su linea di comando indirizzati ad una specifica locazione, in un syslog, ed invia alert a video.
- Intrusion Detection: può essere usato come IDS su reti dove sono richieste alte prestazioni. Ha un piccolo sistema di firme ed è disegnato per essere un tool veloce di alerting per gli amministratori quando sono individuate attività sospette.

Suricata

Suricata[5] è un sistema di rilevamento di minacce open source che fornisce sia capacità di intrusion detection (IDS) che di intrusion prevention (IPS, a differenza dell'IDS, è posizionato inline, ossia direttamente sul percorso di comunicazione che c'è tra sorgente e destinatario ed analizza il traffico, eseguendo azioni sul traffico che

entra nella rete se ritenuto malevolo [54], per esempio terminare la connessione TCP o bloccare completamente un indirizzo ip dal poter accedere a qualunque applicazione).



Struttura di IDS ed IPS a confronto [7]

Molto simile a Snort, Suricata, differisce per [6, 55]:

- E' un software multi-threaded, per cui una singola istanza può performare per un alto volume di traffico;
- Fornisce maggiore supporto disponibile per protocolli di livello applicativo;
- Supporta hashing e file extraction;
- Programmato in Lua, fornisce la possibilità di modificare l'output e creare una logica di rilevamento molto complessa e dettagliata;

3 Architettura del Sistema

In questa sezione descriverò l'architettura del sistema, specificando quali librerie ed hardware ho utilizzato per sviluppare il progetto ed il perchè di tali scelte.

In breve il software è composto da una sezione che si occupa di sniffare pacchetti, da una che raccoglie i pacchetti ottenuti in flussi e sui quali viene fatta la Deep Packet Inspection; dopodichè, quando viene individuato il protocollo di livello applicativo, viene fatto un controllo del rischio di quel flusso che, se ritenuto pericoloso, viene salvato con le informazioni relative al flusso all'interno di un file di log apposito.

3.1 libpcap

Quando utilizziamo uno sniffer [16,18], il driver di rete manda i pacchetti ad una sezione del kernel chiamata packet filter, la quale invia i pacchetti allo sniffer o al software di monitoraggio (come mostrato in figura).

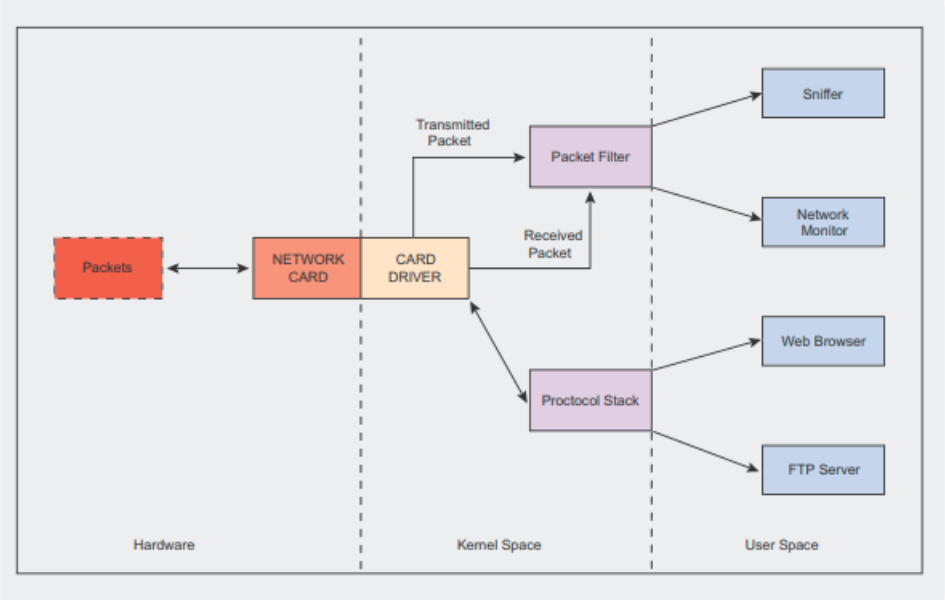


Figure 1. *Elements involved in the capture process*

[16]

Per effettuare la cattura e filtraggio di pacchetti ho utilizzato libpcap in quanto una delle migliori e più usate librerie Open Source in circolazione [19].
Come descritto nella documentazione ufficiale [17], libpcap è una libreria Open Source di cattura di pacchetti; disponibile per diverse piattaforme, permette non solo di catturare pacchetti su una determinata interfaccia di rete ma anche di creare e manipolare pacchetti da file salvati.

Tuttavia come vedremo più avanti dai test effettuati (Capitolo 5), libpcap cattura tutti i pacchetti, non permettendo di interrompere la cattura dei pacchetti per quel determinato indirizzo-ip/porta o per quel determinato flusso, senza considerare che non fornisce supporto

hardware il chè comporta che nel caso di una connessione superiore alla 2 Gbps, non riesce a sniffare pacchetti abbastanza in fretta, iniziando a perderli. Per questo motivo sono ricaduto sulle migliorie che un hardware offload può fornire, in quanto grazie alla FPGA questo mi ha migliorato notevolmente le prestazioni riuscendo a gestire anche diversi Gbps di connessione.

3.2 Napatech Link™ NT200A02 SmartNIC

La scelta è ricaduta su una scheda di rete Napatech, modello NT200A02. Questo perchè questa scheda, fornisce una libreria (nt.h, brevemente spiegata successivamente) che permette di classificare i flussi a livello hardware e permette inoltre di scartare i pacchetti dei flussi già classificati (utilizzando in un modo specifico le funzioni fornite, approfondito nel Capitolo 4), migliorando così notevolmente le prestazioni.
La scheda (come descritto nella documentazione ufficiale [20]) fornisce cattura di pacchetti fino a 100 Gbps senza perdita di pacchetti, inoltre fornisce un buffer che permette la bufferizzazione di pacchetti per prevenirne la perdita durante i picchi. Il timestamp dei pacchetti è inserito a livello hardware quando arrivano sulla porta di rete in modo da avere sempre un timestamp affidabile. Supporta 8x10Gbps / 2x10 / 25Gbps / 4x10 / 25Gbps/ 2x40Gbps / 2x100Gbps.

scheda di rete Napatech NT200A02 [20]

nt.h

Header contenente le funzioni fornite per comunicare con la scheda di rete Napatech. Permette varie funzionalità [21], tra le quali sniffing e filtering di pacchetti (infatti verrà utilizzata al posto della libreria libpcap, in presenza di una scheda di rete Napatech), gestione dei flussi, per cui aggiunta, rimozione, aggiornamento di contatori di flussi e controllo di terminazione di flussi, ed infine fornisce varie funzioni utili per ottenere informazioni varie sui pacchetti catturati (per esempio il timestamp, offset programmabili che puntano a precise informazioni del pacchetto, ecc).

Per quanto riguarda invece la libreria di Deep Packet Inspection e di recupero informazioni sul flusso, tra le quali il rischio relativo al flusso che indica la pericolosità del flusso, ho utilizzato la libreria open source nDPI.

3.3 nDPI

nDPI [24] è una delle migliori librerie Open Source di Deep Packet Inspection (come mostrato in alcune classifiche [8,22]), tecnica utilizzata, come descritto nel capitolo precedente, per analizzare i pacchetti fino al livello 7 del modello ISO OSI e riconoscere così i protocolli di livello applicativo, la quale ci permette di acquisire maggiori metadati sui pacchetti analizzati. Come descritto nella documentazione[23], essa inoltre fornisce, dopo aver riconosciuto il protocollo utilizzato, un certo grado di rischio, che combinandolo insieme ad una maschera viene utilizzato per stabilire se un flusso può essere dannoso o meno.

ndpi_main.h

Contenente le funzioni principali di nDPI, comprese funzioni per la gestione della hash table che sono possibili da utilizzare per la gestione dei flussi e per la classificazione dei protocolli dei pacchetti. Inoltre contiente alcuni metodi che permettono di formattare le informazioni finora ottenute in formato Json [68] (formato che utilizzerò per salvare le varie info nei files di log).

ndpi_typedef.h

Contenente le strutture dati utilizzate da nDPI, molto utili per la classificazione delle informazioni ottenute dai pacchetti.

4 Implementazione del Software

Il sistema è suddivisibile in due parti: la sezione che gestisce i flussi e quella che gestisce il parsing dei pacchetti.
A sua volta entrambe le parti sono suddivise in: Pcap e Napatech, ognuna delle quali gestisce il corrispettivo caso (ossia nel caso generale in cui venga utilizzata la libpcap per lo sniffing dei pacchetti ed il caso specifico in cui lo sniffing avvenga da parte di una scheda di rete Napatech).

Il software è composto da 3 thread:

- Il primo viene utilizzato per la corretta terminazione del programma, mettendosi in attesa passiva per n secondi, dopodichè si risveglia, controlla se è il momento di terminare il programma o meno e informa l'utente sullo stato attuale del sistema (numero pacchetti e bytes analizzati dal software);
- Il secondo viene utilizzato per la registrazione dei flussi su un file di log '.json' attraverso il quale è possibile vedere (se non specificato tramite l'apposita opzione all'avvio del programma) solamente i flussi ritenuti insicuri in base alla maschera configurata;
- Il terzo invece è il thread principale che si occupa di sniffing e parsing dei pacchetti; nel caso di utilizzo di una scheda di rete Napatech, verrà avviato anche un quarto thread che gestirà unicamente i pacchetti che non è possibile mandare a livello software a causa di picchi di traffico sulla rete (così come indi-

cato nella documentazione ufficiale [21]).
Inoltre nel caso Napatech, se il software è avviato tramite l'apposita opzione '-n < *num* >' è possibile avviare fino a num thread che gestiscono i diversi pacchetti.

4.1 Parsing dei pacchetti

Overview

Come descritto sopra, la gestione dei pacchetti è separata, in quanto le due librerie forniscono gestione degli header, contatori e puntatori ai pacchetti in modo differente (un esempio è il timestamp del pacchetto, il quale, mentre la libreria libpcap lo fornisce in microsecondi, la libreria Napatech lo fornisce in nanosecondi). Questa suddivisione avviene anche perchè la Napatech fornisce la possibilità di poter programmare offset specifici della scheda di rete, settati a livello hardware, permettendo così di poter accedere direttamente a campi specifici, senza dover fare il parsing di tutto il pacchetto permettendo così varie ottimizzazioni, cosa altrimenti impossibile tramite libpcap.
Il parsing dei pacchetti differente viene tuttavia suddiviso dal livello 2 al livello 4, dopodichè continueranno il processo di analisi del protocollo di livello 7 (modello ISO OSI) in modo uguale, questo per poter permettere anche una maggiore espandibilità ad altre schede di rete o librerie in futuro.

Specifica

All'arrivo di un pacchetto, controllo se è di tipo IPv4 o IPv6, questo in quanto mi servirà successivamente per poter fare il parsing dell'header dei vari livelli in modo corretto. Dopodichè in base al protocollo riesco a controllare le informazioni contenute nell'header di livello 3 col supporto di due 'struct' che sono rispettivamente 'struct ndpi_iphdr' per IPv4 e 'ndpi_ipv6hdr' per IPv6, ottenendo così indirizzo sorgente e destinatario del pacchetto.
Successivamente passo ad analizzare il livello 4, il quale mi permette di ottenere le porte sorgente, destinatario ed il protocollo utilizzato (TCP, UDP, ecc.); così facendo ottengo tutte le informazioni necessarie per ricercare all'interno della mia struttura dati che contiene i flussi se il pacchetto appartiene ad un flusso nuovo o meno.

Proprio a questo punto vediamo una ottimizzazione possibile con la scheda Napatech (cosa non possibile con libpcap): essa ci fornisce la possibilità di impostare due offset durante la configurazione della cattura dei pacchetti ed il colore.

```
ColorBits=FlowID, Offset0=Layer3Header[12]; ColorMask=" STR(COLOR_IPV4)
ColorBits=FlowID, Offset0=Layer3Header[8];  ColorMask=" STR(COLOR_IPV6)
```

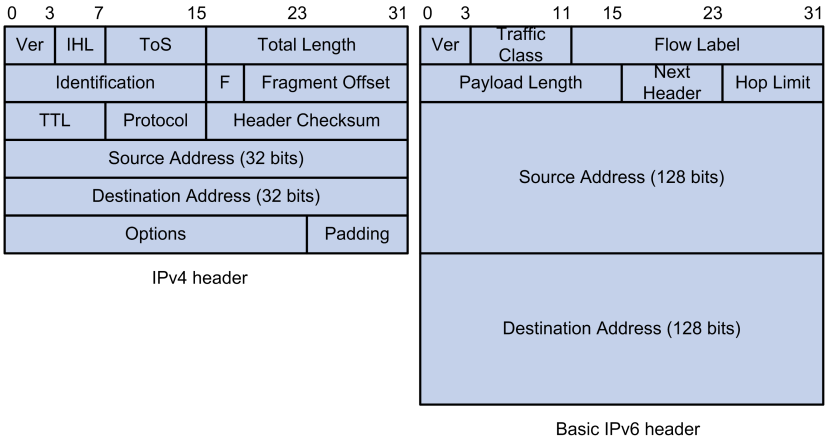
Parte di configurazione della cattura dei pacchetti

Il colore viene utilizzato per capire se il pacchetto è di tipo IPv4 o IPv6, mentre l'offset0, in questo modo, se il pacchetto è di tipo IPv4 punterà al 12esimo bytes dell'header di livello 3 (che come vediamo nella figura corrisponde all'indirizzo ip sorgente), mentre se è di tipo IPv6 punterà all'8avo bytes dell'header di livello 3 (corrisponde sempre all'indirizzo ip sorgente ma nel caso IPv6).

```
if (pDyn1->color & (1 << 6)) {
    tracer->traceEvent(1, "Packet contain an error and decoding cannot be trusted\n");
    return -1;
} else {
    if (pDyn1->color & (1 << 5)) {
        tracer->traceEvent(1, "A non IPv4,IPv6 packet received\n");
        return -1;
    } else {
        switch (pDyn1->color >> 2) {
        case 0:  // IPv4
                return DumpIPv4(args, flow, pDyn1, packet, pkt_infos);
        case 1:  // IPv6
                return DumpIPv6(args, flow, pDyn1, packet, pkt_infos);
        case 2:  // Tunneled IPv4
                return DumpIPv4(args, flow, pDyn1, packet, pkt_infos);
        case 3:  // Tunneled IPv6
                return DumpIPv6(args, flow, pDyn1, packet, pkt_infos);
        }
    }
}
```

'switch case' sul colore per capire se è IPv4 o IPv6

Il secondo offset, (offset1) punta di default all'inizio del livello 4 del pacchetto, permettendomi così sia nel caso UDP che TCP di ottenere subito le informazioni necessarie riguardanti le porte sorgente e destinatario.
Utilizzando quindi gli offset riesco ad ottenere immediatamente le informazioni necessarie per ricercare se il pacchetto appartiene ad un flusso già esistente all'interno della tabella dei flussi del programma o meno evitando così tutto il parsing del pacchetto



header Ipv4 ed IPv6

Una volta ricercato il flusso all'interno dell'apposita tabella tramite le informazioni ottenute dall'analisi del pacchetto, in caso sia un nuovo flusso, aggiungo un nuovo record alla tabella con le info necessarie (spiegato in maniera approfondita nella sezione successiva riguardante la gestione dei flussi), altrimenti vado ad aggiornare i relativi contatori del flusso e, nel caso il protocollo di livello applicativo sia già stato trovato da nDPI, allora termino e mi metto in attesa del pacchetto successivo, altrimenti richiamo gli appositi metodi per il riconoscimento del protocollo.

4.2 Riconoscimento del protocollo applicativo

Come già accennato sopra, al termine dell'analisi di un pacchetto, se il flusso relativo non è ancora stato classificato, vengono passate le informazioni necessarie alla libreria di nDPI per l'analisi del protocollo; qui possono presentarsi tre casi:

- Il caso in cui sono già stati analizzati un numero specifico di pacchetti ma la libreria non è riuscita ad individuare il protocollo, in questo caso termino l'analisi;

```
/* Try to detect the protocol */
if (pkt_infos.flow_to_process->ndpi_flow->num_processed_pkts == 0xFF) {
        return;
```

caso in cui il numero massimo di pacchetti da analizzare è stato raggiunto

- Il caso in cui c'è l'ultimo tentativo di analisi, quindi questo pacchetto è l'ultimo prima che si torni al primo caso descritto sopra;

```
} else if (pkt_infos.flow_to_process->ndpi_flow->num_processed_pkts == 0xFE) {
        /* last chance to guess something, better then nothing */
        uint8_t protocol_was_guessed = 0;
        pkt_infos.flow_to_process->guessed_protocol =
                ndpi_detection_giveup(reader->getNdpiStruct(),
                                      pkt_infos.flow_to_process->ndpi_flow,
                                      1, &protocol_was_guessed);
        pkt_infos.flow_to_process->ended_dpi = 1;
        reader->setNewFlow(true);
        reader->setIdFlow(pkt_infos.flow_to_process->flow_id);
```

Ultimo tentativo di riconoscimento del protocollo

- Il caso generale in cui il pacchetto viene passato alla funzione specifica per individuare il protocollo.

Una volta riconosciuto il protocollo, vado a controllare il rischio del flusso: il rischio è un valore impostato dalle funzioni di nDPI che va a rappresentare il tipo di anomalia che il flusso porta con sè, infatti se il rischio è a 0 non è stata riscontrata nessuna anomalia altrimenti il valore sarà diverso da 0. Il rischio viene impostato tramite una maschera, dove ogni singolo bit della maschera rappresenta una anomalia differente; per cui in base al bit impostato ad 1 del rischio è possibile risalire all'anomalia associata. Di default tutte le anomalie vengono rilevate e riconosciute come pericolo, tuttavia se necessario è possibile impostare tramite l'opzione all'avvio '-r' le anomalie che devono essere riconosciute come pericolo e quelle che non devono.

```
if(pkt_infos.flow_to_process->ndpi_flow->risk) {
    uint32_t j = mask & pkt_infos.flow_to_process->ndpi_flow->risk;
    for(int i=NDPI_NO_RISK; i<NDPI_MAX_RISK; i++)
        if(NDPI_ISSET_BIT(j, i) != 0) {
            tracer->traceEvent(1, "[** %s ** | flow %lu ] src ip: %s | dst ip: %s | src port: %u | dst port: %u\n",
                    ndpi_risk2str((ndpi_risk_enum) i), pkt_infos.flow_to_process->flow_id, src_addr_str,
                    dst_addr_str, pkt_infos.flow_to_process->src_port, pkt_infos.flow_to_process->dst_port);
            if(this->flowToJson(reader, pkt_infos.flow_to_process, 0) != 0)
                tracer->traceEvent(0, "Error while creating the record of flow %lu\n",
                        pkt_infos.flow_to_process->flow_id);

            return;
        }
}
```

In caso un flusso sia pericoloso, vengono salvate le informazioni del flusso all'interno di un file di log in formato Json [25] così da poterle mettere a disposizione dell'amministrazione del sistema.
In caso tuttavia si vogliano raccogliere e salvare tutte le informazioni su tutti i flussi (compresi quelli dichiarati non nocivi), è possibile farlo aggiungendo all'avvio l'opzione '-v'.

4.3 Gestione dei flussi

Per quanto riguarda la gestione dei flussi, sia nel caso Napatech che nel caso generico, utilizzo una tabella dei flussi implementata utilizzando la std::unordered_set, struttura dati fornita dalla libreria standard del C++ e che permette di effettuare ricerche, inserimento ed eliminazione, al caso medio, in tempo costante.
Il controllo dei flussi su quelli attivi se sono diventati inattivi, viene effettuato in un intervallo di tempo costante (ogni 60 secondi di default); se un flusso viene ritenuto inattivo (ossia non ha ricevuto più un pacchetto per un periodo di tempo che supera il massimo consentito) viene eliminato, fornendo le informazioni finora raccolte su di esso.
Nello specifico l'id del flusso, il numero di bytes e di pacchetti di quel flusso ed infine indirizzo sorgente e destinatario e porte e, nel caso in cui la classificazione sia terminata, anche il protocollo applicativo.

Come già detto precendetemente, ogni volta che arriva un nuovo pacchetto appartenente ad un flusso, vengono aggiornate le informazioni relative (quali il numero di bytes e di pacchetti) e nel caso il protocollo applicativo del flusso non sia ancora stato classificato,

viene passato alla funzione apposita di nDPI la quale richiede, oltre ai vari parametri, tre strutture che servono per aggiornare le informazioni sul flusso.

```
struct ndpi_flow_struct * ndpi_flow;
struct ndpi_id_struct * ndpi_src;
struct ndpi_id_struct * ndpi_dst;
```

Queste tre struct vengono utilizzate dalla libreria di nDPI per salvarsi varie info sul flusso stesso, utilizzate successivamente per essere salvate sul file di log Json in caso il flusso sia pericoloso.


```
{"src_ip":"172.16.16.16","dest_ip":"172.16.16.30","src_port":52028,"dst_port":636,"proto":"TCP","ndpi": {"flow_risk": {"15":"TLS (probably) not carrying HTTPS"},"proto":"TLS","category":"Web"},"tls": {"version":"TLSv1.2","client_requested_server_name":"","ja3":"75fe51990656df4f7a249d5b86aa29ae","ja3s":"","unsafe_cipher":0,"cipher":"TLS_NULL_WITH_NULL_NULL","tls_supported_versions":"TLSv1.3,TLSv1.2,TLSv1.1,TLSv1"}}
```


Esempio di record di un flusso salvato all'interno del file Json

Nello specifico contengono informazioni riguardanti il livello applicativo, quali protocollo, categoria, rischio del flusso, cipher ed altre informazioni sul livello 7.

Un'altra ottimizzazione importante che ci offre la scheda di rete è proprio nella gestione dei flussi. Per quanto riguarda la versione generale gestita con libpcap, una volta che un flusso è stato classificato i pacchetti successivi che appartengono a quel flusso continuano ad arrivare, essendo quindi sempre obbligato a fare il parsing del pacchetto, ottenere le info necessarie per cercare il flusso relativo, cercarlo e quando vedo che è completato, aggiorno i contatori relativi al numero di bytes e di pacchetti del flusso e poi termino, saltando la parte di analisi dal livello 5 in sù.
Così facendo tuttavia spreco molto tempo nel parsing e analisi dei pacchetti dei flussi già analizzati, cosa che vorrei evitare; questa

ottimizzazione la fornisce proprio la scheda di rete permettendo di registrare i flussi in una tabella di flussi gestita a livello hardware e configurando correttamente la scheda, è possibile non far più arrivare a livello software i pacchetti di flussi già registrati ad hardware, evitando così il pesante processo di parsing e di ricerca.

Nello specifico configuro gli stream della scheda in modo tale che facciano passare solamente i pacchetti di flussi non ancora registrati (figura sotto).

```
 ntplCall(hCfgStream, "Assign[StreamId=" STR(STREAM_ID_MISS) "
lowID, Offset0=Layer3Header[12]; ColorMask=" STR(COLOR_IPV4) "
nd Key(kd4, KeyID=" STR(KEY_ID_IPV4) ")==MISS");
```

In questo caso invio i pacchetti allo stream_id_miss, tutti i pacchetti con key()==MISS, ossia tutti i pacchetti appartenenti a flussi non ancora registrati ad hardware

Una volta che il protocollo di livello applicativo è stato trovato (oppure quando la libreria si arrende), registro questo flusso a livello hardware, in questo modo dopo che ho stimato il protocollo tramite nDPI non mi arrivano più pacchetti di quel flusso e questa cosa mi permette di avere buone prestazioni anche in presenza di un alto numero di pacchetti.

```
ntplCall(hCfgStream, "Assign[StreamId=Drop] = LearnFilter
STR(KEY_ID_IPV4) ", CounterSet=CSA)==" STR(KEY_SET_ID));
```

Configurazione dello stream drop, ossia tutti i pacchetti non più appartenenti allo stream miss, vengono passati allo stream drop il quale aggiorna i contatori hardware e poi li scarta, evitando di passarli così a livello software

Per quanto riguarda il salvataggio del flusso nella tabella dei flussi della scheda, prima devo creare un'oggetto di tipo 'NtFlow_t', dopodichè gli assegno i valori necessari (quali id, protocollo, ecc., figura sotto)

```
flow.id              = this->idFlow;
flow.color           = 0;
flow.overwrite       = 0;
flow.streamId        = 0;
flow.ipProtocolField = pDyn1->ipProtocol;
flow.keySetId        = KEY_SET_ID;
flow.op              = 1;
flow.gfi             = 1;
flow.tau             = 0;
```

Esempio della creazione di un flusso ed impostazione dei vari flag, quali per esempio l'id dello stream, tcp-autounlearn, ecc.

ed infine salvo indirizzi ip sorgente e destinatario e porte nell'apposito campo flow.data ed imposto il key_id in base all'indirizzo se è di tipo IPv4 o IPv6

```
switch (pDyn1->color >> 2) {
    case 0:  // IPv4
            std::memcpy(flow.keyData,      packet + pDyn1->offset0,     4);  // IPv4 src
            std::memcpy(flow.keyData + 4,  packet + pDyn1->offset0 + 4, 4);  // IPv4 dst
            std::memcpy(flow.keyData + 8,  packet + pDyn1->offset1,     2);  // TCP port src
            std::memcpy(flow.keyData + 10, packet + pDyn1->offset1 + 2, 2);  // TCP port dst
            flow.keyId = KEY_ID_IPV4;  // Key ID as used in the NTPL Key Test
            break;
    case 1:  // IPv6
            std::memcpy(flow.keyData,      packet + pDyn1->offset0,      16);  // IPv6 src
            std::memcpy(flow.keyData + 16, packet + pDyn1->offset0 + 16, 16);  // IPv6 dst
            std::memcpy(flow.keyData + 32, packet + pDyn1->offset1,      2);   // TCP port src
            std::memcpy(flow.keyData + 34, packet + pDyn1->offset1 + 2,  2);   // TCP port dst
            flow.keyId = KEY_ID_IPV6;  // Key ID as used in the NTPL Key Test
            break;
    case 2:  // Tunneled IPv4
            std::memcpy(flow.keyData,      packet + pDyn1->offset0,     4);  // IPv4 src
            std::memcpy(flow.keyData + 4,  packet + pDyn1->offset0 + 4, 4);  // IPv4 dst
            std::memcpy(flow.keyData + 8,  packet + pDyn1->offset1,     2);  // TCP port src
            std::memcpy(flow.keyData + 10, packet + pDyn1->offset1 + 2, 2);  // TCP port dst
            flow.keyId = KEY_ID_IPV4;  // Key ID as used in the NTPL Key Test
            break;
    case 3:  // Tunneled IPv6
            std::memcpy(flow.keyData,      packet + pDyn1->offset0,      16);  // IPv6 src
            std::memcpy(flow.keyData + 16, packet + pDyn1->offset0 + 16, 16);  // IPv6 dst
            std::memcpy(flow.keyData + 32, packet + pDyn1->offset1,      2);   // TCP port src
            std::memcpy(flow.keyData + 34, packet + pDyn1->offset1 + 2,  2);   // TCP port dst
            flow.keyId = KEY_ID_IPV6;  // Key ID as used in the NTPL Key Test
            break;
}
```

Impostazione dei campi indirizzo ip sorgente e destinatario e porte

Dopo aver fatto questo, scrivo il flusso all'interno della tabella dei flussi della scheda, tramite una chiamata di funzione apposita.

Una volta scritto tuttavia devo sempre controllare se il flusso di-

venta inattivo o meno; anche a questo tuttavia ci pensa la scheda ed essendo solamente obbligato a richiamare la funzione apposita per controllare le varie informazioni ricevute in seguito al controllo. Anche qui in caso di timeout, la scheda rimuove automaticamente le informazioni sul flusso passandole a livello software, permettendo così di salvarle nel file di log apposito.

```
void printFlowStreamInfo(NtFlowStream_t& flowStream)
{
    const char* ip;
    NtFlowInfo_t flowInfo;
    NtFlowStatus_t flowStatus;

    // For each element in internal flow stream queue print the flow info record.
    // The flow info record is only available when NtFlow_t.gfi is set to 1.
    // Maintaining the flow info record has a performance overhead, so if the
    // info record is not need, it is recommended to set NtFlow_t.gfi to 0.
    while(NT_FlowRead(flowStream, &flowInfo, 0) == NT_SUCCESS) {
        uint64_t tot_pkts = 0, tot_bytes = 0;

        tot_pkts = (flowInfo.packetsA + flowInfo.packetsB);
        tot_bytes = (flowInfo.octetsA + flowInfo.octetsB);

        switch(flowInfo.cause) {
                case 0:  tracer->traceEvent(2, "[flow %llu unlearned] Tot. packets: %llu |
Tot. octets: %llu | Unlearn cause: Software", flowInfo.id, tot_pkts, tot_bytes); break;
                case 1:  tracer->traceEvent(2, "[flow %llu unlearned] Tot. packets: %llu |
Tot. octets: %llu | Unlearn cause: Timeout", flowInfo.id, tot_pkts, tot_bytes); break;
                case 2:  tracer->traceEvent(2, "[flow %llu unlearned] Tot. packets: %llu |
Tot. octets: %llu | Unlearn cause: Termination", flowInfo.id, tot_pkts, tot_bytes); break;
                default: tracer->traceEvent(2, "[flow %llu unlearned] Tot. packets: %llu |
Tot. octets: %llu | Unlearn cause: Not Supported", flowInfo.id, tot_pkts, tot_bytes); break
;
        }
    }
}
```

Controllo della scadenza dei flussi dalla tabella hardware

4.4 Configurazione della cattura dei pacchetti scheda Napatech

In questa sezione descriverò in modo completo la configurazione della cattura dei pacchetti impostata sulla scheda di rete che mi ha permesso di effettuare le operazioni trattate fino a questo momento.

All'inizio della configurazione ho impostato i bits su cui la scheda calcola il valore hash per riconoscere il flusso bidirezionale.

```
   // Set new filters and flow tables settings
   ntplCall(hCfgStream, "KeyType[Name=kt4] = {sw_32_32,   sw_16_16}");
   ntplCall(hCfgStream, "KeyType[Name=kt6] = {sw_128_128, sw_16_16}");
   ntplCall(hCfgStream, "KeyDef[Name=kd4; KeyType=kt4; IpProtocolField=Outer] =
(Layer3Header[12]/32/32,  Layer4Header[0]/16/16)");
   ntplCall(hCfgStream, "keydef[Name=kd6; KeyType=kt6; IpProtocolField=Outer] =
(Layer3Header[8]/128/128, Layer4Header[0]/16/16)");
```

Come per la tabella gestita a livello software anche qui separo i flussi in base all'indirizzo ip sorgente e destinatario e alle porte.

Dopodichè definisco le impostazioni dei vari stream, definendo il valore dell'offset0, del colore e le altre informazioni di cui ho parlato sopra, comprese le configurazioni dei tre stream, miss (per i pacchetti non ancora registrati all'interno della tabella dei flussi hardware), drop (per i pacchetti registrati nella tabella dei flussi hardware), unhandled (per i pacchetti che non vengono passati a livello software a causa di una forte quantità di pacchetti che la scheda non ce la fa a gestire).

```
    ntplCall(hCfgStream, "Assign[StreamId=" STR(STREAM_ID_MISS) "; Descriptor=DYN1, ColorBits=FlowID, Offset0=
Layer3Header[12]; ColorMask=" STR(COLOR_IPV4) "] = LearnFilterCheck(0,ipv4) and Key(kd4, KeyID=" STR(KEY_ID_IP
V4) ")==MISS");
    ntplCall(hCfgStream, "Assign[StreamId=" STR(STREAM_ID_MISS) "; Descriptor=DYN1, ColorBits=FlowID, Offset0=
Layer3Header[8];  ColorMask=" STR(COLOR_IPV6) "] = LearnFilterCheck(0,ipv6) and Key(kd6, KeyID=" STR(KEY_ID_IP
V6) ")==MISS");
    ntplCall(hCfgStream, "Assign[StreamId=" STR(STREAM_ID_MISS) "; Descriptor=DYN1, ColorBits=FlowID, Offset0=
Layer3Header[12]; ColorMask=" STR(COLOR_IPV4) "] = LearnFilterCheck(1,ipv4) and Key(kd4, KeyID=" STR(KEY_ID_IP
V4) ", FieldAction=Swap)==MISS");
    ntplCall(hCfgStream, "Assign[StreamId=" STR(STREAM_ID_MISS) "; Descriptor=DYN1, ColorBits=FlowID, Offset0=
Layer3Header[8];  ColorMask=" STR(COLOR_IPV6) "] = LearnFilterCheck(1,ipv6) and Key(kd6, KeyID=" STR(KEY_ID_IP
V6) ", FieldAction=Swap)==MISS");

    ntplCall(hCfgStream, "Assign[StreamId=" STR(STREAM_ID_UNHA) "] = LearnFilterCheck(0,ipv4) and Key(kd4, Key
ID=" STR(KEY_ID_IPV4) ")==UNHANDLED");
    ntplCall(hCfgStream, "Assign[StreamId=" STR(STREAM_ID_UNHA) "] = LearnFilterCheck(0,ipv6) and Key(kd6, Key
ID=" STR(KEY_ID_IPV6) ")==UNHANDLED");
    ntplCall(hCfgStream, "Assign[StreamId=" STR(STREAM_ID_UNHA) "] = LearnFilterCheck(1,ipv4) and Key(kd4, Key
ID=" STR(KEY_ID_IPV4) ", FieldAction=Swap)==UNHANDLED");
    ntplCall(hCfgStream, "Assign[StreamId=" STR(STREAM_ID_UNHA) "] = LearnFilterCheck(1,ipv6) and Key(kd6, Key
ID=" STR(KEY_ID_IPV6) ", FieldAction=Swap)==UNHANDLED");
    ntplCall(hCfgStream, "Assign[StreamId=Drop] = LearnFilterCheck(0,ipv4) and Key(kd4, KeyID=" STR(KEY_ID_IPV
4) ", CounterSet=CSA)==" STR(KEY_SET_ID));
    ntplCall(hCfgStream, "Assign[StreamId=Drop] = LearnFilterCheck(0,ipv6) and Key(kd6, KeyID=" STR(KEY_ID_IPV
6) ", CounterSet=CSA)==" STR(KEY_SET_ID));
    ntplCall(hCfgStream, "Assign[StreamId=Drop] = LearnFilterCheck(1,ipv4) and Key(kd4, KeyID=" STR(KEY_ID_IPV
4) ", CounterSet=CSB, FieldAction=Swap)==" STR(KEY_SET_ID));
    ntplCall(hCfgStream, "Assign[StreamId=Drop] = LearnFilterCheck(1,ipv6) and Key(kd6, KeyID=" STR(KEY_ID_IPV
6) ", CounterSet=CSB, FieldAction=Swap)==" STR(KEY_SET_ID));
```

Configurazione per la gestione dei vari stream

Un'altra configurazione importante impostata è la possibilità di passare a thread differenti pacchetti appartenenti a flussi differenti. Questa spartizione viene effettuata direttamente dalla scheda di rete se inserita l'apposita richiesta (figura sotto).

```
ntplCall(hCfgStream, "HashMode=Hash2TupleSorted");
```

Configurazione per la spartizione dei pacchetti al livello software

Con questa tecnica tutti pacchetti il cui valore hash calcolato su indirizzo ip sorgente e destinatario corrisponde ad un determinato valore vengono passati ad un determinato thread o ad un altro; inoltre il valore è calcolato tramite una funzione hash tale che pacchetti con indirizzi ip sorgente e destinatario scambiati vengono passati comunque al solito thread.

4.5 Trace.h

Per quanto riguarda invece la stampa delle informazioni (sia nei log che a schermo), ho utilizzato una serie di metodi presi come spunto dalla libreria open source ntopng [49] (Trace.h e Trace.cpp). Essenzialmente questi metodi mi permettono di salvare le informazioni che dovrebbero essere stampate a schermo, all'interno di un file log, con l'aggiunta di data ed ora e "livello" di informazione tipica dei files di log (per maggiori informazioni [50]).

Una caratteristica importa è che il file di log è impostato ad una dimensione massima, così da non eccedere mai una certa quantità di spazio utilizzato; per questo motivo una volta raggiunta la dimensione massima (rappresentata in "righe", dove ogni chiamata della funzione di salvataggio dell'informazione nel log è una riga), inizio a riscrivere il file dall'inizio, sovrascrivendo così i dati più vecchi.

5 Validazione del Progetto

In questo capitolo verranno presentati i risultati finali e la validazione relativa al software sviluppato (Per validazione si intende un controllo mirato a confrontare i risultati ottenuti con i requisiti iniziali e con altri software equivalenti [60], requisiti descritti nell'introduzione della sezione 'Obiettivi ed Approfondimenti', Capitolo 2 p.8-9).

Come descritto anche nell'articolo [49], effettuare la valutazione di un software IDS è un'operazione molto complessa, in quanto sono molti i fattori che ne influenzano il risultato quali numero di nuovi flussi al secondo, numero totale di flussi, tipo del protocollo applicativo dei vari flussi, numero di pacchetti al secondo, ecc.
Per questo motivo i test che ho effettuato valutano due fattori principali per quanto riguarda gli IDS, ossia la perdita dei pacchetti che si presenta con l'aumentare del traffico di rete, in base anche al numero di nuovi flussi al secondo, ed il consumo della CPU e come esso varia nel tempo. I risultati ottenuti poi saranno confrontati con Zeek ed 'nProbe Cento' [61], software che classifica i vari flussi e permette di effettuare azioni su di essi quale anche quella di IDS, e con vari risultati riportati in alcuni articoli che trattano software IDS.

5.1 Confronto tra software con Hardware Offloading e senza

Come già descritto nel capitolo 4, ho sviluppato il software in maniera tale che è possibile avviarlo con e senza hardware offloading.

Prima ho effettuato tre differenti test sul software senza l'utilizzo dell'hardware offload per misurarne le prestazioni e successivamente le ho confrontate con i risultati ottenuti invece dagli stessi test utilizzando l'hardware offloading (i risultati sono riportati di seguito). I test eseguiti sono strutturati nel seguente modo: nel primo test vengono generati 1'000 nuovi flussi al secondo, nel secondo 10'000 nuovi flussi al secondo ed infine nel terzo 100'000 nuovi flussi al secondo ed in tutti i casi vengono generati nuovi flussi al secondo, fino al raggiungimento di un totale di 1 milione di flussi.
Questi tre test sono stati effettuati con un esempio di applicazione del software 'PF_RING', chiamato 'pfsend', il quale permette di generare traffico di rete [62], tramite il comando 'sudo pfsend -i nt:1 -b 1000000 -A 1000' (dove l'opzione '-i' viene utilizzata per specificare l'interfaccia a cui mandare i pacchetti, '-b' serve per indicare il numero massimo di flussi e '-A' indica il numero di nuovi flussi al secondo generati, per cui deve essere modificato l'ultimo valore per effettuare i 3 test correttamente). Esso genera pacchetti di dimensione pari a 64 bytes, con protocollo livello 4 (secondo il modello ISO OSI) UDP i quali sono spediti all'interfaccia di rete con una velocità pari a 17 Gbps, ossia mediamente 25 milioni di pacchetti al secondo.
Questa modalità di test va a verificare il comportamento del software al caso pessimo, in quanto i pacchetti generati da 'pfsend' sono pacchetti senza un protocollo applicativo specifico, per cui il software necessiterà, per ogni flusso, di analizzare il numero massimo di pacchetti necessari alla libreria nDPI prima di arrendersi nell'analisi del protocollo applicativo e di classificarlo conseguentemente come

flusso con protocollo applicativo sconosciuto.

Per avviare il software senza hardware offloading ho utilizzato il seguente comando ‘sudo ./nDPILight -i pcap:nt:0‘.

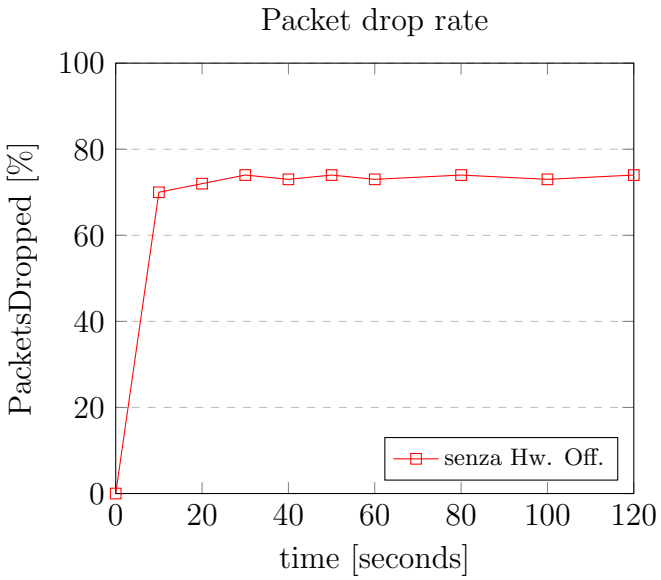


Figura 1: vengono spediti 1’000 nuovi flussi al secondo, fino al raggiungimento di 1 Milione di flussi totali. Viene testata la percentuale di perdita dei pacchetti del software senza l’utilizzo di Hardware offloading.

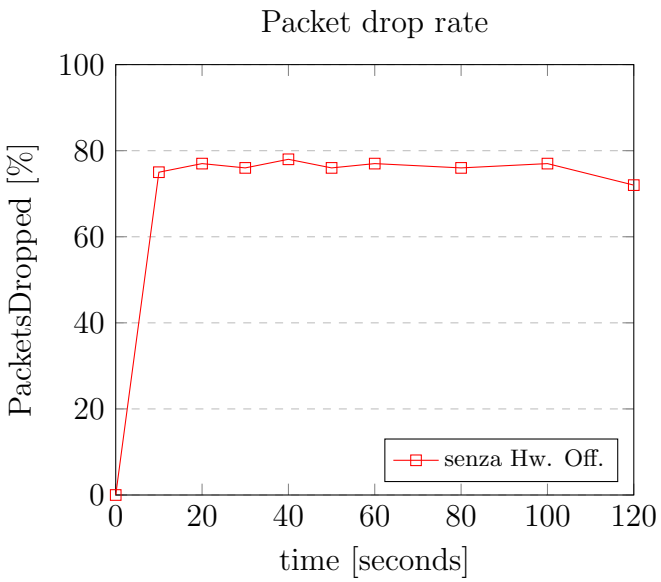


Figura 2: vengono spediti 10'000 nuovi flussi al secondo, fino al raggiungimento di 1 Milione di flussi totali. Viene testata la percentuale di perdita dei pacchetti del software senza l'utilizzo di Hardware offloading.

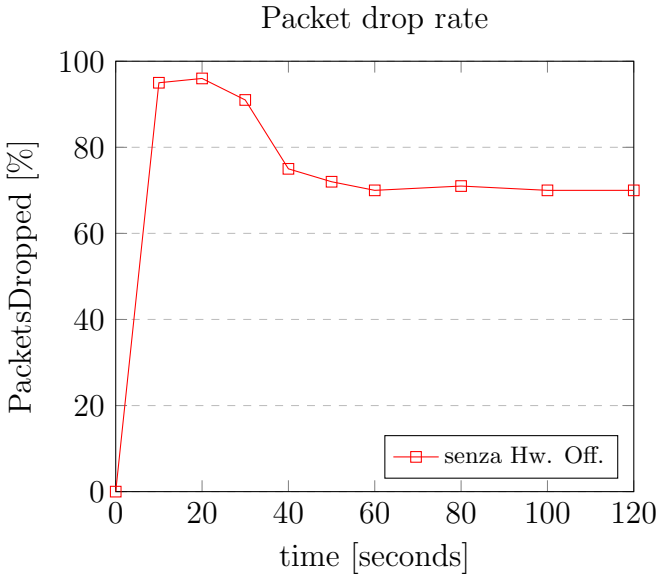


Figura 3: vengono spediti 100'000 nuovi flussi al secondo, fino al raggiungimento di 1 Milione di flussi totali. Qui viene testata la percentuale di perdita dei pacchetti del software senza l'utilizzo di Hardware offloading.

Questi risultati mostrano come, partendo già da un numero basso di nuovi flussi al secondo, il software perde buona parte dei pacchetti ricevuti (nella figura 1, perde tra il 70% ed il 74% dei pacchetti, mentre nella figura 2 perde tra il 75 ed il 78% di pacchetti) e questo dipende dal fatto che per ogni pacchetto il software deve recuperare tutte le informazioni necessarie per effettuare una ricerca all'interno della tabella dei flussi e consecutivamente controllare se il flusso esiste già oppure se è nuovo. In caso sia nuovo, deve aggiungere il nuovo record alla tabella dei flussi ed infine, se la fase di rilevamento del protocollo applicativo non è ancora terminata deve anche richiamare le funzioni necessarie per effettuarla.
La fase di recupero info e ricerca richiede mediamente 5'500 cicli di CPU, la fase di aggiunta 13'500 cicli di CPU e la fase di rilevamento circa 60'000 cicli di CPU.
Di queste tre fasi di processamento dei pacchetti, le più costose in realtà non vengono sempre effettuate, in quanto la seconda viene effettuata solamente per il primo pacchetto del flusso e la terza viene evitata nel caso in cui il rilevamento del protocollo sia già terminato e quindi non sia più necessario effettuarlo (questa cosa si può osservare nella figura 2 dopo 100 secondi e si sottolinea in maniera più chiara nella figura 3 dove, dopo 30 secondi la maggior parte dei flussi è già stata aggiunta alla tabella dei flussi e la fase di rilevamento è terminata, per cui cala drasticamente la quantità di pacchetti persi).
Anche se le due operazioni che richiedono più cicli vengono evitate per la maggior parte dei pacchetti, l'operazione di ricerca viene sempre effettuata, anche nel caso in cui il flusso sia già stato classificato interamente.
Proprio in questo, l'hardware offloading viene in aiuto, diminuendo

fortemente il carico di lavoro a livello software, come spiegato approfonditamente nel capitolo 4: una volta classificato il flusso esso viene aggiunto alla tabella dei flussi hardware e non vengono più passati pacchetti a livello software appartenenti a quel flusso, permettendo di evitare tutte le operazioni sopra elencate e dovendole effettuare solamente per i flussi nuovi o con protocollo applicativo non ancora finito di analizzare.

La differenza si può infatti notare dai grafici sottostanti, che riportano sia il software con utilizzo di hardware offloading (per avviarlo utilizzare il comando 'sudo ./nDPILight -i nt:0') che senza.

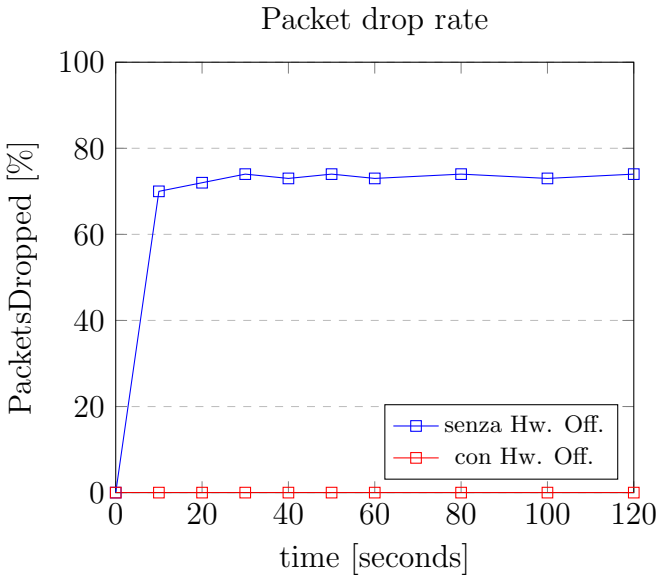


Figura 4: vengono inviati 1'000 nuovi flussi al secondo fino al raggiungimento di 1 Milione di flussi totali. A confronto la percentuale di perdita di pacchetti del software senza e con l'utilizzo di Hardware offloading.

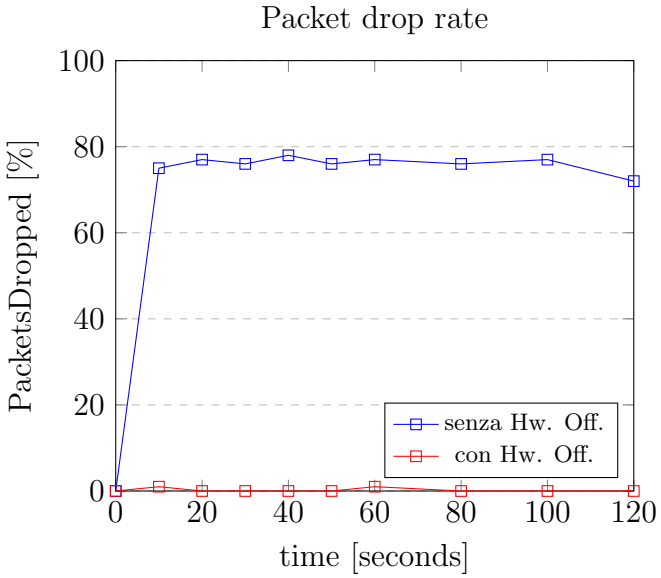


Figura 5: vengono inviati 10'000 nuovi flussi al secondo fino al raggiungimento di 1 Milione di flussi totali. A confronto la percentuale di perdita di pacchetti del software senza e con l'utilizzo di Hardware offloading.

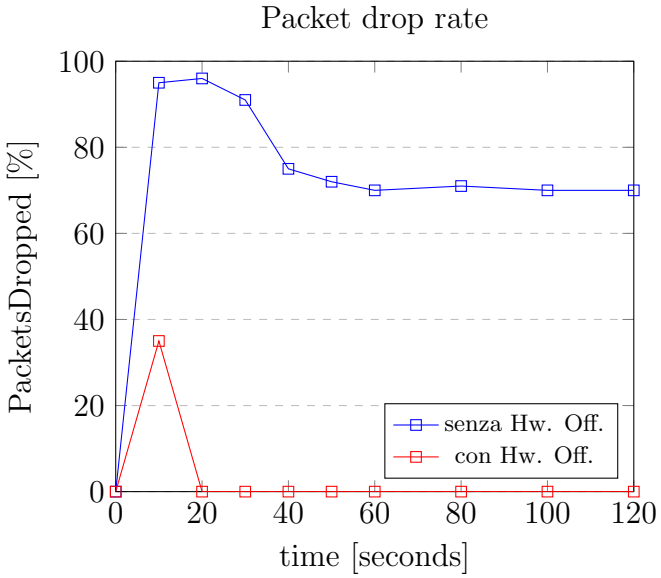


Figura 6: vengono inviati 100'000 nuovi flussi al secondo fino al raggiungimento di 1 Milione di flussi totali. A confronto la percentuale di perdita di pacchetti del software senza e con l'utilizzo di Hardware offloading.

Come si può vedere dai grafici il software che utilizza l'accelerazione hardware non ha perdita di pacchetti nelle prime due figure (eccezion fatta per l'1% di perdita al secondo 60 della figura 5, dovuta al controllo della scadenza dei flussi effettuata ogni 60 secondi) e ha solamente una perdita compresa tra il 25% ed il 30% di pacchetti nel caso di 100'000 nuovi flussi al secondo (infatti in seguito ad un test effettuato per testare i limiti del software, il limite di nuovi flussi al secondo che riesce a sostenere senza avere perdita di pacchetti alcuna, è di circa 30'000 nuovi flussi al secondo).

Non solo il miglioramento si può vedere nella perdita di pacchetti, ma anche nell'utilizzo di CPU.

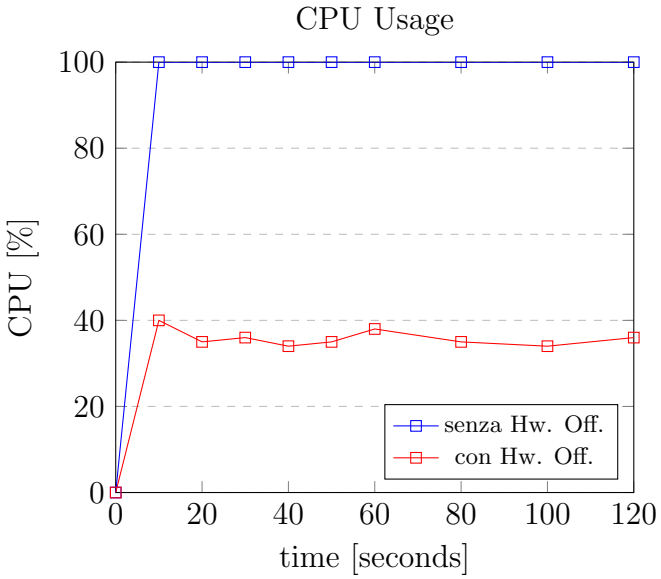


Figura 7: percentuale di utilizzo della CPU del software senza e con uso di hardware offloading nel caso di ricezione di 1'000 nuovi flussi al secondo fino ad un limite di 1 Milione di flussi totali.

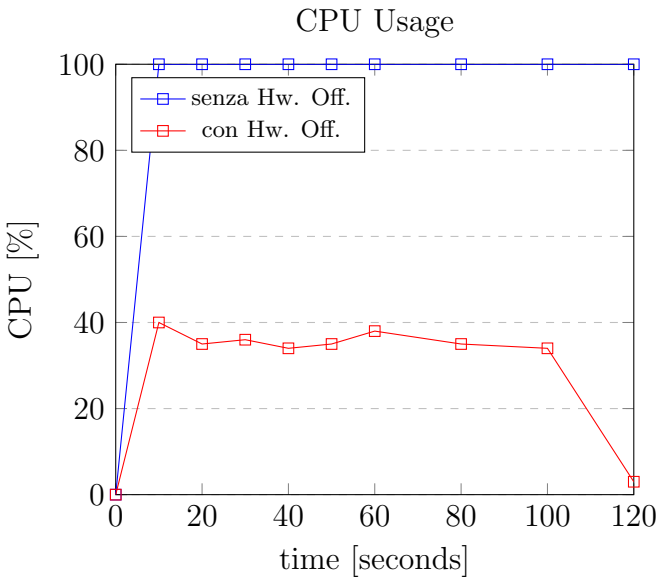


Figura 8: percentuale di utilizzo della CPU del software senza e con uso di hardware offloading nel caso di ricezione di 10'000 nuovi flussi al secondo fino ad un limite di 1 Milione di flussi totali.

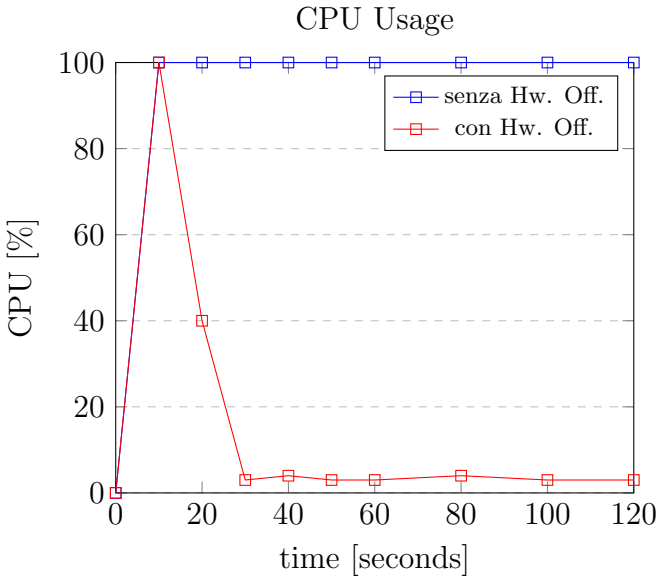


Figura 9: percentuale di utilizzo della CPU del software senza e con uso di hardware offloading nel caso di ricezione di 100'000 nuovi flussi al secondo fino ad un limite di 1 Milione di flussi totali (da notare l'utilizzo di risorse del software con Hardware offloading dopo il secondo 20, ossia in presenza di unicamente flussi già classificati).

Una perdita di pacchetti comporta anche un utilizzo massiccio di CPU [65], infatti è possibile osservare dalle figure 7,8 e 9 come il software senza hardware offloading (quindi ha sempre perdita di pacchetti) utilizzi costantemente la CPU al 100%, a differenza del software utilizzante hardware offloading che nel momento in cui deve catturare e processare i vari pacchetti usufruisce dal 35% al 40% di CPU.
E' possibile inoltre notare nelle figure 8 (dopo 120 secondi) e 9 (dopo 30 secondi) quanto effettivamente l'hardware offloading aiuti il software, infatti l'utilizzo di CPU cala al 3/4% nel caso tutti i flussi siano stati analizzati, per cui deve pensarci solamente l'hardware non più il software ad estrarre le informazioni utili dai pacchetti.

Infine, ho effettuato un ultimo test per verificarne la scalabilità con l'utilizzo di multithreading, infatti abilitando l'opzione '-n' ed indicando successivamente il numero di thread che devono gestire i pacchetti è possibile avviare il software con il relativo numero di thread che catturano e raccolgono le informazioni dai pacchetti (fino ad un massimo di 8 thread, limite imposto dalla scheda di rete). Il test è stato effettuato utilizzando 3 thread ('sudo ./nDPILight -i nt:0 -n 3').
Come era infatti prevedibile dai risultati finora ottenuti, il software con 100'000 nuovi flussi al secondo perde solamente il 2% dei pacchetti totali ricevuti e va ad utilizzare, nel punto di perdita, il 60% di CPU, per poi ricalare ad il 3% dovuto all'utilizzo dell'accelerazione hardware.

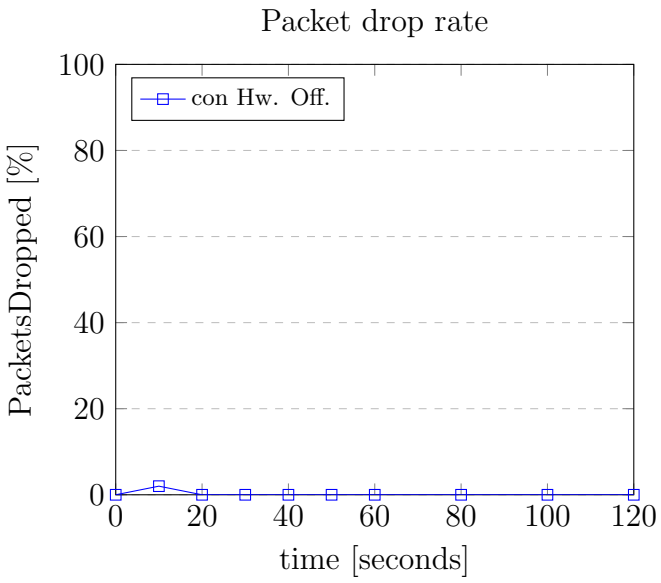
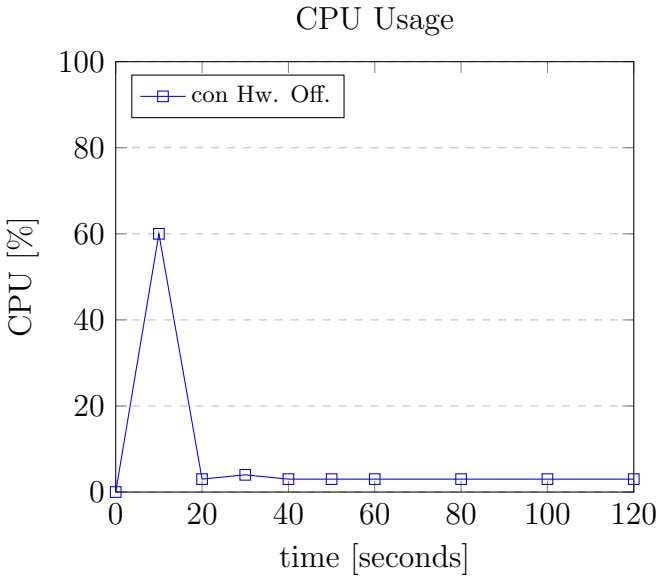


Figura 10: percentuale di perdita dei pacchetti e di utilizzo della CPU del software utilizzando l'Hardware offloading con l'utilizzo di 3 thread che gestiscono i pacchetti in caso di ricezione di 100'000 nuovi flussi al secondo.



Validazione

In base a quanto descritto all'inizio del capitolo 2 gli obiettivi prefissiti sono stati raggiunti, infatti il software riesce senza alcun problema a supportare la 10 Gbps. Inoltre anche la 20 Gbps viene supportata senza premesse irragionevoli (quali un numero massimo di 30'000 nuovi flussi al secondo) e nel caso queste premesse non siano soddisfatte, come possiamo vedere dai risultati, abbiamo comunque una perdita ragionevole di pacchetti (figura 9).
Inoltre i test sono stati effettuati (escluso il test riportato nella figura 10) con l'utilizzo di un solo thread che gestisce i pacchetti ma il software supporta anche la versione multithread e non essendoci concorrenza tra i vari thread le prestazioni migliorano considerevolmente utilizzando più thread (come viene evidenziato nella figura 10).
Sono stati inoltre raggiunti anche obiettivi inaspettati per quanto riguarda l'utilizzo di CPU dato che, nel caso in cui il software non perda pacchetti (figura 7 e 8), l'utilizzo della CPU è in un range compreso tra il 30% ed il 40%.

5.2 Test con IDS vari

Qui vado a confrontare il software sviluppato con altri software IDS, quali Cento [61], Zeek(Bro) [3] ed altri IDS molto simili che utilizzano invece hardware offloading [63, 64].

Sia Cento che Zeek(Bro) sono stati scelti perchè tra i migliori IDS open-source [6, 48, 61] di tipo anomaly-based in circolazione che non utilizzano accelerazione hardware.
Cento è un software che permette di abilitare la funzionalità di IDS tramite un'opzione apposita; tuttavia, in quanto Cento non supporta hardware offloading, l'ho confrontanto con il software sviluppato senza l'utilizzo dell'accelerazione hardware.
Anche in questo caso ho effettuato gli stessi identici test che avevo effettuato tra software con e senza offloading ed in seguito sono riportati i risultati (per eseguire Cento, 'sudo cento -i nt:0 -D 2 -w 1024000', dove l'opzione -D 2 avvia le funzionalità di IDS).

Zeek è un analizzatore passivo del traffico di rete open source. Utilizzato principalmente come sistema di monitoraggio per ispezionare tutto il traffico su un link al fine di individuare attività sospette (approfondimento nel Capitolo 2, pag. 23). Zeek permette di effettuare l'analisi dei pacchetti utilizzando libpcap [17] oppure PF_RING [59] (libreria che cattura anch'essa pacchetti con tuttavia funzionalità aggiuntive e più efficiente rispetto a libpcap).

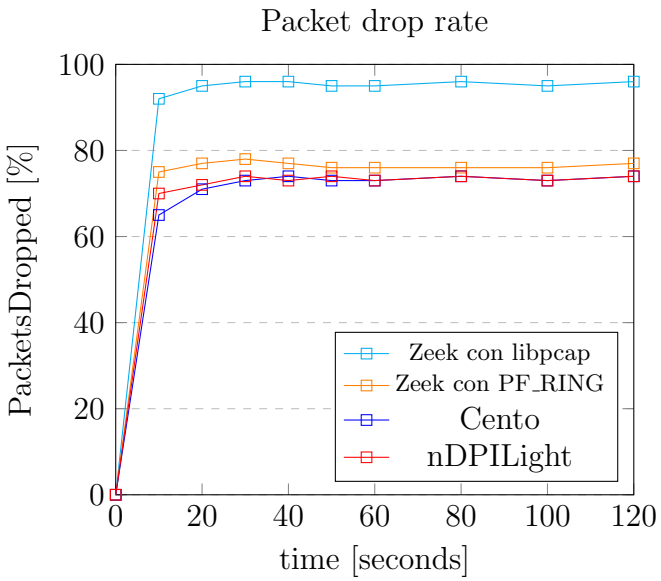


Figura 11: vengono inviati 1'000 nuovi flussi al secondo fino al raggiungimento di 1 Milione di flussi totali. A confronto la percentuale di perdita di pacchetti del software senza l'utilizzo dell'Hardware Offloading, Cento, Zeek con l'utilizzo di libpcap e Zeek con l'utilizzo di PF_RING.

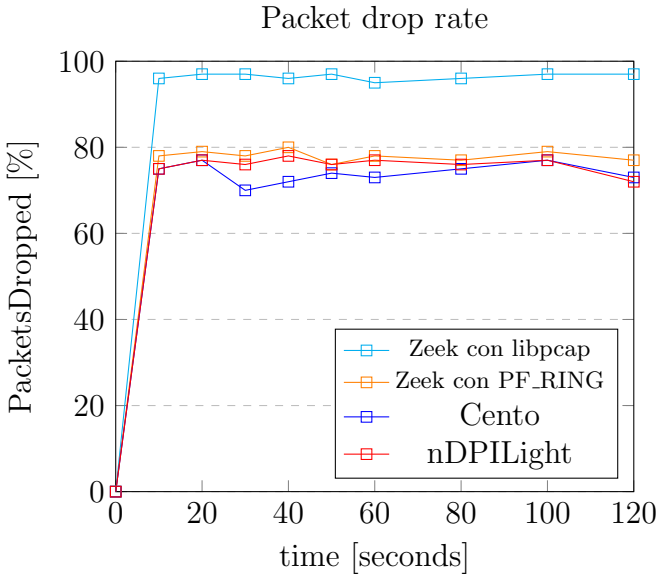


Figura 12: vengono inviati 10'000 nuovi flussi al secondo fino al raggiungimento di 1 Milione di flussi totali. A confronto la percentuale di perdita di pacchetti del software senza l'utilizzo dell'Hardware Offloading, Cento, Zeek con l'utilizzo di libpcap e Zeek con l'utilizzo di PF_RING.

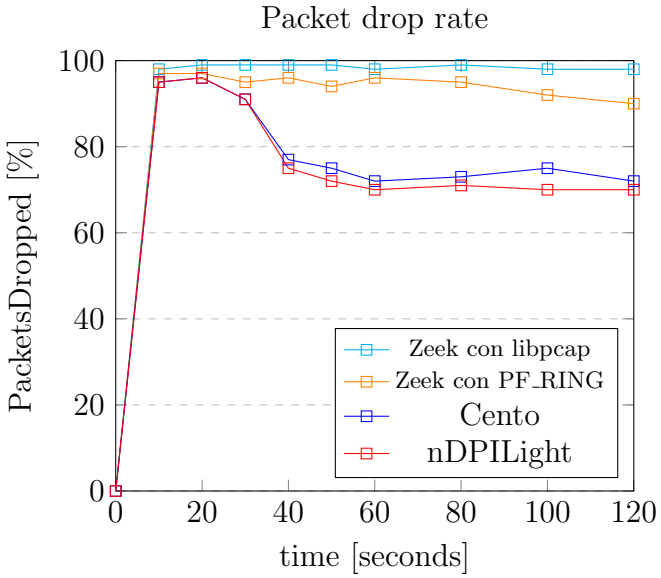


Figura 12: vengono inviati 100'000 nuovi flussi al secondo fino al raggiungimento di 1 Milione di flussi totali. A confronto la percentuale di perdita di pacchetti del software senza l'utilizzo dell'Hardware Offloading, Cento, Zeek con l'utilizzo di libpcap e Zeek con l'utilizzo di PF_RING.

Come possiamo osservare, sia Cento che il software sviluppato nel tirocinio hanno un comportamento molto simile, sia per quanto riguarda la perdita di pacchetti che per quanto riguarda l'utilizzo della CPU, droppando entrambi una alta quantità di pacchetti utilizzano il 100% di CPU costantemente; invece si vede un leggero distacco rispetto alle prestazioni di Zeek con l'utilizzo di PF_RING nei primi due test, aggravato tuttavia nell'ultimo test, mentre un enorme divario di prestazioni si può osservare in Zeek che utilizza libpcap. In alcuni momenti Cento è il migliore, mentre in altri è leggermente peggiore di nDPILight, in quanto oltre che ad avere la funzione di IDS esegue altre operazioni di analisi più approfondita sui flussi (a questo si deve il leggero peggioramento di prestazioni). Infine ho effettuato un ultimo test con un file pcap di 7GB di

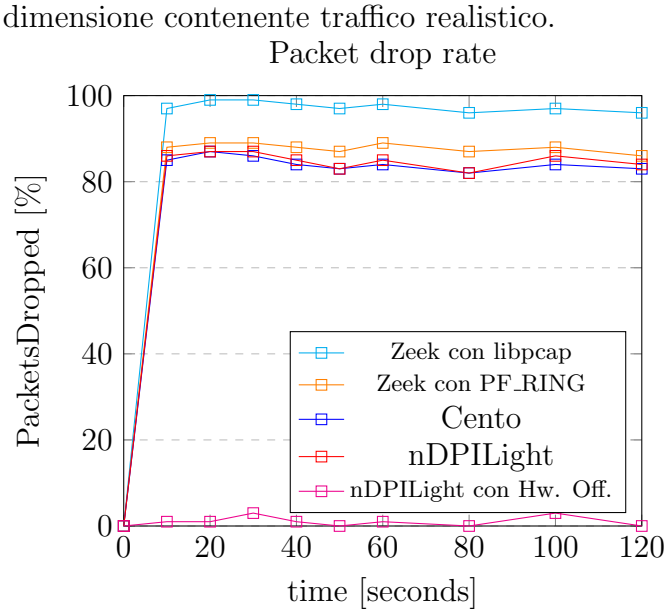


Figura 13: qui viene inviato ad una velocità di 17Gbps un traffico realistico catturato in un file '.pcap' di circa 7 GB di dimensione e mandato a ripetizione; il numero di flussi è molto variabile e varia da 14'000 nuovi flussi nuovi al secondo a 40'000 circa. A confronto la percentuale di perdita di pacchetti del software senza e con l'utilizzo dell'Hardware Offloading, Cento, Zeek con l'utilizzo di libpcap e Zeek con l'utilizzo di PF_RING.

Per quanto riguarda la parte di accelerazione hardware ho invece confrontato i risultati ottenuti con due software IDS, presentati in General IDS Acceleration for High-Speed Networks [63] e in FIX-IDS: A High-Speed Signature-based Flow Intrusion Detection System [64].

In tutti e tre i casi i software hanno lo stesso principio, ossia quello di collezionare informazioni sui flussi, nel frattempo valutare anche se i flussi potrebbero essere rischiosi o meno e nel caso prendere decisioni sulle azioni da effettuare ed una volta finita la classificazione, passare i pacchetti all'hardware e lasciarlo gestire ad esso. Tuttavia nel secondo articolo il software si distacca dagli altri due per un aspetto, ossia i flussi generati sono flussi IPFIX (seguono un protocollo

di nome IPFIX, sviluppato da CISCO [66]) gestiti da un software di nome 'nProbe' [67]; inoltre esso si basa sul classificare principalmente traffico che utilizza protocollo Http.
Da un punto di vista di prestazioni in realtà quest'ultimo si distacca un po' dagli altri due software (quello sviluppato durante il tirocinio e quello presentato in [63]) infatti possiamo osservare dai risultati riportati che inizia a perdere pacchetti con 5'000/6'000 nuovi flussi al secondo.

Allo stesso tempo il primo, a differenza degli altri due, non raccoglie più informazioni sui flussi ormai classificati (quindi i flussi ritenuti sicuri o meno) e va a scartare tutti i metadati contenuti in essi, cosa a parer mio errata dato che si vanno a perdere molte informazioni che possono tornare utili (soprattutto in caso di attacco non rilevato dal software).
Da un punto di vista di prestazioni tuttavia è in realtà molto simile al software sviluppato durante il tirocinio, come è possibile osservare dai risultati riportati sulla pubblicazione.

Riassunto risultati

Percentuale perdita pacchetti			
Test numero	nDPILight senza Hw.Off.	Cento	nDPILight con Hw.Off.
Test n.1	70-74%	65-74%	0%
Test n.2	75-78%	70-77%	0%
Test n.3	91-97%	91-97%	30%
Test n.3 dopo 30 sec	70-75%	72-77%	0%
Test realistico	82-87%	82-86%	0-3%

Percentuale perdita pacchetti		
Test numero	Zeek con libpcap	Zeek con PF_RING
Test n.1	92-96%	75-78%
Test n.2	96-97%	77-80%
Test n.3	98-99%	95-98%
Test n.3 dopo 30 sec	98-99%	90-95%
Test realistico	96-99%	86-89%

N. di pacchetti catturati al sec.			
Tipo di info	nDPILight senza Hw.Off.	Cento	nDPILight con Hw.Off.
Test n.1	6.5-7.5 Mpps	6.5-8.5 Mpps	25 Mpps
Test n.2	5.5-6 Mpps	5.7-7.5 Mpps	25 Mpps
Test n.3	0.6-2 Mpps	0.6-2 Mpps	16 Mpps

N. di pacchetti catturati al sec.		
	Zeek con libpcap	Zeek con PF_RING
Test n.1	1-2 Mpps	5.3-6 Mpps
Test n.2	0.75-1 Mpps	5-5.7 Mpps
Test n.3	0.25-0.5 Mpps	0.5-1.2 Mpps

Mpps sta per Milioni di pacchetti al secondo

N. di nuovi flussi catturati al sec.			
Tipo di info	nDPILight senza Hw.Off.	Cento	nDPILight con Hw.Off.
Test n.1	1'000	1'000	1'000
Test n.2	10'000	10'000	10'000
Test n.3	35'000-45'000	35'000-50'000	100'000

N. di pacchetti catturati al sec.		
	Zeek con libpcap	Zeek con PF_RING
Test n.1	1'000	1'000
Test n.2	10'000	10'000
Test n.3	20'000-25'000	35'000-45'000

Percentuale utilizzo CPU			
Test numero	nDPILight senza Hw.Off.	Cento	nDPILight con Hw.Off.
Test n.1	100%	100%	40%
Test n.2	100%	100%	40%
Test n.3	100%	100%	100%
Test n.3 dopo 30 sec	100%	100%	3-4%

Percentuale utilizzo CPU		
Test numero	Zeek con libpcap	Zeek con PF_RING.
Test n.1	100%	100%
Test n.2	100%	100%
Test n.3	100%	100%
Test n.3 dopo 30 sec	100%	100%

Confronto software con Hardware Acceleration			
	nDPILight con Hw.Off.	FIXIDS [64]	General IDS [63]
Raccolta dati post classificazione	SI	SI	NO
Inizio perdita pacchetti	oltre 10 Gbps	3 Gbps	oltre 10 Gbps

6 Lavoro Futuro

In futuro potrebbe essere interessante apportare alcune migliorie al software; una di queste potrebbe essere quella di renderlo oltre che IDS anche un software IPS (Intrusion Prevention System). In questo modo si fornirebbe non solo la possibilità di individuare il traffico malevolo ma anche l'opportunità di poterlo scartare, così da bloccare le intrusioni invece che limitarsi soltanto ad individuarle.

Una problematica ravvisabile in questo software è invece l'assenza di analisi delle relazioni che potrebbero esserci tra flussi differenti; per esempio, questo software non è in grado di individuare un port scanning, vale a dire quella tecnica progettata per sondare un server o un host al fine di stabilire quali porte siano in ascolto sulla macchina. Questo proprio perchè esso non va ad analizzare possibili legami tra i vari flussi, per cui non riesce a riconoscere se un host ha inviato pacchetti a tutte le porte effettuando così il port scanning.

Altra problematica affrontata è l'efficienza del software senza l'utilizzo del supporto hardware fornito dalla Napatech. Sarebbe infatti molto interessante ricercare delle tecniche per migliorarlo, per esempio utilizzando la libreria 'PF_RING'[59] al posto di libpcap per la cattura dei pacchetti, permettendo anche al software senza il supporto dell'accelerazione hardware di fornire una buona difesa del sistema senza dover obbligatoriamente avere una scheda Napatech.

7 Conclusione

Il recente incremento di traffico malevolo sulla rete ha portato all' aumento dello sviluppo di software efficienti che permettono di difendersi da essi (come gli IDS). Tuttavia gli attuali sistemi di IDS continuano ad avere diversi problemi, sia da un punto di vista di efficienza, in quanto possono condurre all'effetto "collo di bottiglia" in caso di rete superiore a 10 Gbps, sia da un punto di vista di affidabilità in quanto la probabilità di non identificare traffici malevoli è tanto alta così come quella di generare falsi positivi.

L'obiettivo del progetto portato avanti durante il tirocinio formativo è stato quello di proporre una soluzione accettabile, come è stato dimostrato dal risultato dei vari test effettuati e descritti in precedenza.
Si è inoltre sottolineato come le prestazioni di un IDS dipendano in modo particolare da fattori quali il numero di nuovi flussi al secondo e il tempo che impiega il software a classificare il protocollo applicativo dei vari flussi.

Nello specifico è stato approfondito l'utilizzo della tecnica della Deep Packet Inspection per poter mitigare la scarsa affidabilità dei sistemi IDS, in quanto andando ad analizzare fino al livello 7 i pacchetti (modello ISO OSI), la probabilità di creare falsi positivi o di non identificare traffici malevoli è molto bassa, pagando tuttavia un prezzo sull'efficienza. Inoltre è stato approfondito come l'utilizzo del supporto Hardware possa controbilanciare i problemi di efficienza

derivanti dall'uso della DPI e degli IDS in generale, riuscendo a colmare le lacune di entrambi e ponendo per cui una buona soluzione al problema.

8 Appendice

8.1 Codice sorgente

Il codice sorgente dell'IDS, assieme da un file README.md che mostra i requisiti, la compilazione e l'esecuzione del software, è reperibile al seguente link:
https://github.com/MatteoBiscosi/Tirocinio

9 Riferimenti

1. *Intrusion Detection System- Types and Prevention.*
   B. Santos Kumar, T. Ch, Ra Sekhara Phani Raju, M. Ratnakar, Sk. Dawood Baba, N. Sudhakar.
   International Journal of Computer Science and Information Technologies, 2013.

2. *7 Best Intrusion Detection Software and Latest IDS Systems.*
   https://www.dnsstuff.com/network-intrusion-detection-software.

3. *Zeek Documentation.*
   https://docs.zeek.org/en/current/

4. *Snort Documentation.*
   https://www.snort.org/documents

5. *Suricata Documentation.*
   https://suricata.readthedocs.io/en/suricata-5.0.3/

6. *Intrusion Detection Systems Explained: 13 Best IDS Software Tools Reviewed.*
   https://www.comparitech.com/net-admin/network-intrusion-detection-tools/

7. *What Is The Difference Between IDS And IPS?.*
   https://purplesec.us/intrusion-detection-vs-intrusion-prevention-systems/

8. *Deep Packet Inspection.*
   https://www.ittsystems.com/deep-packet-inspection-guide-and-software/

9. *Network Monitoring Approaches: An Overview.*
Jakub Svoboda, Ibrahim Ghafir, Vaclav Prenosil.
International Journal of Advances in Computer Networks and Its Security, 30 October, 2015.

10. *Using string matching for deep packet inspection.*
Po-Ching Lin, Ying-Dar Lin, Yuan-Cheng Lai, Tsern-Huei Lee.
IEEE Computer Society 2008.

11. *Deep Packet Inspection for Intrusion Detection Systems: A Survey.*
Tamer AbuHmed, Abedelaziz Mohaisen, and DaeHun Nyang.
Information Security Research Laboratory, Inha University, Incheon 402-751, Korea.

12. *Napatech SmartNIC solution for hardware offload*
https://www.napatech.com/support/resources/solution-descriptions/napatech-smartnic-solution-for-hardware-offload/

13. *Compression PoC for Nokia proves 40x performance improvement*
https://www.napatech.com/ support/resources/case-studies/compression-poc-for-nokia- proves-40x-performance-improvement/

14. *Introduction to Cisco IOS NetFlow - A Technical Overview*
https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html

15. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information Types and attributes [online]. [cit. 2015- 04-21].*
http://tools.ietf.org/search/rfc7011.

16. *Programming with libpcap - Sniffing the network from our own application.*
Luis Martin Garcia. hakin9 2/2008.

17. *libpcap*
https://www.tcpdump.org/

18. *Packet sniffing: a brief introduction.*
S. Ansari, S.G. Rajeev, H.S. Chandrashekar.
IEEE Potentials, Dec. 2002-Jan. 2003.

19. *Network Traffic Analysis and Intrusion Detection Using Packet Sniffer.*
Mohammed Abdul Qadeer, Arshad Iqbal, Mohammad Zahid, Misbahur Rahman Siddiqui.
Second International Conference on Communication Software and Networks, 26-28 Feb. 2010.

20. https://www.napatech.com/support/resources/data-sheets/link-nt200a02-smartnic/

21. https://docs.napatech.com/reader/pJUmcBG 8TBSxBwqHxwLrQ/d2adHH6XXAaoNTkKdeGKyw

22. *The Top 12 Dpi Open Source Projects*
https://awesomeopensource.com/projects/dpi

23. https://github.com/ntop/nDPI

24. *nDPI: Open-Source High-Speed Deep Packet Inspection.*
Luca Deri, Maurizio Martinelli, Tomasz Bujlow, Alfredo Cardigliano.
2014 International Wireless Communications and Mobile Computing Conference (IWCMC), 4-8 Aug. 2014.

25. *Foundations of JSON Schema.*
Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, Domagoj Vrgoč.
WWW '16: Proceedings of the 25th International Conference on World Wide Web, April 2016.

26. *DIDS (Distributed Intrusion Detection System) Motivation, Architecture, and An Early Prototype.*
Steven R. Snapp, James Brentano, Gihan V. Dias, Terrance L. Goan, L. Todd Heberlein, Che-Lin Ho, Karl N. Levitt, Biswanath Mukherjee, Stephen E. Smahal, Tim Grance, Daniel M. Teal, and Doug Mansur.
Computer Security Laboratory, Division of Computer Science, University of California.

27. *The Base-Rate Fallacy and the Difficulty of Intrusion Detection.*
Stefan Axelsson.
ACM Transactions on Information and System Security, August 2000.

28. *Survey of Current Network Intrusion Detection Techniques.*
Richa Srivastava, Prof. & Head Vineet Richhariya.
International Conference on Recent Trends in Applied Sciences with Engineering Applications, 2013.

29. *Wired and wireless intrusion detection system: Classifications,good characteristics and state-of-the-art..* Tarek S. Sobh.
Information System Department, Egyptian Armed Forces, Cairo, Egypt August 2005.

30. *BlueBoX: A Policy-Driven, Host-Based Intrusion Detection System.*
Suresh N. Chari and Pau-Chen Cheng.
IBM Thomas J. Watson Research Center, May 2003.

31. *Signature based Intrusion Detection for Wireless Ad-Hoc Networks: A Comparative study of various routing protocols.*
Farooq Anjum, Dhanant Subhadrabandhu and Saswati Sarkar.
2003 IEEE 58th Vehicular Technology Conference.

32. *The Design and Implementation of Host-based Intrusion Detection System.*
Ying Lin, Yan Zhang, Yang-jia Ou.
2010 Third International Symposium on Intelligent Information Technology and Security Informatics.

33. *Host-based Intrusion Detection System.*
L. Vokorokos, A. Baláž.
2010 IEEE 14th International Conference on Intelligent Engineering Systems.

34. *Intrusion Detection: Host-Based and Network-Based Intrusion Detection Systems.*
Harley Kozushko.
2003, Independent Study.

35. *A hybrid intrusion detection system design for computer network security.*
M. Ali AydınA. Halim ZaimK. Gökhan Ceylan.
Computers and Electrical Engineering, February 2009.

36. *A Review of Anomaly based Intrusion Detection Systems.*
V. Jyothsna, V. V. Rama Prasad, K. Munivara Prasad.
International Journal of Computer Applications, August 2011.

37. *Firewall Evolution - Deep Packet Inspection.*
Ido Dubrawsky.
SecurityFocus HOME Infocus, 2003.

38. *Performance Improvement of Deep Packet Inspection for Intrusion Detection.*
Thaksen J. Parvat, Pravin Chandra.
2014 IEEE Global Conference on Wireless Computing & Networking (GCWCN)

39. *A survey on deep packet inspection.*
Reham Taher El-Maghraby, Nada Mostafa Abd Elazim, Ayman M. Bahaa-Eldin.
2017 12th International Conference on Computer Engineering and Systems. (ICCES)

40. *High-Performance Pattern-Matching for Intrusion Detection.*
J. van Lunteren. Proceedings IEEE INFOCOM 2006.
25TH IEEE International Conference on Computer Communications.

41. *Real-Time Classification of Multimedia Traffic Using FPGA.*
Weirong Jiang, Maya Gokhale.
2010, International Conference on Field Programmable Logic and Applications.

42. *Hardware acceleration for power efficient deep packet inspec-*

tion.
Zhou Yachao.
PhD thesis, Dublin City University, 2012.

43. *Specialized hardware components.*
Dimitrios Serpanos, Tilman Wolf.
Architecture of Network Systems, 2011

44. *Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations.*
Nagender Bandi, Chengyu Sun, Divyakant Agrawal, Amr El Abbadi.

45. *Critical Path Based Hardware Acceleration for Cryptosystems.*
Chen Liu , Rolando Duarte , Omar Granados , Jie Tang , Jean Andrian.

46. *Hardware Acceleration*
https://www.omnisci.com/technical-glossary/hardware-acceleration

47. *Separating VNF and Network Control for Hardware-Acceleration of SDN/NFV Architecture.*
Tong Duan , Julong Lan, Yuxiang Hu, Penghao Sun, 11 August 2017.

48. *Performance Evaluation Study of Intrusion Detection Systems.*
Adeeb Alhomoud, Rashid Munir, Jules Pagna Disso, Irfan Awan, A. Al-Dhelaan.
The 2nd International Conference on Ambient Systems, Networks and Technologies, 2011.

49. *ntopng*
https://github.com/ntop/ntopng

50. *Trace level*
https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Level.html

51. https://it.wikipedia.org/wiki/Scheda_di_rete

52. *Controlling the design and development cycle.*
João M.P.Cardoso,José Gabriel F.Coutinho, Pedro C.Diniz.
Embedded Computing for High Performance, 2017.

53. *Lowering the latency of data processing pipelines through FPGA based hardware acceleration.*
Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, Pierre-Etienne Melet.
Proceedings of the VLDB Endowment, September 2019.

54. *Difference between Intrusion Detection System (IDS) and Intrusion Prevention System (IPS).*
Asmaa Shaker Ashoor, Sharad Gore.
International Conference on Network Security and Applications, 2011.

55. *A Realistic Experimental Comparison of the Suricata and Snort Intrusion-Detection Systems.*
Eugene Albin, Neil C. Rowe.
26th International Conference on Advanced Information Networking and Applications Workshops, 2012.

56. *Hybrid Intrusion Detection System for DDoS Attacks.*
Özge Cepheli, Saliha Büyükçorak, GüneG Karabulut Kurt.

Journal of Electrical and Computer Engineering, 2016.

57. *Study of intrusion detection system for DDoS attacks in cloud computing.*
Naresh Kumar, Shalini Sharma.
Tenth International Conference on Wireless and Optical Communications Networks, 2013. (WOCN)

58. *Network Intrusion Detection System as a Service in OpenStack Cloud.*
Chen Xu, Ruipeng Zhang, Mengjun Xie, Li Yang.
International Conference on Computing, Networking and Communications (ICNC), 2020.

59. https://www.ntop.org/products/packet-capture/pf_ring/

60. *Verifica e Validazione.*
Carlo Montangero, Laura Semini, Giovanni Cignoni, 2014.

61. https://www.ntop.org/products/netflow/nprobe-cento/

62. https://www.ntop.org/solutions/wire-speed-traffic-generation/

63. *General IDS Acceleration for High-Speed Networks.*
Jan Kucera, Lukas Kekely, Adam Piecek, Jan Korenek.
IEEE 36th International Conference on Computer Design, 2018.

64. *FIXIDS: A High-Speed Signature-based Flow Intrusion Detection System.*
Felix Erlacher and Falko Dressler.
IEEE/IFIP Network Operations and Management Symposium, 2018.

65. *Performance Evaluation of Packet Capturing Systems for High-Speed Networks.*
Fabian Schneider, Jorg Wallerich.
CoNEXT '05: Proceedings of the 2005 ACM conference on Emerging network experiment and technology.

66. https://tools.ietf.org/html/rfc7011

67. https://www.ntop.org/products/netflow/nprobe/

68. https://en.wikipedia.org/wiki/JSON