UNIVERSITÀ DELLA CALABRIA
DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA
DIMES

Corso di Laurea Magistrale in
Ingegneria Informatica

Relazione per il progetto del corso *Network Security*

Analisi di sicurezza in reti SDN con controller POX

Professore
Floriano De Rango

Gruppo
Colantonio Viviana (224473)
Salatino Francesco (227543)
Salerno Giuseppe (224525)

Anno Accademico 2021/2022

Sommario

Introduzione

All'interno delle reti convenzionali, per controllare e monitorare il flusso di dati che attraversa la rete, determinare quanti e quali device sono connessi alla rete, ed anche per definire i path di routing, si utilizzano particolari algoritmi implementati all'interno di device dedicati. Solitamente alla ricezione di un pacchetto da parte di un device di routing, segue l'attuazione di un set di regole definite all'interno del suo firmware, in modo da determinare la destinazione e il path. All'interno di dispostivi più costosi, quali ad esempio i router Cisco, si possono definire tali operazioni anche sulla base della tipologia di pacchetto e del loro contenuto, definendo inoltre delle priorità per i diversi flussi. Uno dei principali problemi di questo approccio è la limitazione dei device di rete rispetto ad una grande mole di traffico, causando una limitazione delle performance dell'intera rete. Infatti, i sistemisti di rete si trovano a dover configurare manualmente protocolli e regole sofisticate all'interno di un contesto completamente eterogeneo, legato alla varietà di tipologie di dispositivi di rete presenti, differenti non solo per caratteristiche ma anche per vendor. Il verificarsi di un errore critico potrebbe quindi portare ad una serie di operazioni che potrebbero richiedere tempistiche di risoluzione abbastanza lunghe da incidere notevolmente sulle performance di rete.

Le reti tradizionali, nate con l'avvento di Internet, sono solitamente costituite da una configurazione gerarchica (originata dal precedente modello di tipo *client-server*, oggi andato via via scemando), dando origine ad una staticità della rete che poco ha a che fare con la dinamicità delle reti odierne. In una rete classica, ad esempio, una modifica apportata ad un suo elemento, potrebbe portare ad una riconfigurazione manuale di molti altri dispositivi, come router, switch, ecc.

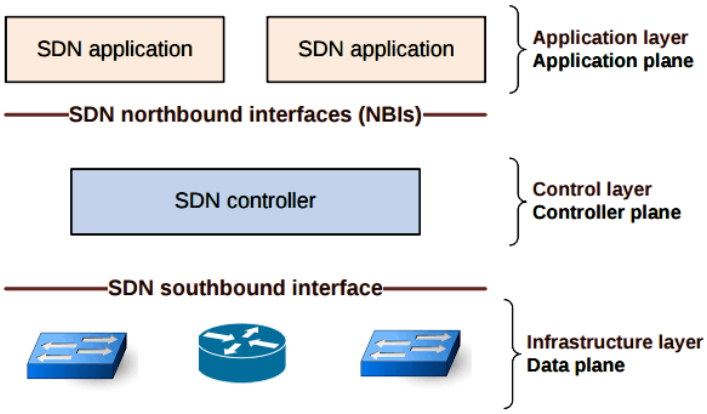
Per tali motivi, negli ultimi anni sono state definiti nuovi modelli di rete che vanno sotto il nome di Software Defined Networking (SDN). Il lavoro progettuale descritto nei paragrafi successivi riguarda in particolare una analisi di sicurezza delle SDN, con proposte di attacchi attuabili e di possibili mitigazioni ad essi.

Per realizzare le analisi dell'idea progettuale è stato utilizzato GNS3[1] come simulatore di rete e, invece, POX [2] come controller della SDN. Si precisa, inoltre, che tutte le topologie di rete prese in considerazione sono sempre dotate di un singolo controller.

1. Funzionamento generale di una SDN

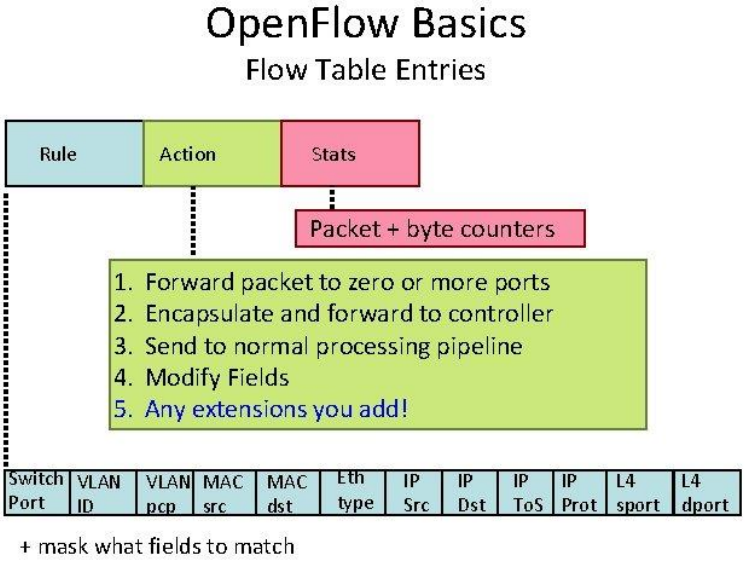
Con Software Defined Networking (SDN) si indica un modello di rete che porta al superamento di molte delle limitazioni delle reti tradizionali, alcune delle quali descritte anche nel paragrafo precedente.

[1] https://www.gns3.com/
[2] https://noxrepo.github.io/pox-doc/html/



Come si evince dall'immagine precedente, all'interno di una SDN il piano di controllo (*control layer*) è separato da quello dei dati (*data plane*). La sostanziale differenza rispetto alle reti tradizionali risiede nel fatto che il controllo del traffico di rete viene gestito all'interno di uno o più calcolatori della rete, anziché su dispositivi di rete interconnessi. I calcolatori della rete dedicati si occupano di amministrare dinamicamente la rete come fosse una entità logica, sfruttando applicazioni specifiche ed altamente programmabili. In un contesto del genere, i dispositivi di commutazione (quali router e switch) diventano semplicemente hardware dedicato all'inoltro dei pacchetti mentre, i calcolatori che fanno parte del *control layer* hanno una visione globale dello stato della rete.

Poiché i router e gli switch sfruttano delle interfacce di controllo sviluppate ad-hoc dalle aziende produttrici, per garantire la comunicazione tra i controller e gli altri dispositivi di rete, si è resa necessaria l'introduzione di una interfaccia di comunicazione generale. Lo standard attuale per le cosiddette *Southbound Interfaces* è il protocollo OpenFlow, altamente programmabile. Il concetto principale sul quale si basa è il concetto di flusso, ovvero specifiche porzioni di traffico con regole precedentemente definite e memorizzate all'interno di tabelle di flusso (*flow table*), con l'obiettivo di generare azioni personalizzate.



All'interno di una SDN, tutte le applicazioni hanno una coscienza globale della rete, e non viceversa. Invece, all'interno di una rete tradizionale manca tale coscienza.

Il controller di rete, sul quale si è intervenuti durante l'attività progettuale che verrà analizzata, rappresenta un piano di controllo logico centralizzato. La sua presenza garantisce una gestione ottimale del traffico di rete, in maniera indipendente dalla posizione dei nodi in essa.

Una ulteriore differenza rispetto alle reti tradizionali risiede nelle operazioni gestite dagli switch. In una rete classica, questi operano a livello di collegamento filtrando, inoltrando ed instradando pacchetti tra host. Le ultime due operazioni sono effettuate sulla base di una tabella di inoltro che contiene indirizzi MAC dei nodi da esso conosciuti. Ogni volta che un pacchetto arriva ad uno switch, questo viene inoltrato sulla interfaccia d'uscita ad esso associata, soltanto se l'indirizzo di destinazione è già presente all'interno della tabella, altrimenti si inoltra il pacchetto su tutte le interfacce d'uscita dello switch, in attesa di una risposta del destinatario per conoscerne e registrare l'interfaccia corretta nella tabella. Tutto ciò può essere riassunto in un controllo decentralizzato della rete, in quanto ogni switch agisce soltanto sulla base della propria tabella ed indipendentemente dal comportamento di altri dispositivi. In una SDN, invece, il controllo logico di uno switch e le decisioni di instradamento sono demandate al controller. La comunicazione con il controller attraverso un canale sicuro avviene quando all'interno di una delle *flow table* presenti nello switch, non vi è nessuna corrispondenza con il pacchetto da processare. Il controller avrà poi pieni poteri sulla gestione del pacchetto, comunicando allo switch se scartarlo, oppure inoltrarlo inserendo una nuova voce nella tabella.

1.1 Protocollo OpenFlow

All'interno di uno switch compatibile con OpenFlow troviamo:

- una o più *flow table*
- una *group table*
- uno o più *OpenFlow channel*

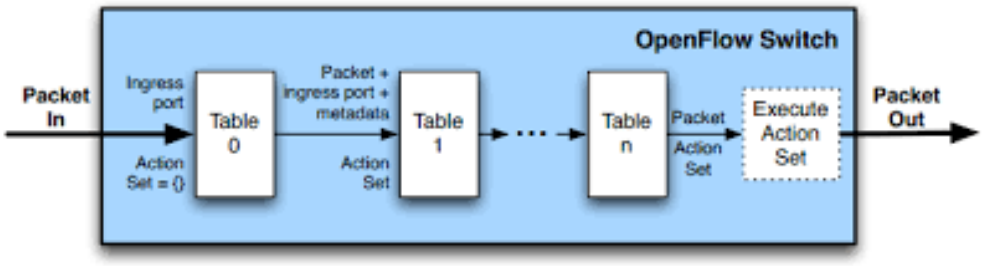
Il protocollo OpenFlow è il mezzo che permette al controller di effettuare modifiche, aggiunte e cancellazioni alle voci delle tabelle presenti negli switch. Tali comunicazioni avvengono attraverso *l'OpenFlow channel*, in modo da garantire lo scambio di messaggi e l'invio o ricezione di pacchetti.

I messaggi OpenFlow scambiati possono essere di tipo:

- *controller-to-switch*, ovvero inviati dal controller e senza la necessità di una risposta da parte dello switch. Tale tipologia di messaggio è solitamente utilizzata per modificare configurazioni dello switch, modificare determinate entry delle tabelle di flusso e inviare ed inoltrare pacchetti ricevuti (*packet-out*)
- *asincroni*, ovvero inviati dagli switch ai controller senza che questo abbia fatto alcuna richiesta. Ricadono in questa categoria tutti i messaggi di *packet-in*, ovvero pacchetti per i quali non esiste una corrispondenza all'interno delle *flow table*. Inoltre, rientrano in questa tipologia di messaggi anche pacchetti che informano il controller relativamente a modifiche di porte dello switch, oppure di rimozioni di voci da una delle tabelle.
- *simmetrici*, ovvero bilaterali. Rientrano nella categoria in esame i pacchetti necessari alla istanziazione della connessione tra controller e switch, oppure relativi alla verifica di connessione ancora attiva.

Ogni volta che uno switch OpenFlow riceve un pacchetto, questo verrà elaborato dalla *pipeline*, ovvero si effettueranno una serie di confronti tra i campi del pacchetto e quelli contenuti nella *flow table*. Poiché, come detto in precedenza, può essere presente più di una tabella di flusso, queste vengono ordinate, in modo da dare priorità al confronto con i campi del pacchetto ricevuto. Se esiste una corrispondenza su un elemento,

la ricerca termina e si eseguono le azioni corrispondenti. Se, al contrario, non esiste alcuna corrispondenza, si associa il pacchetto ad una voce di *table-miss*, per poi decidere se scartarlo oppure inoltrarlo al controller.



(a) Packets are matched against multiple tables in the pipeline

Ogni voce all'interno di una *flow table* sarà costituita dai seguenti valori:

- *Match fields* → sono i valori sfruttati per verificare o meno la corrispondenza del pacchetto in arrivo. Solitamente comprendono:
  - Porta di ingresso
  - Header di pacchetto
  - Eventuali metadati di altre tabelle (se presente un meccanismo di pipeline)
- *Counters* → contatori incrementati soltanto nel caso si verifichi corrispondenza di pacchetto
- *Priority* → valore numerico assegnato in caso di due o più corrispondenze di pacchetto
- *Instructions* → azioni da eseguire in caso di matching di pacchetto
- *Time out* → tempo di permanenza della entry all'interno della tabella, prima che questa venga rimossa
- *Cookie* → valori usati dal controller per ottenere informazioni statistiche
- *Flags* → valori sfruttati per differenziare la gestione di una entry della tabella.

Affinchè le comunicazioni possano avvenire, ogni switch OpenFlow alloca un determinato numero di porte disponibili. Ogni pacchetto si caratterizza di una porta di ingresso dello switch e di una porta di uscita, determinata a seguito del processamento in *pipeline*.

Nella figura sottostante si riporta una regola installata su uno degli switch della rete, a seguito di un ping tra due degli host presenti.

```
/ # ovs-ofctl dump-flows br0
 cookie=0x0, duration=40.829s, table=0, n_packets=40, n_bytes=3920, idle_timeout=10, priority=65535,icmp,in_p
ort=eth1,vlan_tci=0x0000,dl_src=de:ad:be:ef:69:01,dl_dst=ca:fe:ba:be:69:01,nw_src=192.168.0.3,nw_dst=192.168.
0.4,nw_tos=0,icmp_type=8,icmp_code=0 actions=mod_dl_dst:ca:fe:ba:be:69:01,output:eth0
```

2. Modulo l3_learning

Il modulo del controller POX che maggiormente è stato utilizzato per lo sviluppo del progetto è stato *l3_learning*, in quanto implementa le funzioni necessarie per il funzionamento di uno switch al livello 3, in grado di commutare i pacchetti osservando sia il loro indirizzo IP che il loro indirizzo fisico, ovvero il MAC. Questo tipo di switch opera quindi sia a livello di rete che a livello di collegamento dati.

Come descritto all'interno del modulo, le funzionalità implementate consentono, per ogni switch, di:

1. Mantenere una tabella con il mapping tra indirizzi IP,MAC e porte dello switch, popolata sulla base delle informazioni contenute in pacchetti ARP e IP.
2. Rispondere ad ARP query sfruttando le informazioni contenute nella tabella descritta al punto precedente. Ciò avviene nel caso in cui esista una corrispondenza, altrimenti si procede tramite flooding.
3. Installare una regola nello switch ogni volta che si riceve un pacchetto IP del quale si conosce già la porta di destinazione (contenuta nella tabella al punto 1).

Il metodo principale del modulo è *_handle_openflow_PacketIn.* Questo viene richiamato ogni volta che uno switch costruisce un pacchetto di tipo *Packet-in*, nel caso in cui non esista già una regola per l'inoltro del pacchetto stesso. Una volta ricevuto un pacchetto, il controller estrae le seguenti informazioni:

- *dpid*, ovvero il MAC di uno switch;
- *inport*, ovvero la porta di ingresso del pacchetto.

Se il pacchetto è di tipo ipv4, si vanno ad estrarre anche gli indirizzi IP sorgente e destinazione. Nel caso in cui l'indirizzo sorgente sia già presente nella tabella delle corrispondenze costruite per quello switch, si procede con l'aggiornamento della entry ad esso relativa. In caso contrario, si aggiunge direttamente una nuova entry alla tabella. A seguito di queste operazioni si procede con il forwarding del pacchetto attraverso un *Packet-out*. Se l'indirizzo di destinazione è già presente all'interno della tabella, allora il controller conosce già la porta di destinazione del corrispettivo switch e procede con l'installazione di una regola sullo switch, in modo che i prossimi pacchetti contenenti gli stessi indirizzi possano essere inoltrati direttamente al destinatario. Nel caso in cui, invece, il controller non conosca la destinazione, si inserisce per prima cosa il pacchetto in un buffer. Successivamente, invia una *ARP request* per l'indirizzo IP destinazione del pacchetto e, una volta ricevute le informazioni necessarie da una *ARP reply*, si procede come nel caso precedente.

2.1 ARP Poisoning

Sfruttando le vulnerabilità del protocollo ARP, usato per risolvere indirizzi IP in indirizzi MAC, è stato implementato un attacco di ARP poisoning, ovvero un "avvelenamento" delle tabelle ARP di un host.

In generale, anche all'interno delle reti classiche, se un host A volesse contattare un host B, per ottenere il suo indirizzo IP deve prima di tutto individuare l'indirizzo MAC ad esso associato e ciò avviene grazie al protocollo ARP, secondo uno schema *request-response*. L'host A invia prima di tutto una richiesta broadcast, ovvero una *ARP request* a tutti i dispositivi della rete, chiedendo l'indirizzo MAC dell'IP dell'host B. Tutti gli host della rete riceveranno il pacchetto inviato dall'host A, che si aspetterà una risposta tramite *ARP reply* soltanto dall'host B. L'attacco può avvenire anche nel caso in cui l'host A non abbia inviato alcun pacchetto di tipo *ARP request*, in quanto il protocollo ARP non controlla che un pacchetto di tipo *ARP reply* sia collegato ad una particolare richiesta effettuata in precedenza.

Nel contesto di un attacco di tipo ARP poisoning, un attaccante che controlla un host della rete cerca di inviare dei pacchetti di ARP reply contenente informazioni false, e di manipolare così la tabella ARP di un altro host. A seguito dell'avvelenamento, tutti i pacchetti che in realtà erano destinati all'host B, verranno inviati all'host controllato dall'attaccante.

Si noti che, nell'implementazione progettuale, la tabella ARP presa in considerazione non è quella di sistema di un determinato host, ma la tabella costruita all'interno del modulo del controller *l3_learning*, descritta in precedenza.

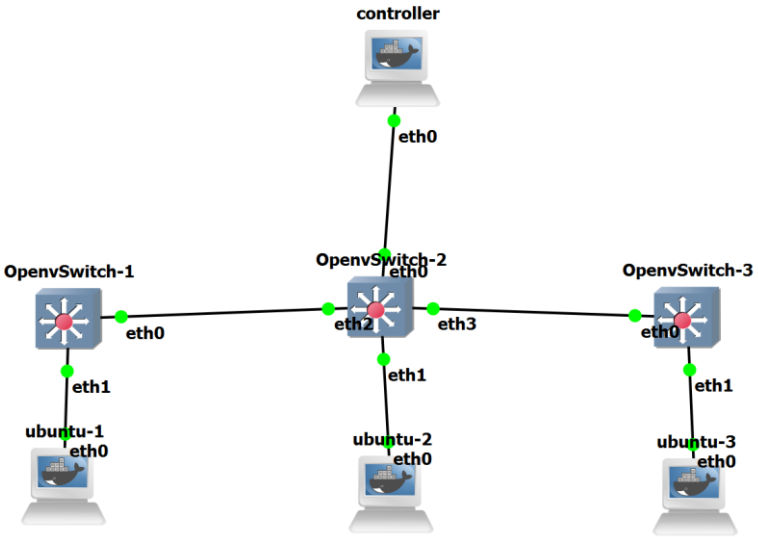
L'attacco utilizzato è stato implementato sfruttando la libreria Scapy, inondando un host della rete forgiando pacchetti di tipo *ARP reply* con i seguenti campi:

- IP sorgente corrispondente a quello dell'host controllato dall'attaccante,
- MAC sorgente fake, ovvero non appartenente ad alcun host all'interno della rete,
- IP destinazione corrispondente ad un host della rete,
- MAC destinazione corrispondente all'host con l'IP destinazione precedente.

In maniera equivalente, come MAC sorgente si potrebbe inserire anche quello dell'host attaccante.

2.2 Esempio di attacco

Supponiamo che, all'interno di una rete costruita come nella figura seguente, l'host denominato *ubuntu-1* voglia effettuare il *poisoning* della tabella ARP dell'host denominato *ubuntu-2.*



Nella figura seguente si riporta la tabella ARP dell'host *ubuntu-2* in due momenti differenti: la prima che contiene le corrispondenze IP-MAC corrette è stata stampata a seguito di un ping tra due host, mentre la seconda a seguito di un attacco. Si nota, quindi, come nel secondo caso il MAC associato all'host *ubuntu-1* corrispondente all'IP 192.168.0.3 non sia quello reale, ma un indirizzo fake scelto dall'attaccante.

```
root@ubuntu-2:/# arp -a
? (192.168.0.3) at de:ad:be:ef:69:01 [ether] on eth0
? (192.168.0.1) at <incomplete> on eth0
root@ubuntu-2:/# arp -a
? (192.168.0.3) at a2:01:c1:3b:4c:1b [ether] on eth0
? (192.168.0.1) at <incomplete> on eth0
```

Il controller POX è stato avviato sfruttando il modulo *l3_learning*.

2.2.1 Mitigazioni proposte

Le mitigazioni ad un attacco di tipo *ARP poisoning* sono state inserite all'interno del modulo *l3_learning*.

In particolare, è stato inserito un metodo denominato *checkArpSpoofing* che, richiamato all'interno del metodo *_handle_openflow_PacketIn* ogni volta che il controller riceve un pacchetto di tipo ARP reply, controlla i dati contenuti nel pacchetto ricevuto e verifica se si tratta di un attacco o meno. L'algoritmo utilizzato per la detection è stato realizzato all'interno di [3], sulla base di una tabella ARP definita staticamente all'interno della classe Python. All'interno di questa struttura dati sono memorizzate le corrispondenze IP-MAC degli host presenti nella rete e, una volta avviato il controller, questa non verrà più modificata. Lo pseudocodice dell'algoritmo è presentato nella figura seguente:

```
If source MAC of Ethernet not like Source MAC of ARP
    Spoofed
Else
    If source MAC-IP addresses mapping in ARP header not found in Main table
        Spoofed
    Else
        If Dest IP of ARP not found in Main Table
            Spoofed
        Else
            If Destination MAC is Broadcast
                If ARP Reply and Dest MAC is Broadcast and Source IP not like Dest IP
                    Spoofed
Else
    Not Spoofed
```

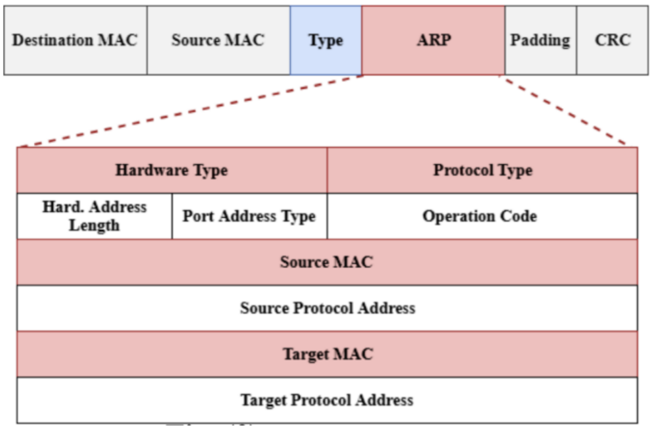
Nel caso in cui il metodo usato per la detection ritorni esito positivo, il metodo *_handle_openflow_PacketIn* viene immediatamente terminato, in modo da non modificare le tabelle ARP contenente i valori corretti definiti all'avvio.

Di seguito si mostra come, a seguito di un attacco il controller sia stato in grado di effettuare la detection appena descritta e di ignorare i pacchetti malevoli, senza inoltrare verso gli switch un pacchetto di tipo *Packet-out*.

[3] *Efficient mechanism for securing software defined network against arp spoofing attack*, HARMAN Y. I. KHALID PARISHAN M. ISMAEL and AHMAD BAHEEJ AL-KHALIL Dept. of Computer Science, College of Science, University of Duhok, Kurdistan Region-Iraq (15 Aprile 2019)

```
root@controller:/pox# python3 pox.py --verbose forwarding.l3_arp_patch
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
DEBUG:core:POX 0.7.0 (gar) going up...
DEBUG:core:Running on CPython (3.8.10/Mar 15 2022 12:22:08)
DEBUG:core:Platform is Linux-5.13.0-30-generic-x86_64-with-glibc2.29
WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[f6-82-57-4c-b9-47 1] connected
INFO:openflow.of_01:[ba-ef-e5-95-e4-4b 2] connected
INFO:openflow.of_01:[06-db-37-ea-e7-4c 3] connected
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 learned 192.168.0.3
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 IP 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 ARP request 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 flooding ARP request 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:271039670827335 3 ARP request 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:271039670827335 3 learned 192.168.0.3
DEBUG:forwarding.l3_arp_patch:271039670827335 3 flooding ARP request 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:7538605745996 1 ARP request 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:7538605745996 1 learned 192.168.0.3
DEBUG:forwarding.l3_arp_patch:7538605745996 1 flooding ARP request 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:271039670827335 2 ARP reply 192.168.0.4 => 192.168.0.3
DEBUG:forwarding.l3_arp_patch:271039670827335 2 learned 192.168.0.4
DEBUG:forwarding.l3_arp_patch:271039670827335 2 flooding ARP reply 192.168.0.4 => 192.168.0.3
DEBUG:forwarding.l3_arp_patch:7538605745996 1 ARP reply 192.168.0.4 => 192.168.0.3
DEBUG:forwarding.l3_arp_patch:7538605745996 1 learned 192.168.0.4
DEBUG:forwarding.l3_arp_patch:7538605745996 1 flooding ARP reply 192.168.0.4 => 192.168.0.3
DEBUG:forwarding.l3_arp_patch:205539511755851 1 ARP reply 192.168.0.4 => 192.168.0.3
DEBUG:forwarding.l3_arp_patch:205539511755851 1 learned 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 1 flooding ARP reply 192.168.0.4 => 192.168.0.3
DEBUG:forwarding.l3_arp_patch:205539511755851 2 ARP request 192.168.0.3 => 192.168.0.4
DEBUG:forwarding.l3_arp_patch:205539511755851 2 answering ARP for 192.168.0.4
Spoofing caso 1
WARNING:forwarding.l3_arp_patch:de:ad:be:ef:69:01->ca:fe:ba:be:69:01 ignorato
Spoofing caso 1
```

In particolare, nell'esempio riportato, l'attacco ricade ne caso in cui gli indirizzi MAC sorgente del livello Ethernet sia differente rispetto a quello a livello ARP. Nello script di attacco, infatti, si è intervenuti nei campi contenuti all'interno del frame ARP, riportato in figura.



2.2.2 Valutazione delle mitigazioni

Evitare che la SDN sia soggetta ad attacchi di ARP poisoning ha sicuramente dei costi.

Poiché le corrispondenze IP-MAC della tabella costruita all'interno del modulo l3_learning sono definite staticamente e prima dell'avvio della rete, viene meno la dinamicità della stessa, in quanto se si aggiungesse un nuovo host, questo verrebbe considerato malevolo ed i pacchetti originati da quest'ultimo non verrebbero inoltrati ad un eventuale destinatario.

L'overhead aggiunto per la soluzione proposta non è eccessivamente elevato, in quanto l'algoritmo di detection non viene richiamato per ogni Packet-in ricevuto, ma soltanto se si tratta di un pacchetto di tipo ARP e, soprattutto, in quanto si è scelto di utilizzare come struttura dati un dizionario con una complessità di *get* di una valore pari ad O(1) nel caso medio. La problematica principale risiede invece nella memoria occupata, in quanto la struttura dati di supporto per la detection contiene una entry per ogni host della rete ed è quindi lineare nella dimensione degli host. Bisogna, però, tener conto che essa è memorizzata in memoria principale, in quanto definita all'interno di un modulo Python.

2.2.3 Flusso di esecuzione di attacco e mitigazioni proposte

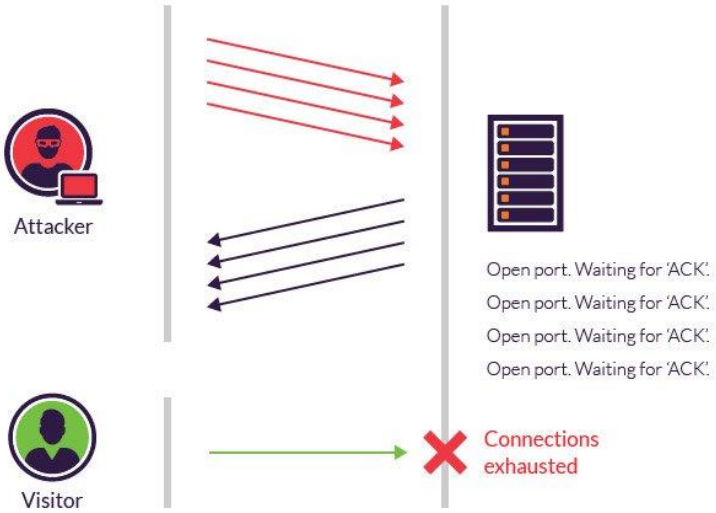
L'attacco può essere lanciato eseguendo lo script Python */attacchi/arp/arp.py* in cui, se eseguito su una rete differente da quella di esempio è possibile modificare i campi relativi ad IP e MAC address sorgente (ovvero dell'attaccante) e della vittima.
Per verificare l'efficacia dell'attacco, invece, il modulo POX da eseguire è *l3_learning* per poi concludere verificando tramite il comando *arp -a* che sull'host vittima la tabella sia stata modificata con i dati fake generati in precedenza.
Infine, l'effetto delle mitigazioni si verifica eseguendo il modulo */pox/forwarding/l3_patch_ARP* e controllando nuovamente che le tabelle ARP della vittima siano rimaste immutate.

2.3 DDOS

Con attacco DDOS (Distributed Denial of Service) si identifica il tentativo da parte di un agente ostile di bloccare il normale traffico di rete di un servizio, una intera rete, un server, ecc., inondando la vittima di traffico Internet anomalo. Rispetto ad un attacco DOS (Denial of Service), in cui la sorgente malevola è costituita da un solo host della rete, nel caso del DDOS si sfruttano come fonti di attacco più sistemi informatici compromessi.

Tra le differenti tipologie di attacco DDOS esistenti in riferimento allo stack ISO/OSI, si è scelto di implementare uno degli attacchi al protocollo, anche noti come attacchi di tipo state-exhaustion che portano, tramite un consumo eccessi di risorse, ad una interruzione del servizio. In particolare, nell'implementazione progettuale, la vittima scelta è stata uno degli host della rete. Poiché, però, se non sono presenti regole di inoltro nello switch i pacchetti passano prima tramite il controller della SDN, un attacco di questo tipo può portare non solo all'esaurimento delle risorse della vittima, ma anche ad un rallentamento delle funzioni svolte dal controller che si trova a dover gestire una quantità enorme di pacchetti anomali.

Tramite il tool *hping* [4] sfruttato per la generazione di pacchetti, si è implementato un attacco di tipo SYN flooding, che sfrutta l'handshake TCP, inviando alla vittima un gran numero di pacchetti SYN di richiesta di connessione iniziale TCP con indirizzi IP di origine contraffatti. La macchina target risponderà alle richieste di connessione e attenderà la conclusione del processo di handshake che, però, non si verificherà mai, giungendo quindi ad un esaurimento delle risorse della vittima.



2.3.1 Mitigazioni proposte

Le mitigazioni per l'attacco DDOS sono state aggiunte all'interno del modulo l3_learning.

Il meccanismo di detection si basa sul costante controllo (in particolare ogni 5 secondi), da parte di un thread dedicato, del numero di pacchetti ricevuti da un determinato IP. Se tale numero ha superato la soglia prefissata, ovvero il massimo numero di pacchetti accettati ogni N secondi, allora l'IP sorgente viene contrassegnato come malevolo e memorizzato all'interno di una apposita struttura dati. Il thread che si occupa della detection effettua i controlli andando a reperire informazioni sul mapping *IP sorgente – numero pacchetti ricevuti* all'interno di un dizionario, opportunamente popolato ed incrementato all'interno del metodo *_handle_openflow_PacketIn* della classe L3, ovvero ogni volta che il controller riceve un nuovo pacchetto.

L'algoritmo utilizzato per bloccare gli host considerati malevoli è basato su una costante posta pari a 3, che indica il numero massimo di segnalazioni raggiungibile da un host, prima che questo venga completamente isolato dalla rete. Tale operazione avviene istruendo gli switch ai quali gli host malevoli sono collegati ad installare delle regole OpenFlow con la lista delle azioni pari ad un array vuoto, in modo che si effettui la drop dei successivi pacchetti, facendo sì che non vengano più inoltrati né verso l'host vittima, né verso il controller. Un ulteriore parametro delle regole sul quale si è agito è la loro durata. In particolare, all'interno di una regola OpenFlow si possono settare due tempi:

- *Idle timeout*, parametro utilizzato per far sì che se la regola non viene matchata entro tale timeout, venga rimossa dallo switch.
- *Hard timeout*, ovvero il tempo di permanenza della regola all'interno dello switch.

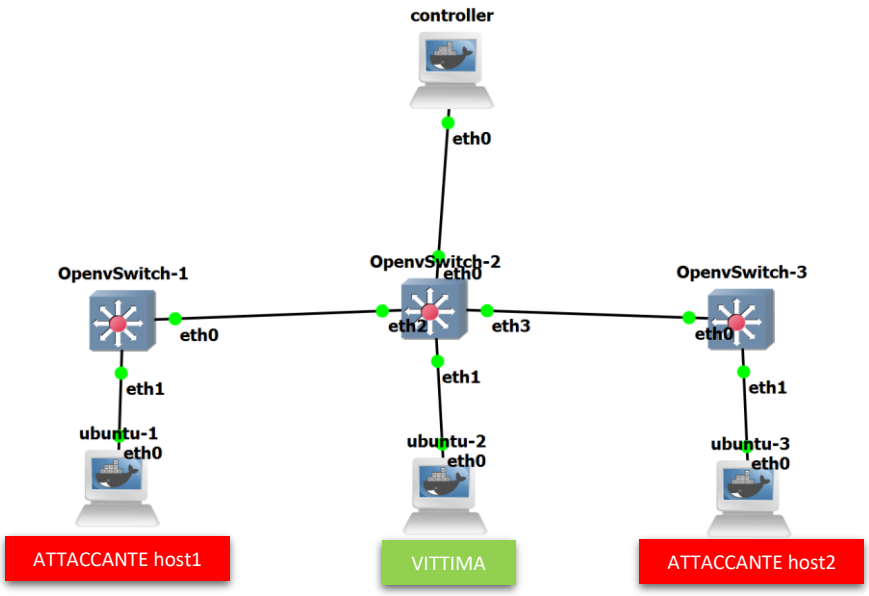
[4] http://www.hping.org/

Si è scelto di settare questi timeout pari a 30 secondi nel caso in cui si ricada in una delle prime tre detection, mentre pari alla costante OFP_FLOW_PERMANENT nel caso in cui l'host malevolo abbia superato il numero massimo di segnalazioni, in modo da rendere permanenti le regole nello switch.

Affinchè un host precedentemente segnalato come malevolo possa uscire dalla blacklist, si è sfruttata una struttura dati per il mapping tra l'indirizzo IP dello stesso, ed il momento in cui è stata inserita la regola nello switch. La struttura, per far si il parametro temporale descritto sia consistente, viene popolata all'interno del metodo *_handle_openflow_PacketIn* ogni volta che si crea un packet-out con una regola di detection temporanea. Successivamente, all'interno dell'algoritmo di detection del thread, si controllerà, nel caso in cui il numero di pacchetti ricevuti da un host non superi la soglia critica, se su questo è ancora presente qualche regola di drop. Il controllo avviene prendendo nuovamente il tempo attuale, sottraendolo a quello presente nella struttura, e verificando che sia maggiore della durata scelta per le regole temporanee. In tal caso, sullo switch non ci sarà nessuna regola di drop, e quindi l'host che non ha superato la soglia critica di pacchetti può essere rimosso dalla blacklist in quanto sta mantenendo un comportamento non malevolo. +

Una ulteriore modifica applicata al modulo *l3_learning* originale riguarda le regole inserite nei *packet_out* nel caso in cui non sia stato rilevato un attacco. Per garantire un conteggio dei pacchetti in arrivo da parte di un host ancora presente nella blacklist il più consistente possibile, si è scelto di non installare regole nello switch ad esso collegato, in modo che tutti i pacchetti inviati arrivino sempre al controller.

Si supponga di considerare la seguente SDN:



```
root@controller:/pox# python3 pox.py forwarding.l3DOSLOCK
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
INFO:openflow.of_01:[f6-82-57-4c-b9-47 1] connected
INFO:openflow.of_01:[06-db-37-ea-e7-4c 2] connected
INFO:openflow.of_01:[ba-ef-e5-95-e4-4b 3] connected
SOGLIA SUPERATA per l'ip: 192.168.0.3
#------------------DDOS DETECTED: TEMPORARY RULE per host: 192.168.0.3 -----------------
SOGLIA NON SUPERATA per l'ip: 192.168.0.3
SOGLIA NON SUPERATA per l'ip: 192.168.0.5
SOGLIA SUPERATA per l'ip: 192.168.0.5
#------------------DDOS DETECTED: TEMPORARY RULE per host: 192.168.0.5 -----------------
SOGLIA NON SUPERATA per l'ip: 192.168.0.5
SOGLIA NON SUPERATA per l'ip: 192.168.0.5
SOGLIA SUPERATA per l'ip: 192.168.0.3
Incremento le rilevazioni per l'ip: 192.168.0.3
#------------------DDOS DETECTED: TEMPORARY RULE per host: 192.168.0.3 -----------------
SOGLIA NON SUPERATA per l'ip: 192.168.0.3
SOGLIA SUPERATA per l'ip: 192.168.0.5
Incremento le rilevazioni per l'ip: 192.168.0.5
#------------------DDOS DETECTED: TEMPORARY RULE per host: 192.168.0.5 -----------------
SOGLIA NON SUPERATA per l'ip: 192.168.0.5
SOGLIA SUPERATA per l'ip: 192.168.0.3
Incremento le rilevazioni per l'ip: 192.168.0.3
#-------------------DDOS DETECTED: PERMANENT RULE per host: 192.168.0.3 -----------------
SOGLIA NON SUPERATA per l'ip: 192.168.0.3
SOGLIA SUPERATA per l'ip: 192.168.0.5
Incremento le rilevazioni per l'ip: 192.168.0.5
#-------------------DDOS DETECTED: PERMANENT RULE per host: 192.168.0.5 -----------------
```

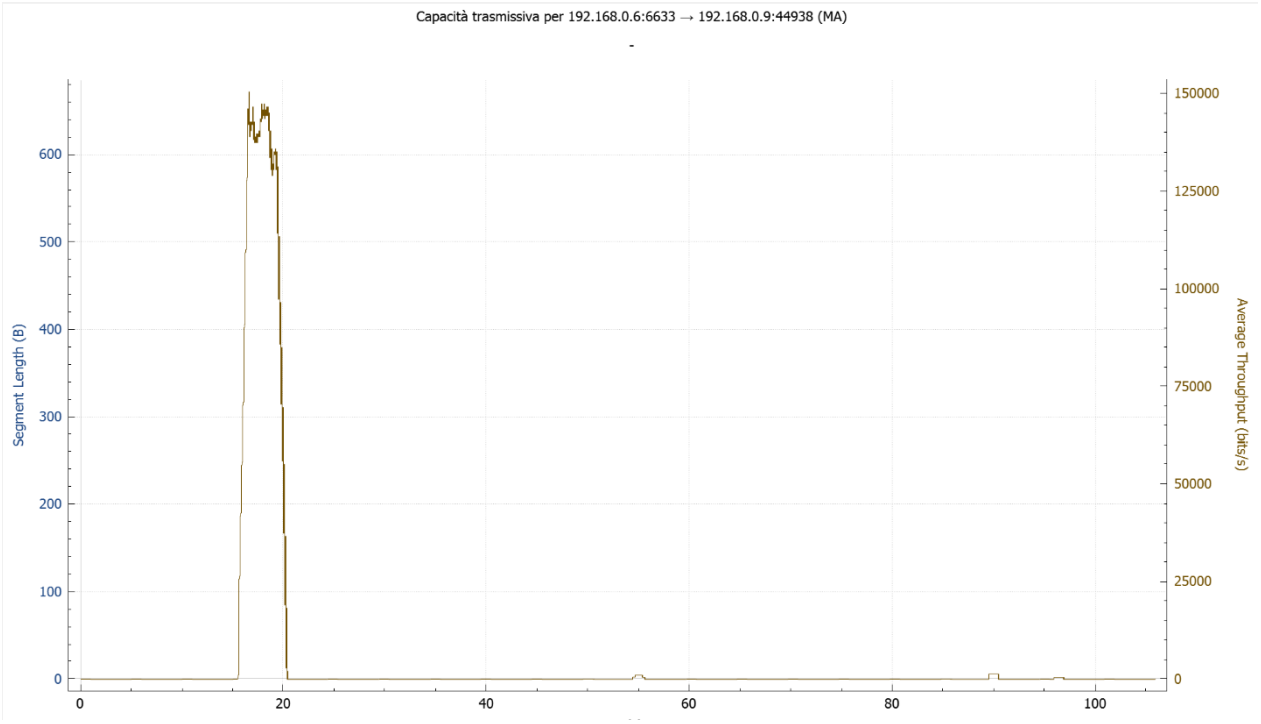
Detection 1 per host1

Detection 1 per host3

Detection 2 per host1

Detection 2 per host3

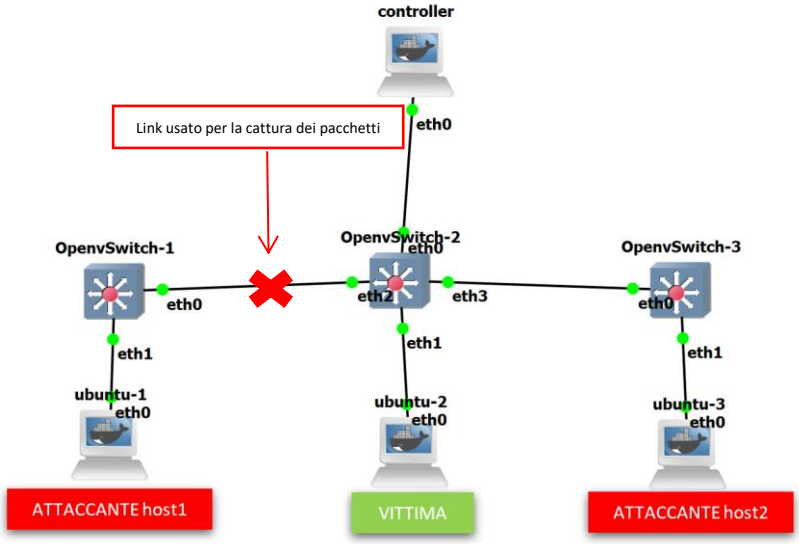
Per gli esperimenti effettuati, si è scelta una soglia pari a 500 pacchetti ed un tempo di sleep del thread dedicato al controllo pari a 5 secondi. Questo implica che il numero massimo di pacchetti accettati affinchè non venga rilevato un attacco è pari a 100. Grazie al tool Wireshark è stato possibile realizzare i seguenti grafici:



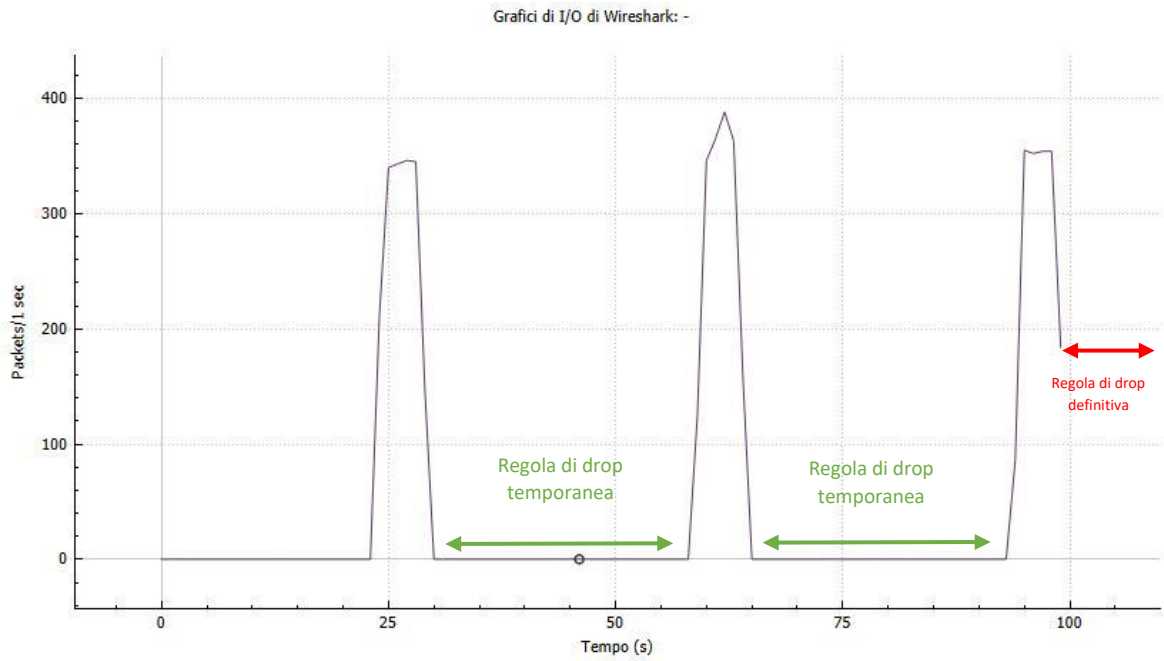
In questa prima rilevazione, effettuata sul link che connette il controller allo switch della topologia di rete precedente (filtrando i pacchetti TCP con flag SYN), si nota come sia presente un picco di 150000 bit/s. Poiché la dimensione di un pacchetto TCP è pari a 1104 bit, ciò equivale a dire che in quell'istante, nella rete sono

stati trasmessi circa 135 pacchetti verso il controller. Quindi, rispetto alla soglia fissata a 100 pacchetti, l'attacco è stato rilevato correttamente.

Si consideri ora di catturare i pacchetti nel link evidenziato nella topologia di seguito:



Utilizzando il modulo con le mitigazioni all'interno del controller, e filtrando i pacchetti che hanno come IP quello dell'host descritto come "ATTACCANTE host1", è stato definito il grafico seguente, che evidenzia come i pacchetti non vengano più trasmessi nei 30 secondi successivi all'installazione di una regola di drop temporanea, e come le comunicazioni vengano totalmente interrotte a seguito della regola di drop permanente.



Per misurare le prestazioni delle mitigazioni utilizzate, è stata misurata la latenza di rete, ottenuta tramite uno script Python che sfrutta la media del tempo di risposta della vittima a pacchetti ICMP. La latenza media è stata calcolata nei seguenti casi:

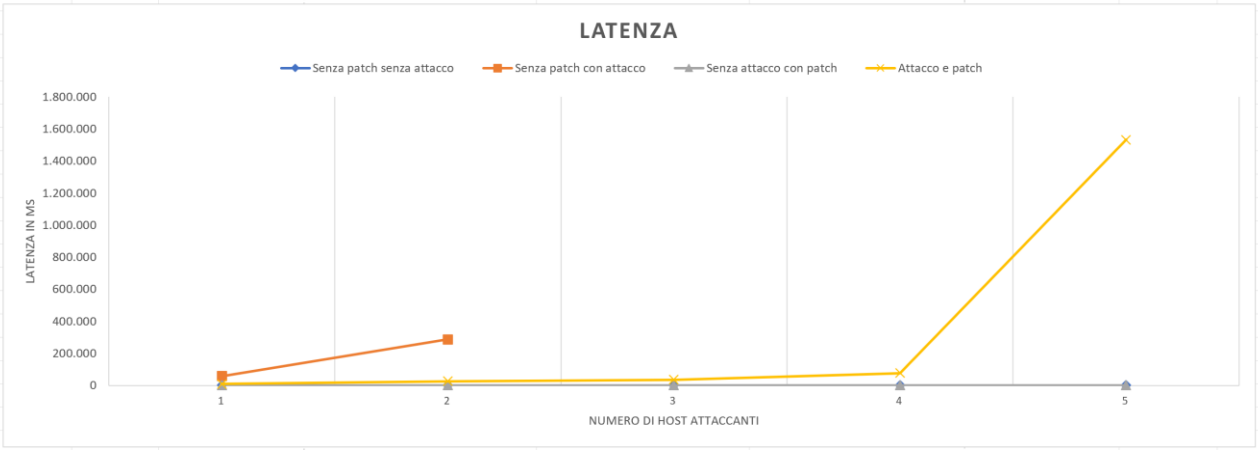
- lanciando il modulo *l3_learning* senza nessun attacco in corso,
- lanciando il modulo *l3_learning* con attacco in corso,

- lanciando il modulo *l3_learning* con mitigazioni attive senza che sia presente alcun attacco,
- lanciando il modulo *l3_learning* con mitigazioni attive ed attacco in corso.

Nelle rilevazioni effettuate con un attacco DDOS in corso, in particolare, la latenza è stata misurata con un numero di host malevoli attivi che va da 1 fino ad un massimo di 5.

Dai risultati ottenuti, si può notare come la latenza di rete nel caso in cui si utilizzi il modulo con le mitigazioni attive aumenta di circa 0.2 ms. Pertanto, la soluzione realizzata non introduce quasi alcun overhead temporale. Interessanti sono anche i valori ottenuti nel caso in cui il controller venga attaccato da più di due host contemporaneamente: se si utilizza il modulo *l3_learning* originale, la vittima dell'attacco non riesce più a rispondere alle richieste e pertanto l'attacco si può ritenere concluso con successo; al contrario, invece, se il controller utilizza la classe con le mitigazioni attive, l'host vittima risulta ancora essere raggiungibile e i valori di latenza rimangono più bassi di quelli registrati nel caso precedente, ma con un numero di attaccanti minore.

Numero host	Senza patch senza attacco	Senza patch con attacco	Senza attacco con patch	Attacco e patch
1	1.348	59.249	1.582	9.976
2	1.348	287.105	1.582	26.387
3	1.348		1.582	36.654
4	1.348		1.582	77.566
5	1.348		1.582	1.533.028



2.3.2 Valutazione delle mitigazioni

L'overhead aggiunto dalla mitigazione presa in esame è sicuramente maggiore rispetto a quello introdotto nel caso di un attacco di tipo ARP poisoning. Infatti, mentre nel caso precedente il meccanismo di detection entra in gioco soltanto nel caso in cui il controller riceva dei pacchetti ARP, nel caso del DDOS si interviene effettuando il checking per ogni pacchetto di tipo IPV4 ricevuto dal controller, il cui numero totale è sicuramente più elevato rispetto all'altra tipologia di pacchetto.

Inoltre, il principale overhead di memoria è causato dalla presenza del dizionario contenente il mapping *IP sorgente – numero pacchetti ricevuti* che, nel caso di reti molto complesse, potrebbe raggiungere dimensioni elevate. Lo stesso si può dire per le altre strutture dati in cui il numero di entry è pari al più al numero di host presenti nella rete.

2.3.3 Flusso di esecuzione di attacco e mitigazioni proposte

L’attacco può essere lanciato eseguendo *hping3 -S --flood -V <ip_host_vittima>* all’interno di un certo numero di host scelti come attaccanti.
Per verificare l’efficacia dell’attacco, invece, il modulo POX da eseguire è *l3_learning* per poi concludere verificando tramite *ping <ip_host_vittima>* che i tempi di risposta siano molto elevati, o addirittura che non si riesca ad ottenere alcun tipo di risposta.
Infine, l’effetto delle mitigazioni si verifica eseguendo il modulo */pox/forwarding/l3_patch_dos* e controllando nuovamente i parametri precedenti.

3. LLDP

Il protocollo LLDP (Link Layer Discovery Protocol), formalmente definito nello standard **IEEE 802.1AB,** consente di identificare le risorse presenti in una rete LAN, ricostruendone la topologia e illustrando in che modo i dati fluiscono nella rete.

I pacchetti LLDP viaggiano al livello 2 (Data Link) della pila protocollare ISO/OSI e obbligatoriamente devono contenere le seguenti informazioni, memorizzate in strutture dati denominate TLV (Type-Length-Value): *Chassis ID (MAC Address), Port ID e Time-to-live*.

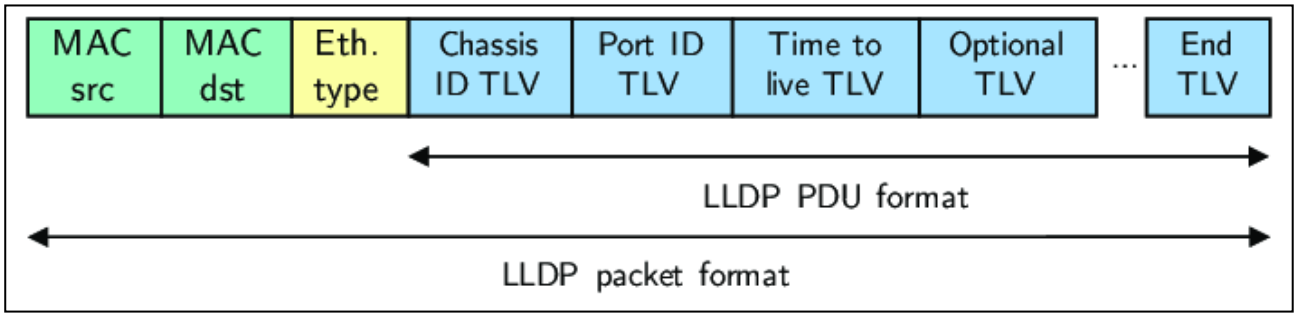


Figura 1: Struttura di un frame LLDP

La tecnologia SDN (Software-Defined-Network) utilizza il protocollo LLDP oltre che per definire la topologia di rete, anche per monitorare in tempo reale la latenza sulla rete e i nodi disponibili, consentendo di non sovraccaricare la rete con pacchetti OpenFlow che risultano computazionalmente più impegnativi da elaborare rispetto ai frame LLDP. Il funzionamento del protocollo LLDP in una SDN è schematizzato in *Figura 2*.

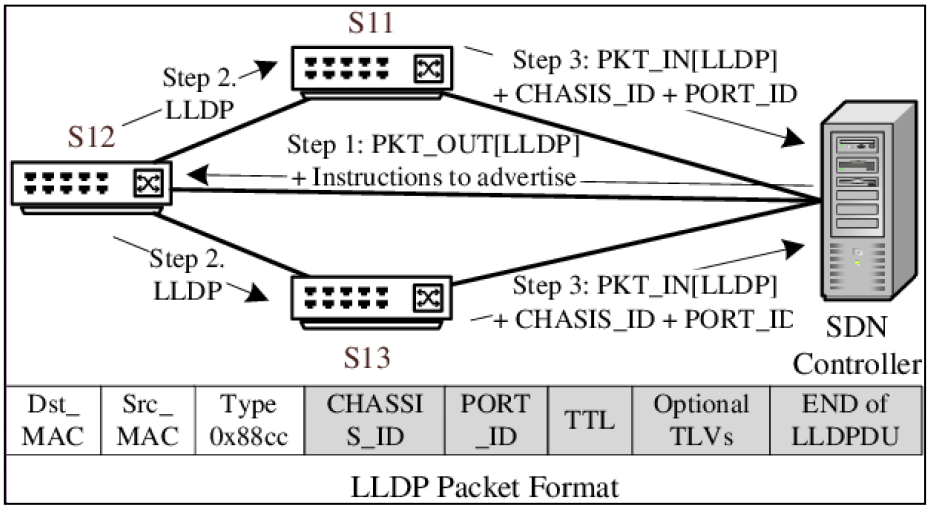


Figura 2: Il Controller invia frame LLDP per definire la topologia di rete

I frame LLDP inviati dal controller attraversano la rete, e in funzione della porta da cui tornano al mittente può definire la topologia della rete, la latenza, e di conseguenza aggiornare i percorsi di instradamento dei pacchetti.

3.1 Caso di studio

Con l'ausilio del simulatore di rete **GNS3** è stata definita la SDN di *Figura 3*. Il software utilizzato per la funzione di Controller è *POX*.

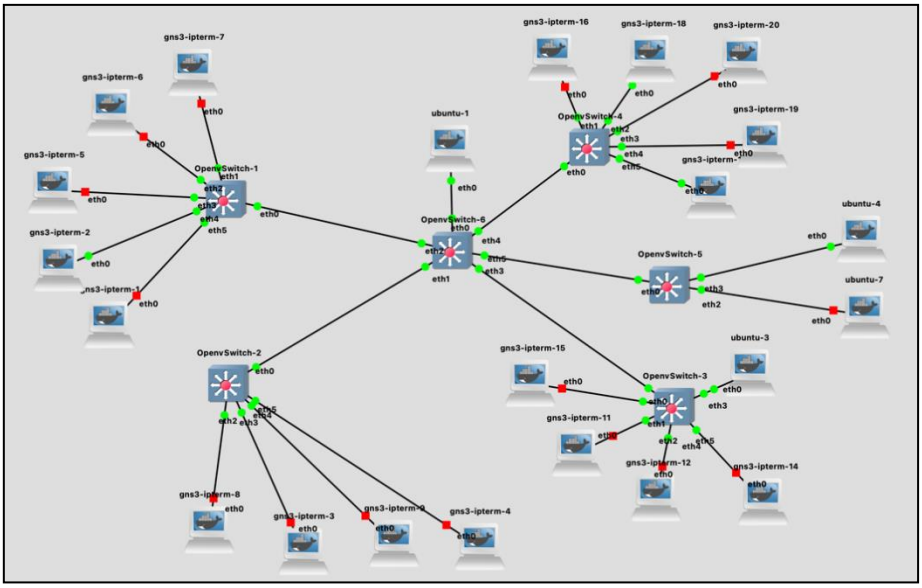


Figura 3: Simulazione di una SDN con Controller centralizzato

Nella rete sono presenti 6 OVS (Open vSwitch) registrati con i seguenti MAC Address:

OVS HOSTNAME	OVS MAC ADDRESS
OPENVSWITCH-1	E6:09:4F:F5:96:4B
OPENVSWITCH-2	16:CC:70:20:64:4D
OPENVSWITCH-3	86:43:D5:3E:D8:44
OPENVSWITCH-4	42:1D:D3:12:07:40
OPENVSWITCH-5	46:A4:A6:AE:BE:49
OPENVSWITCH-6	9E:68:9F:4F:2D:44

Il Controller *POX* dispone del modulo *Discover.py* che permette di ricostruire la topologia della rete con l'ausilio del protocollo LLDP. Questa componente invia messaggi LLDP appositamente predisposti per gli OVS: quando viene rilevato un collegamento si genera un evento e il Controller memorizza le informazioni ricevute. Il processo di discovery, di default, è eseguito ogni 5 minuti.

Essendo LLDP un protocollo di livello 2 della pila IOS/OSI, la rilevazione degli OVS avviene registrando i rispettivi MAC Address. La struttura dati che gestisce i fra me si compone come segue:

NOME	VALORE
DPID1	MAC Address di uno degli OVS coinvolti nel collegamento
PORT1	Porta dell'OVS con dpid1
DPID2	MAC Address di uno degli OVS coinvolti nel collegamento
PORT2	Porta dell'OVS con dpid2
UNI	Indica il verso del collegamento
END[0 OR 1]	Terminale del collegamento, es. end[0]=(dpid1,port1)

Sono essenziali, del modulo Discovery.py i metodi *create_discovery_packet()* e *_handle_openflow_PacketIn()* che si occupano rispettivamente di forgiare i pacchetti LLDP e di analizzarli per campi dopo averli ricevuti. Si è lavorato in questa sezione al fine di mitigare lo scenario d'attacco di seguito proposto.

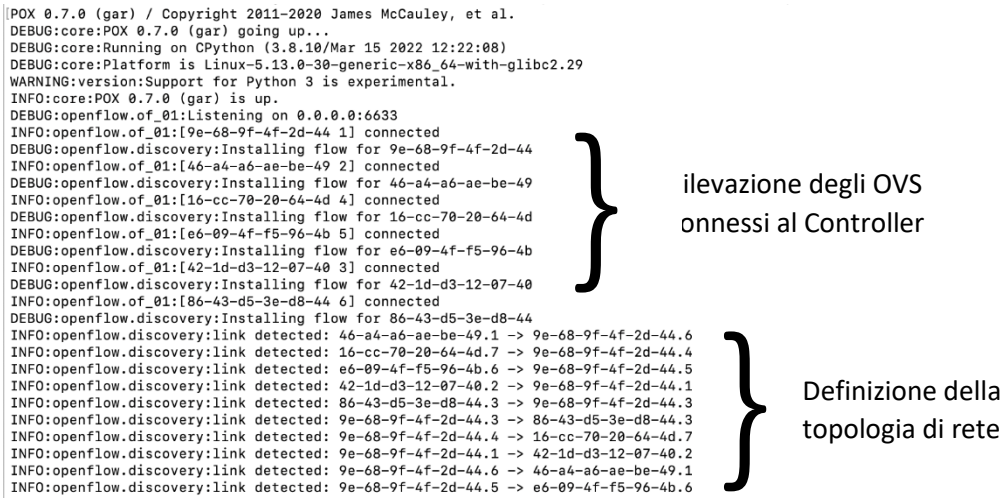


Figura 4: Esecuzione del modulo *Discovery.py*

3.2 Scenario d'attacco

In questo scenario d'attacco l'obiettivo è quello di violare il protocollo LLDP, generando false informazioni sulla topologia di rete che verranno memorizzate ed elaborate dal Controller di rete.

Lo scopo dell'attacco è ingannare il Controller, generando frame LLDP che immedesimino un link, in realtà inesistente, tra OpenvSwitch-3 e OpenvSwitch-5. Si suppone che l'attaccante abbia il controllo dell'host *Ubuntu3*.

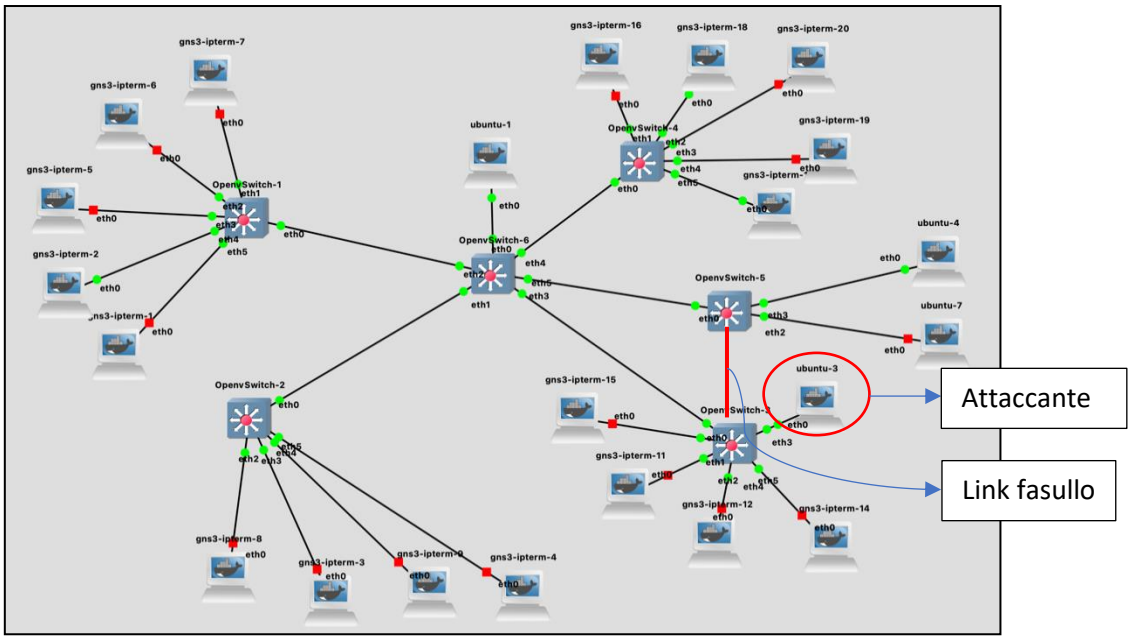


Figura 5: Scenario d'attacco

Il primo passo dell'attacco consiste nel catturare il traffico di rete e isolare i pacchetti LLDP: a tale scopo è stato utilizzato Scapy, un framework Python dedicato al networking. Analizzando il frame, si può notare il campo *dpid*, contente il MAC Address dello switch che lo ha inoltrato (OpenvSwitch-3).

```
>>> lldp_packet
<Ether  dst=01:23:20:00:00:01 src=8a:d6:d8:cc:ff:f0 type=LLDP |<Raw  load='\x02\x12\x07dpid:8643d53ed844\x04\x02\x026\x06\x02\x00x\x0c\x
1dpid:8643d53ed844\x00\x00' |>>
```

Figura 6: Frame LLDP catturato con Scapy

Successivamente si modifica il frame, inserendo nel campo dpid il MAC Address di OpenvSwitch-5.

```
>>> crafted_lldp_packet
<Ether  dst=01:23:20:00:00:01 src=8a:d6:d8:cc:ff:f0 type=LLDP |<Raw  load='\x02\x12\x07dpid:46a4a6aebe49\x04\x02\x026\x06\x02\x00x\x0c\x
1dpid:46a4a6aebe49\x00\x00' |>>
```

Figura 7: Frame LLDP modificato

Ritrasmettendo il frame sulla rete si ottiene l'effetto sperato: il Controller crede che sia attivo un link tra OpenvSwitch-3 e OpenvSwtich-5.

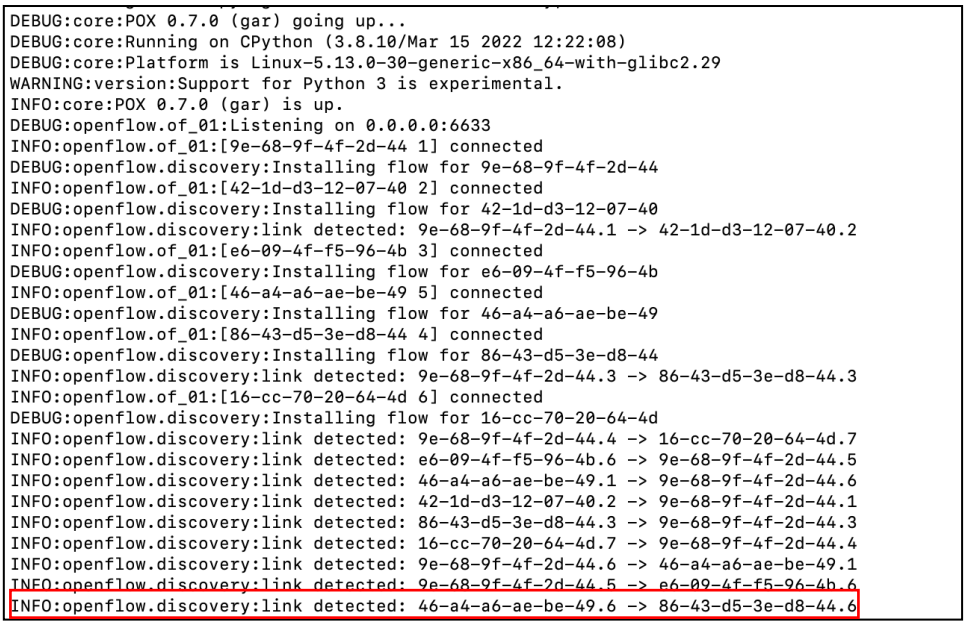


Figura 8: Nuovo link (inesistente) individuato dal Controller

3.3 Mitigazioni proposte

Il principale problema del protocollo LLDP, nella sua forma originale, risiede nell'assenza di meccanismi di sicurezza. Si è pensato dunque, di implementare un controllo di integrità dei frame LLDP inviati sulla rete, al fine di scongiurare attacchi che ne modifichino il contenuto.

La modalità scelta, fa uso di HMAC con funzione hash SHA-256:

$$\mathrm{HMAC}_K(M) = H\Big((K' \oplus \mathrm{opad}) \| H\big((K' \oplus \mathrm{ipad}) \| M'\big)\Big)$$

La chiave K, di dimensione pari a 16 bytes, generata in modo pseudocasuale, viene aggiornata ogni volta che il Controller inizia un nuovo ciclo di discovery sulla rete. Il messaggio M è formato dalla concatenazione di *chassis_id, dpid e ttl*: M= CHASSIS_ID|DPID|TTL.

Il frame LLDP a questo punto conterrà il tag generato con HMAC.

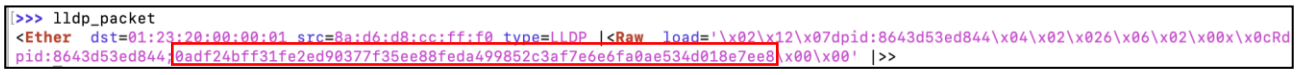


Figura 9: Frame LLDP contenente il tag generato da HMAC

Se l'attaccante provasse ad effettuare l'attacco mostrato prima non avrebbe alcun effetto, in quanto non sarebbe verificata l'integrità del frame LLDP.

La mitigazione è stata applicata come patch del modulo *Discovery.py,* integrando le funzioni di crittografia necessarie.

```
1 _hmac_key = secrets.token_hex(16) #Generazione chiave pseudocasuale
2
3 def hmac_encryption(port_id, chassis_id, ttl, key):
4     msg = port_id + chassis_id + str(ttl).encode()
5     h = hmac.new(key.encode(), msg, hashlib.sha256)
6     return h.hexdigest()
```

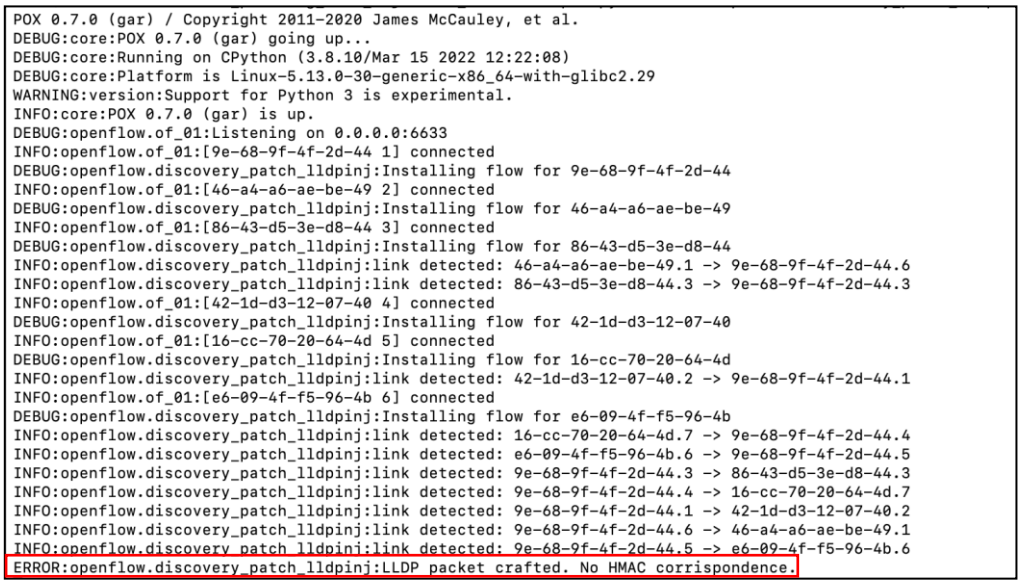


Figura 10: Rilevazione di un pacchetto LLDP modificato

3.4 Analisi delle prestazioni

Le analisi effettuate sulla rete di *Figura 3* non rilevano un aumento dei tempi necessari ad elaborare i frame LLDP ricevuti dal Controller. Sono stati eseguiti test con 3,5 e 6 OVS attivi, confrontando le prestazioni del modulo Discovery.py originale con quello contenente la mitigazione basta su HMAC.

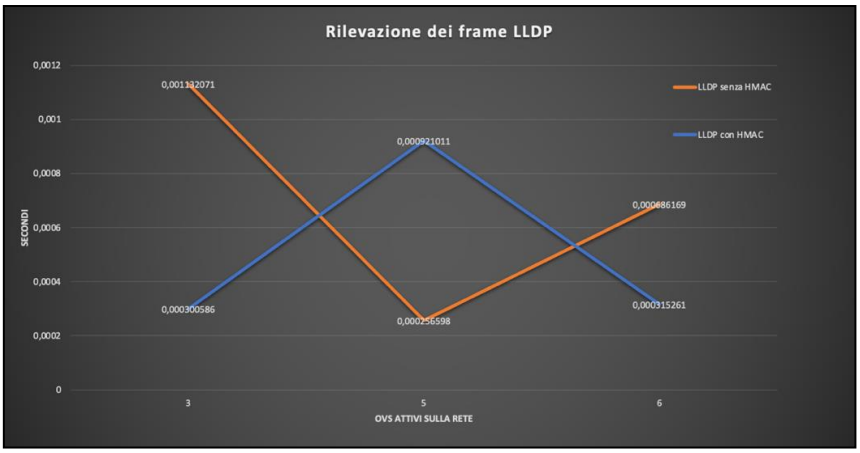


Figura 11: Analisi delle prestazioni del modulo Discovery.py in fase di ricezione dei frame LLDP

Per quanto concerne la generazione dei frame LLDP si è registrato, mediamente, un incremento di 6 ms nella fase di creazione e invio sulla rete.

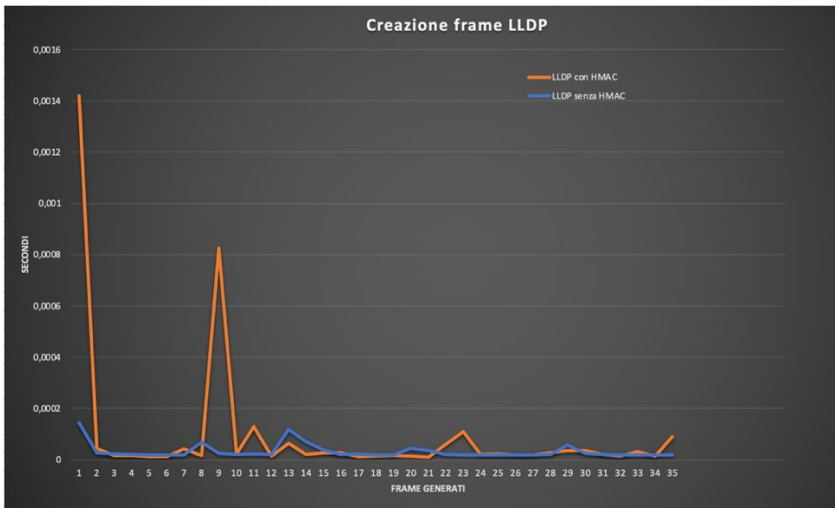


Figura 12: Analisi delle prestazioni in fase di generazione dei frame LLDP

3.5 Flusso di esecuzione di attacco e mitigazioni proposte

L'attacco può essere lanciato eseguendo lo script Python */attacchi/LLDPInjection/lldp_attack.py* all'interno di un host scelto come malevolo.
Per verificare l'efficacia dell'attacco, invece, il modulo POX da eseguire è */openflow/discovery.py* per poi concludere controllando la rilevazione di link inesistenti tra switch. Dal terminale del controller:

python3 pox.py openflow.discovery

Infine, l'effetto delle mitigazioni si verifica eseguendo il modulo */pox/openflow/ discovery_patch_lldpinj.py* e controllando che, questa volta, i pacchetti inviati dall'attaccante non vengano riconosciuti come integri, a causa di errori nel matching dell'HMAC.

4. Host Tracker Service

Nel contesto di una Software-Defined Networking, il modulo "Host Tracker Service" si occupa di mantenere informazioni e meta-dati sugli host: configurazione degli indirizzi IP e MAC e posizione degli host nella rete, monitorando i messaggi di packet-In che il controller riceve dagli switch.

Host Tracker è un servizio di monitoring, infatti non inoltra nessun pacchetto ma estrae solo informazioni (indirizzo MAC/IP, porta, locazione) che vengono usate per aggiornare una tabella che dovrebbe descrivere fedelmente lo stato corrente degli host nella rete.

Ad esempio, se un host cambia la sua posizione e invia un pacchetto da quest'ultima, HTS può riconoscere tale migrazione ricevendo un messaggio Packet-In che incapsula il pacchetto e aggiorna le rispettive tabelle.

Tuttavia, HTS potrebbe non supportare meccanismi di autenticazione dell'host. Pertanto, un utente malintenzionato potrebbe facilmente comprometterne il funzionamento.

Il modulo host_tracker.py di POX offre proprio questo servizio. Tiene traccia degli host nella rete tramite la classe *MacEntry*, che memorizza per ognuno di essi una tripla: $< dpid, port, macaddress >$. Dove *dpid* e *port* sono rispettivamente l'identificatore e la porta dello switch a cui è collegato l'host, questa coppia identifica univocamente la posizione di un host nella rete.

Il metodo principale del modulo è *_handle_PacketIn.* Questo viene richiamato ogni volta che uno switch costruisce un pacchetto di tipo *Packet-in*, nel caso in cui non esista già una regola per l'inoltro del pacchetto stesso. Il servizio di host tracking usa i metadati del pacchetto per estrarre informazioni (*dpid, port, macaddress*) e aggiornare la relativa entry o aggiungerne una nuova nel caso in cui non ne esistesse una avente il *macaddress* appena estratto. In sostanza per ogni nuovo indirizzo *MAC* viene creata una entry che identifica l'esatta posizione dell'host nella rete, mentre per ogni indirizzo *MAC* già presente nella tabella, se il *PacketIn* presenta *dpid e/o port* differenti rispetto a quelli memorizzati, viene eseguita una *moved* del relativo host nella nuova posizione.

```
INFO:host_tracker:Learned 271039670827335 2 ca:fe:ba:be:69:01
INFO:host_tracker:Learned 271039670827335 2 ca:fe:ba:be:69:01 got IP 192.168.0.4
```

Per mantenere la struttura dati sempre in uno stato consistente, il servizio di host tracking, fa uso di un meccanismo di controllo periodico dello stato della rete. In particolare, tramite il metodo *__check_timeouts* si controlla che le entry presenti nella struttura dati siano ancora valide e quindi non ci siano host scaduti. Nel caso in cui un host non sia più attivo, la relativa porta a cui era collegato viene liberata.

4.1 Scenari di attacco

Lo studio del modulo host_tracker.py in POX ha l'obiettivo di analizzare le vulnerabilità del servizio di Host Tracking quando nella SDN sono presenti utenti malintenzionati.

Precedentemente è stato accennato il funzionamento del metodo *__handle_PacketIn*, questo è suscettibile a diversi attacchi poiché non esegue nessun tipo di controllo sulla consistenza dei pacchetti che riceve. In particolar modo se da un pacchetto estrae metadati *(dpid,port)* che corrispondono ad una porta di uno switch

già occupata, il modulo non se ne accorge ed esegue normalmente l'aggiunta di un nuovo host (o una moved se l'host era già presente nella rete).

Sono stati analizzati i seguenti scenari d'attacco:

- *Fake host generation:* un host malevolo può creare pacchetti forgiando i campi IP sorgente e MAC sorgente in maniera random e inviarli al controller (o ad un altro host).

```
def new_fake_host():
    #RANDOM SRC MAC, RANDOM SRC IP
    p = Ether(src='bb:bb:bb:bb:69:01', dst='0a:87:9c:b8:df:29')/IP(src='192.168.0.70', dst='192.168.0.6')
    sendp(p)
    time.sleep(1.5)
```

  Il servizio di host tracking rileverà l'aggiunta di un nuovo host alla rete sulla porta dello switch occupata dall'host attaccante!

```
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
INFO:host_tracker:host_tracker ready
WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
INFO:openflow.of_01:[9e-68-9f-4f-2d-44 1] connected
INFO:host_tracker:Learned 174172186553668 3 6e:5c:9b:01:e6:8a
INFO:openflow.of_01:[86-43-d5-3e-d8-44 2] connected
INFO:openflow.of_01:[42-1d-d3-12-07-40 3] connected
INFO:openflow.of_01:[46-a4-a6-ae-be-49 4] connected
INFO:openflow.discovery:link detected: 86-43-d5-3e-d8-44.3 -> 9e-68-9f-4f-2d-44.3
INFO:openflow.of_01:[e6-09-4f-f5-96-4b 6] connected
INFO:openflow.discovery:link detected: 46-a4-a6-ae-be-49.1 -> 9e-68-9f-4f-2d-44.6
INFO:openflow.of_01:[16-cc-70-20-64-4d 5] connected
INFO:openflow.discovery:link detected: e6-09-4f-f5-96-4b.6 -> 9e-68-9f-4f-2d-44.5
INFO:openflow.discovery:link detected: 42-1d-d3-12-07-40.2 -> 9e-68-9f-4f-2d-44.1
INFO:openflow.discovery:link detected: 16-cc-70-20-64-4d.7 -> 9e-68-9f-4f-2d-44.4
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.3 -> 86-43-d5-3e-d8-44.3
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.4 -> 16-cc-70-20-64-4d.7
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.1 -> 42-1d-d3-12-07-40.2
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.6 -> 46-a4-a6-ae-be-49.1
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.5 -> e6-09-4f-f5-96-4b.6
INFO:host_tracker:Learned 174172186553668 2 66:05:ea:b7:ce:77
INFO:host_tracker:Learned 174172186553668 2 66:05:ea:b7:ce:77 got IP 192.168.0.100
INFO:host_tracker:Learned 147625898596420 1 fe:7b:e3:e2:50:9a
INFO:host_tracker:Learned 147625898596420 6 ba:54:67:3a:ed:69
INFO:host_tracker:Entry 174172186553668 3 6e:5c:9b:01:e6:8a expired
INFO:host_tracker:Learned 252927670589003 3 fe:be:f4:a7:da:3a
INFO:host_tracker:Learned 252927670589003 3 fe:be:f4:a7:da:3a got IP 192.168.0.14
INFO:host_tracker:Entry 147625898596420 1 fe:7b:e3:e2:50:9a expired
INFO:host_tracker:Entry 147625898596420 6 ba:54:67:3a:ed:69 expired
INFO:host_tracker:Learned 147625898596420 4 e2:91:a5:63:37:a7
INFO:host_tracker:Learned 147625898596420 4 e2:91:a5:63:37:a7 got IP 192.168.0.200
INFO:host_tracker:Learned 147625898596420 4 bb:bb:bb:bb:69:01
INFO:host_tracker:Learned 147625898596420 4 bb:bb:bb:bb:69:01 got IP 192.168.0.140
INFO:host_tracker:Learned 252927670589003 65534 e6:09:4f:f5:96:4b
```

  Nel caso estremo, questo attacco può portare anche ad un Denial of Service, infatti si potrebbe sovraccaricare il controller inondandolo di pacchetti e facendogli credere che stiano entrando nella rete un numero indefinito di host.
- *Host impersonation attack:* un host malevolo può creare pacchetti settando i campi IP sorgente e MAC sorgente con quelli di un altro host già presente nella SDN sulla porta X e inviarli al controller (o ad un altro host).

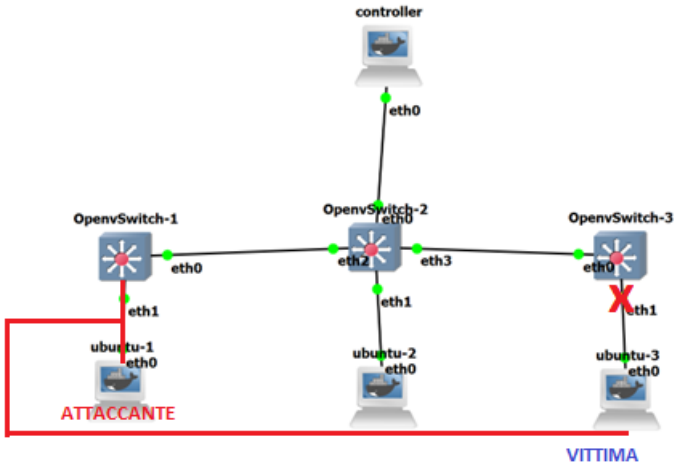
```
def poisoning_porta():
    #HOST TARGET: {ip: 192.168.0.4, mac: ca:fe:ba:be:69:01}
    p = Ether(src='ca:fe:ba:be:69:01', dst='0a:87:9c:b8:df:29')/IP(src='192.168.0.4', dst='192.168.0.6')
    sendp(p)
    time.sleep(1.5)
```

Il servizio di host tracking rileverà lo spostamento dell'host target dalla porta X alla porta Y aggiornando la rispettiva entry nella tabella.

```
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
INFO:host_tracker:host_tracker ready
WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
INFO:openflow.of_01:[9e-68-9f-4f-2d-44 1] connected
INFO:openflow.of_01:[42-1d-d3-12-07-40 2] connected
INFO:openflow.of_01:[46-a4-a6-ae-be-49 3] connected
INFO:openflow.of_01:[16-cc-70-20-64-4d 4] connected
INFO:openflow.of_01:[86-43-d5-3e-d8-44 5] connected
INFO:openflow.of_01:[e6-09-4f-f5-96-4b 6] connected
INFO:openflow.discovery:link detected: 42-1d-d3-12-07-40.2 -> 9e-68-9f-4f-2d-44.1
INFO:openflow.discovery:link detected: 46-a4-a6-ae-be-49.1 -> 9e-68-9f-4f-2d-44.6
INFO:openflow.discovery:link detected: 16-cc-70-20-64-4d.7 -> 9e-68-9f-4f-2d-44.4
INFO:openflow.discovery:link detected: 86-43-d5-3e-d8-44.3 -> 9e-68-9f-4f-2d-44.3
INFO:openflow.discovery:link detected: e6-09-4f-f5-96-4b.6 -> 9e-68-9f-4f-2d-44.5
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.3 -> 86-43-d5-3e-d8-44.3
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.4 -> 16-cc-70-20-64-4d.7
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.1 -> 42-1d-d3-12-07-40.2
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.6 -> 46-a4-a6-ae-be-49.1
INFO:openflow.discovery:link detected: 9e-68-9f-4f-2d-44.5 -> e6-09-4f-f5-96-4b.6
INFO:host_tracker:Learned 252927670589003 3 fe:be:f4:a7:da:3a
INFO:host_tracker:Learned 252927670589003 3 fe:be:f4:a7:da:3a got IP 192.168.0.14
INFO:host_tracker:Learned 174172186553668 2 66:05:ea:b7:ce:77
INFO:host_tracker:Learned 174172186553668 2 66:05:ea:b7:ce:77 got IP 192.168.0.100
INFO:host_tracker:Learned 147625898596420 4 e2:91:a5:63:37:a7
INFO:host_tracker:Learned 147625898596420 4 e2:91:a5:63:37:a7 got IP 192.168.0.200
INFO:host_tracker:Learned 252927670589003 3 fe:be:f4:a7:da:3a moved to 147625898596420 4
WARNING:host_tracker:Possible duplicate: 252927670589003 3 fe:be:f4:a7:da:3a at time 1657275908, now (147625898596420 4), time 1657275919
```

In verde è evidenziata la posizione iniziale (e reale) dell'host vittima, di cui precedentemente il modulo host tracker era a conoscenza. Dopo l'attacco vediamo come il servizio di host tracking rileva una moved della vittima verso la porta Y.

Da questo momento in poi il controller crederà che l'host vittima si trovi sulla porta Y, dove invece risiede l'attaccante e dalla quale può inviare messaggi arbitrari. Quindi anche in questo caso la porta dello switch è come se fosse occupata da due host diversi.



4.2 Mitigazioni proposte

Il problema principale che porta alla perfetta riuscita degli attacchi appena descritti, è la completa assenza di controlli sulle porte degli switch che sono già occupate da altri host.

La mitigazione proposta fa leva sul fatto che, se una porta di uno switch, identificata dalla coppia *(dpid,port),* è attualmente occupata da un host attivo nella rete, non è possibile che risulti un pacchetto inviato da quella stessa porta ma da un host differente.

Per questo motivo è stato deciso di aggiungere al modulo, una struttura dati dizionario così formata:

```
#---------STRUTTURE PATCH-----------------
#key = dpid-porta values = mac
tab_check_join = {}
```

La struttura conterrà, per ogni porta occupata, una entry nella tabella il cui valore corrisponderà all'indirizzo MAC dell'host collegato ad essa.

Con l'ausilio di essa, sia quando si riceve un pacchetto da un nuovo host che da un host già presente nella rete, si controlla che la porta da cui proviene il pacchetto non sia già occupata da un altro host:

```
  #------PATCH JOIN NUOvO HOST----------------

  tab_check_join[key] = packet.src

  #controllare occorrenze uguali dpid-port
  macEntry = MacEntry(dpid,inport,packet.src)
  print(macEntry)
  self.entryByMAC[packet.src] = macEntry
  log.info("Learned %s", str(macEntry))
  self.raiseEventNoErrors(HostEvent, macEntry, join=True)
elif macEntry != (dpid, inport, packet.src):
  # there is already an entry of host with that MAC, but host has moved
  # should we raise a HostMoved event (at the end)?

  #------------PATCH MOVED SU PORTA OCCUPATA-------------
  if key in tab_check_join.keys():
    print('ATTACCO MOVED ILLECITA')
    return
```

Con questa logica si è riuscito a mitigare le vulnerabilità precedentemente introdotte.

```
INFO:host_tracker:Learned 271039670827335 1 52:8c:c8:c7:9e:fe
INFO:host_tracker:Learned 271039670827335 1 52:8c:c8:c7:9e:fe got IP 192.168.0.6
UPDATE
DEBUG:forwarding.l2_learning:installing flow for 52:8c:c8:c7:9e:fe.1 -> de:ad:be:ef:69:01.3
DEBUG:forwarding.l2_learning:installing flow for 52:8c:c8:c7:9e:fe.1 -> de:ad:be:ef:69:01.2
ATTACCO_MOVED_ILLECITA      ATTACCO RILEVATO!
DEBUG:forwarding.l2_learning:installing flow for ca:fe:ba:be:69:01.2 -> 52:8c:c8:c7:9e:fe.1
DEBUG:forwarding.l2_learning:installing flow for ca:fe:ba:be:69:01.3 -> 52:8c:c8:c7:9e:fe.1
UPDATE
DEBUG:forwarding.l2_learning:installing flow for 52:8c:c8:c7:9e:fe.1 -> ca:fe:ba:be:69:01.3
DEBUG:forwarding.l2_learning:installing flow for 52:8c:c8:c7:9e:fe.1 -> ca:fe:ba:be:69:01.2
```

È importante far notare che l'aggiunta/moved di un host su una porta può essere fatta quando essa è libera o quando un host che la occupava non è più attivo. La mitigazione introdotta deve agire, in accordo con il modulo principale, quando un host risulta scaduto e rimuoverlo dalla tabella *tab_check_join:*

```
def _check_timeouts (self):
  """
  Checks for timed out entries
  """
  for macEntry in list(self.entryByMAC.values()):
    entryPinged = False
    for ip_addr, ipEntry in list(macEntry.ipAddrs.items()):
      if ipEntry.expired():
        if ipEntry.pings.failed():
          del macEntry.ipAddrs[ip_addr]
          log.info("Entry %s: IP address %s expired",
                   str(macEntry), str(ip_addr) )
        else:
          self.sendPing(macEntry,ip_addr)
          ipEntry.pings.sent()
          entryPinged = True
    if macEntry.expired() and not entryPinged:
      log.info("Entry %s expired", str(macEntry))
      # sanity check: there should be no IP addresses left
      if len(macEntry.ipAddrs) > 0:
        for ip_addr in macEntry.ipAddrs.keys():
          log.warning("Entry %s expired but still had IP address %s",
                      str(macEntry), str(ip_addr) )
        macEntry.ipAddrs.clear()
      self.raiseEventNoErrors(HostEvent, macEntry, leave=True)
      del self.entryByMAC[macEntry.macaddr]
```

```
#-----------REMOVE HOST EXPIRED-----------------
for key in tab_check_join.keys():
    if tab_check_join[key] == macEntry.macaddr:
        del tab_check_join[key]
        break
```

4.3 Flusso di esecuzione di attacco e mitigazioni proposte

L'attacco può essere lanciato eseguendo lo script Python */attacchi/host_tracker/host_tracker_attack.py* all'interno di un host scelto come malevolo.
Per monitorare l'efficacia dell'attacco, i moduli POX da eseguire sono i seguenti:

python3 pox.py forwarding.l2_learning host_tracker openflow.discovery

Per verificare l'effetto delle mitigazioni, invece, il modulo POX da eseguire è */host_tracker/host_patch.py* . Attenzione: affinchè il controller riconosca il modulo, la patch deve essere rinominata in *host_tracker.py* ed eseguita come segue:

python3 pox.py forwarding.l2_learning host_tracker openflow.discovery

Inoltre, per evitare che i pacchetti malevoli generati dall'attaccante non arrivino al controller a causa di regole di forwarding installate sugli switch, i tempi di *hard_timeout* e *idle_timeout* del modulo *l2_learning* sono stati settati a pochi secondi.