ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

RILEVAMENTO E MITIGAZIONE DI ATTACCHI DDOS IN RETI SDN

Elaborato in:
Reti di telecomunicazione

Relatore
Prof. Franco Callegati

Presentata da
Luca Fabri

Sessione di Laurea III
Anno Accademico 2020-2021

Indice

1 Introduzione **1**

2 State of the Art **3**

2.1 Internet oggi . . . 3

2.1.1 Control Plane . . . 3

2.1.2 Data plane . . . 3

2.2 Software Defined Networking (SDN) . . . 4

2.2.1 Background . . . 4

2.2.2 Use cases . . . 5

2.3 Openflow . . . 5

2.3.1 Componenti dello switch . . . 6

2.3.2 Pipeline processing . . . 7

2.3.3 Openflow protocol . . . 7

2.4 Open vSwitch . . . 9

2.4.1 Componenti . . . 9

2.5 Attacchi DDoS (Distrubuted Denial-of-Service) . . . 10

2.5.1 Tassonomia . . . 11

2.5.2 Tassonomia delle vulnerabilità nel SDN . . . 11

2.5.3 OFA Overloading . . . 12

2.5.4 Packet Buffer Overflow . . . 13

2.5.5 Durata delle table entries . . . 13

2.5.6 Flow table overload . . . 14

2.5.7 Saturazione del Control channel ed esaurimento delle risorse del controller . . . 15

2.6 Strategie di rilevamento di attacchi DDoS . . . 15

2.6.1 Analisi statistiche . . . 15

2.6.2 Machine Learning . . . 17

2.6.3 Entropia dell'informazione . . . 24

3 Analisi e progettazione **29**
3.1 Requisiti ed analisi . . . 29
3.2 Tecnologie utilizzate . . . 30
3.2.1 Mininet . . . 30
3.2.2 Ryu Controller . . . 32
3.2.3 scikit-learn . . . 33
3.3 Progetto finale . . . 35
3.3.1 Rete virtuale . . . 35
3.3.2 Controller Ryu . . . 35
3.3.3 App Python . . . 35

4 Design **37**
4.1 Parametri di addestramento della Support Vector Machine . . . 37
4.1.1 Velocità degli indirizzi IP sorgente . . . 37
4.1.2 Deviazione standard del numero di pacchetti nelle flow entries 37
4.1.3 Deviazione standard del numero di bytes nelle flow entries . 38
4.1.4 Velocità delle flow entries . . . 38
4.1.5 Rapporto tra flow interattivi e flow totali . . . 38
4.2 Topologia di rete . . . 39
4.3 Architettura dell'applicazione . . . 41
4.3.1 Modellazione di DDoSControllerThread . . . 42
4.3.2 Modellazione di FeaturesController . . . 44

5 Sviluppo **47**
5.1 Realizzazione della topologia di rete . . . 47
5.2 Addestramento della SVM . . . 51
5.2.1 Simulazione di traffico “normale” . . . 52
5.2.2 Simulazione di traffico “anomalo” . . . 53
5.2.3 Calcolo dei parametri . . . 54
5.3 Sviluppo dell'applicazione . . . 57
5.3.1 Controllers . . . 57
5.3.2 View . . . 62

6 Conclusioni **65**

Ringraziamenti **67**

Elenco dei listati

3.1 Esempio di topologia 31
3.2 scikit-learn SVM 34
5.1 DDoS experimentation topology 47
5.2 Controller C1 48
5.3 Controller C2 50
5.4 Normal traffic 52
5.5 Traffic generator 52
5.6 FeaturesCalculator 54
5.7 DDoSControllerThread 58
5.8 Mitigazione 60

Capitolo 1

Introduzione

Nel contesto delle attuali architetture di networking il Software Defined Networking riveste un ruolo chiave infatti, rendendo la rete programmabile, permette di rispondere alla dinamicità delle applicazioni moderne, consente di rispondere in tempo reale alle richieste di larghezza di banda e di reagire al cambiamento controllando il flusso delle risorse.

Se da un lato presenta questi e molti altri vantaggi, dall'altro, la natura centralizzata del piano di controllo costituisce un single point of failure e rappresenta una lama a doppio taglio per gli attaccanti, che oltre a mettere fuori gioco un server all'interno della rete, riescono a negare il servizio agli utenti legittimi connessi nell'intera Software-Defined Network.

L'attacco in questione è l'attacco Distributed Denial-of-Service (DDoS).

In questo scenario, la mia tesi espone le problematiche relative alle vulnerabilità nell'architettura delle SDN e descrive le soluzioni presenti in letteratura per il rilevamento degli attacchi DDoS.

Successivamente, viene proposta un'implementazione di una di queste soluzioni, in particolare la soluzione di rilevamento che utilizza le Support Vector Machines (SVM), attraverso le quali è possibile classificare il traffico che scorre all'interno della rete.

Infine, attraverso l'applicazione sviluppata, l'attacco viene mitigato.

Il Capitolo 2 contiene un background sul Software Defined Networking, sul protocollo Openflow e le componenti degli switch abilitati al SDN. A seguire viene mostrata la tassonomia degli attacchi DDoS, la tassonomia delle vulnerabilità nell'architettura SDN e lo stato dell'arte delle soluzioni adottate per il rilevamento.

Nel capitolo 3 vengono descritte le specifiche del progetto ed elencate le tecnologie utilizzate.

Nel capitolo 4 si espongono i parametri scelti per l'addestramento, la topologia di rete virtuale e l'architettura dell'applicazione, marcando la modellazione delle

classi principali.

A seguire nel capitiolo capitolo 5 viene verificato il comportamento dei parametri della SVM ed è elencato il codice principale delle varie sottocomponenti del sistema.

Infine nel capitolo 6 si traggono le conclusioni.

Capitolo 2

State of the Art

2.1 Internet oggi

La rete Internet odierna è ancora composta in gran parte da dispositivi di rete quali switch e routers che svolgono le proprie funzioni attraverso hardware specializzato. Purtroppo, questi dispositivi, pur essendo particolarmente veloci risultano poco flessibili e difficilmente configurabili sia per limiti progettuali sia perché gestiti da software proprietario. Le funzioni svolte da questi dispositivi sono di Control Plane e Data Plane che analizzeremo nei paragrafi successivi.

2.1.1 Control Plane

Per Control plane si intendono tutte quelle funzioni che determinano come i pacchetti devono essere instradati nella rete, cioè il percorso che questi devono effettuare per arrivare a destinazione. Per determinare il percorso vengono utilizzati dal router degli algoritmi di instradamento ed esistono vari protocolli adatti a questo scopo che costruiscono di conseguenza le tabelle di routing.

I protocolli principali sono: BGP (Border Gateway Protocol), ISIS, OSPF (Open Shortest Path First), RIP (Routing Information Protocol).

2.1.2 Data plane

Chiamato anche forwarding plane, il data plane rappresenta tutte le funzioni che inoltrano il pacchetto da un'interfaccia ad un'altra, determinata dal control plane. Le funzionalità di switching sono implementate attraverso hardware dedicato per ottimizzare le performance, come gli Application Specific Integrated Circuits (ASIC).

2.2 Software Defined Networking (SDN)

2.2.1 Background

Il Software Defined Networking (SDN) è un approccio al networking che prevede la programmazione delle reti attraverso la separazione del control plane e data plane. Il SDN rappresenta un punto di rottura dalla rete tradizionale poichè la rete non è più controllata da singole grandi aziende che producono device chiusi e non modificabili ma dagli amministratori della rete che possono configurarla dinamicamente secondo le esigenze.

Nel SDN, l'intelligenza è contenuta in controllers centralizzati dove risiede il piano di controllo e che appaiono come uno switch logico alle applicazioni. Il piano di dati, invece, continua a risiedere nei dispositivi di rete, il cui compito è eseguire il forwarding secondo le regole dettate dal controller.

A questo scopo è necessaria un'interfaccia di comunicazione tra controllers e switch. La più conosciuta è Openflow, sviluppata dall'ONF (Open Networking Foundation) che rappresenta il primo protocollo di interfacciamento tra il livello di controllo e dati.

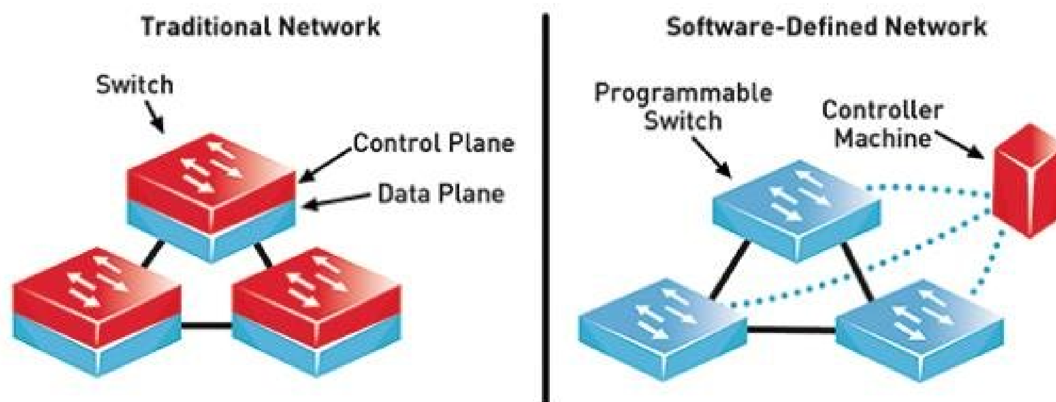

Figura 2.1: Traditional network vs SDN

Il Software Defined Networking ha quindi il vantaggio di essere:

- direttamente programmabile: il control plane è programmabile poichè è separato dal data plane;
- agile: l'amministratore di rete può adeguare il flusso di traffico a seconda delle esigenze;
- gestita centralmente: il controller ha una visione globale della rete;

- configurabile programmaticamente: le risorse di rete sono configurabili dinamicamente;
- maggiormente indipendente dai produttori di hardware;
- meno costosa a livello di manutenzione e di amministrazione;
- più semplice a livello di progettazione e funzionamento poichè i dispositivi e i protocolli non dipendono da fornitori privati.

2.2.2 Use cases

Tra i casi d'uso il SDN è utilizzato per scopi di:

- Sicurezza: integrare le NFV (Network Function Virtualization) con le SDN permette la creazione di una rete astratta completamente virtuale e capace di adattarsi ai cambiamenti. Quando un attacco è in corso e ogni secondo è critico, questa infrastruttura aiuta a creare un sistema capace di rispondere velocemente ai problemi;
- QoS (Quality of Service): il traffico di rete può essere regolato per soddisfare le richieste di larghezza di banda;
- Hyper-Converged storage/Converged storage;
- Applicazioni video;
- Orchestrazione di servizi di rete mobile;
- Scalabilità delle reti di Data centers;

2.3 Openflow

Openflow è il primo protocollo dell'ONF che standardizza l'interfaccia tra control plane e data plane, abilitando il deployment del SDN. Il protocollo Openflow permette di modificare il forwarding plane degli switch/routers, specificando le primitive di base che devono essere utilizzate. Il forwarding non è basato sull'IP destinazione ma viene utilizzato un approccio flow-based.

2.3.1 Componenti dello switch

Uno switch Openflow contiene una o più **flow tables**, una **group table** che esegue il lookup ed il forwarding dei pacchetti e un **Openflow channel** per la comunicazione con il controller.

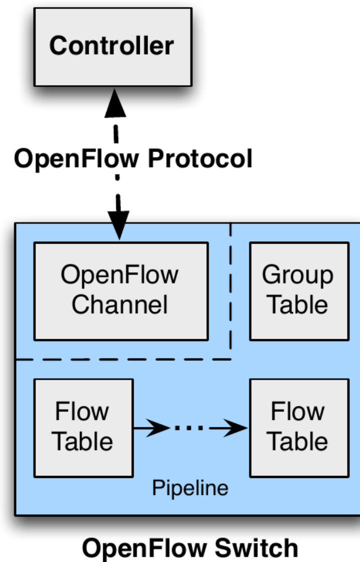

Figura 2.2: Componenti principali di uno switch Openflow

Utilizzando Openflow, il controller può aggiungere, modificare ed eliminare le flow entries di una flow table. Ognuna di queste specifica i campi per il matching, i contatori e una serie di istruzioni che devono essere eseguite per i pacchetti che fanno match.

Il matching di un pacchetto inizia dalla prima flow table e può proseguire sulle successive. Le flow entries di una tabella vengono ispezionate secondo un criterio di priorità ed al primo match vengono eseguite le istruzioni associate alla flow entry. Se in una flow table nessun match è soddisfatto viene controllata la table-miss flow entry. Questo tipo di entry può essere configurata in modo tale che invii il pacchetto al controller, che scarti il pacchetto (drop) o che lo spedisca alla flow table successiva [1].

Le istruzioni relative ad una flow entry possono essere di due tipologie:

- **azioni**: le azioni descrivono il forwarding del pacchetto, la modifica di questo e la group table. La group table specifica un processing aggiuntivo per il flooding e per forwarding più complesso come il multipath, fast rerouting e link aggregation. La group table contiene anch'essa delle group entries che a

loro volta contengono le action buckets. Le azioni di ciascuna action bucket vengono eseguite per ogni pacchetto spedito alla group entry;

- **pipeline processing**. Questo tipo di istruzione permette al pacchetto di essere spedito a flow tables successive per processing aggiuntivo e permette di scambiare informazioni tra tabelle sotto forma di metadati.

2.3.2 Pipeline processing

Le flow tables di uno switch Openflow sono sequenzialmente numerate partendo da 0.

Quando il pacchetto è processato, viene eseguito il matching in una flow entry di una flow table. Se una flow entry viene individuata, il suo set di istruzioni viene eseguito: se questo contiene l'istruzione Goto, il pacchetto viene spedito in una tabella successiva. Il numero di quest'ultima può essere solamente maggiore della flow table corrente, in altre parole, non si può navigare tra le tabelle in senso decrescente.

Se la flow entry non contiene l'istruzione Goto, il pipeline processing si ferma e viene eseguito il set di azioni e il pacchetto viene spedito [1].

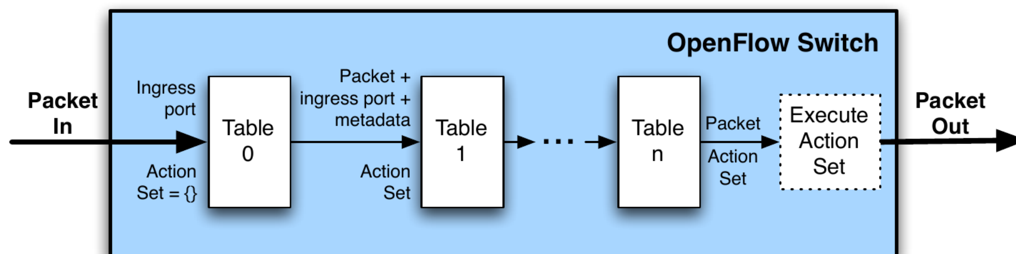

Figura 2.3: Pipeline processing

2.3.3 Openflow protocol

Il protocollo Openflow supporta tre tipologie di messaggi tra controller e switch [1]:

- **Controller-to-Switch**: questo tipo di messaggi sono inizializzati dal controller e vengono solitamente utilizzati per ispezionare lo stato dello switch. Si suddividono in ulteriori categorie:
  - Features: il controller interroga lo switch prendendo informazioni riguardo le sue capacità;

  - Configuration: il controller esegue query e setta i parametri di configurazione dello switch;
  - Modify-state: questo tipo di messaggio permette al controller di aggiungere e modificare le flow/group entries;
  - Read-state: permette di ottenere dallo switch informazioni riguardo la sua configurazione, capacità e statistiche;
  - Packet-out: viene utilizzato dal controller per spedire un messaggio dallo switch da una determinata porta. Il pacchetto da spedire va specificato nel messaggio in forma intera o attraverso un buffer ID se il pacchetto è bufferizzato nello switch. Inoltre vanno dichiarate una serie di azioni che lo switch deve eseguire, se tale insieme è vuoto, il pacchetto viene scartato (drop).
  - Barrier: request e reply per notificare l'avvenuto completamento delle operazioni;
  - Role-request: lo scopo di questo messaggio è assegnare il ruolo del canale tra controller e switch. È utile quando lo switch è connesso a molteplici controllers;
  - Asynchronous-Configuration: questo tipo di messaggio permette di specificare dei filtri sui messaggi Asynchronous ricevuti dallo switch. Anche questo è utile quando lo switch è connesso a molteplici controllers.

- **Asynchronous**: gli switch spediscono questo tipo di messaggio ad un controller per informarlo di un pacchetto in arrivo, di un cambiamento di stato o per un errore verificatosi. Si distinguono in quattro tipi:

  - Packet-in: per demandare il controllo del pacchetto al controller;
  - Flow-Removed: utilizzato per notificare il controller della rimozione di una flow entry in una flow table;
  - Port-status: per informare il controller di un cambiamento dello stato di una porta;
  - Error: per la notifica di eventuali problemi.

- **Symmetric**: messaggi in entrambe le direzioni, spediti senza sollecitazioni.

  - Hello: a connessione attivata questo messaggio viene spedito;
  - Echo: request e reply. Viene utilizzato per verificare che la connessione tra controller e switch sia ancora instaurata e può essere anche utilizzato per misurare la latenza;
  - Experimenter: "open" a sperimentazioni future;

2.4 Open vSwitch

Open vSwitch è uno switch software multilayer con licenza Apache 2. Supporta le interfacce di gestione standard e permette la programmabilità delle funzioni di forwarding. Open vSwitch è adatto ad eseguire le sue funzioni di switch virtuale in ambienti VM. Inoltre, oltre ad esporre le interfacce di visibilità e controllo allo strato di networking virtuale, è adatto ad essere eseguito in molteplici server hardware. Supporta infatti varie VM Linux-based tra cui Xen/XenServer, KVM, and VirtualBox.

Scritto quasi interamente in linguaggio C portabile, supporta le seguenti funzionalità:

- Modello standard 802.1Q VLAN con trunk e porte di accesso;
- Collegamento NIC con o senza LACP;
- Netflow, sFlow(R) e mirroring per una maggiore visibilità;
- Configurazione QoS;
- Openflow 1.0 e numerose estensioni;
- High-performance forwarding tramite modulo del kernel Linux.

2.4.1 Componenti

Le componenti di Open vSwitch sono:

- `ovs-vswitchd`: demone che implementa lo switch, insieme ad un modulo del kernel Linux per lo switching basato sui flussi;
- `ovsdb-server`: un lightweight database server che `ovs-vswitchd` interroga per ottenere la sua configurazione;
- `ovs-dpctl`: strumento per configurare il modulo kernel dello switch;
- `ovs-vsctl`: utility per interrogare ed aggiornare la configurazione del demone `ovs-vswitchd`;
- `ovs-appctl`: utility che spedisce comandi a demoni Open vSwitch operanti.

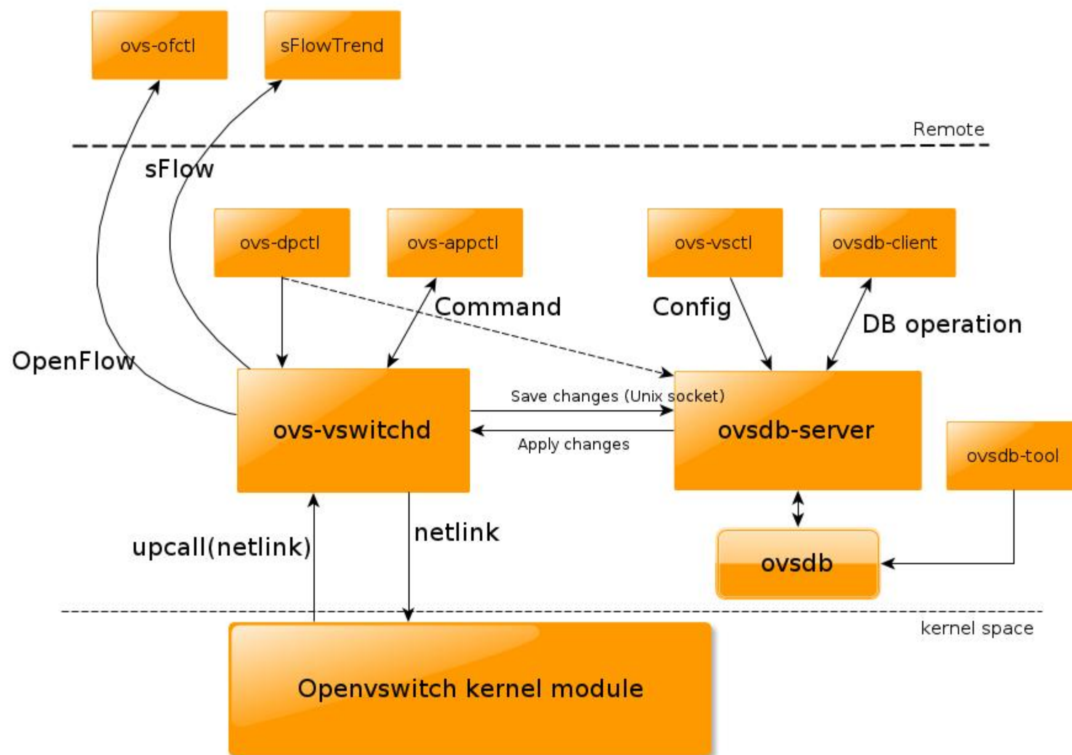

Figura 2.4: Componenti di Open vSwitch [2]

2.5 Attacchi DDoS (Distrubuted Denial-of-Service)

Il tradizionale attacco Denial of Service (DoS) è normalmente un attacco punto-a-punto. L'attaccante sfrutta la capacità di inviare richieste per un servizio per saturare le risorse di quest'ultimo, allo scopo di forzare il server al crash o rendere il servizio incapace di rispondere a richieste provenienti da utenti legittimi.

L'attacco Distributed Denial-of-Service (DDoS) è un'estensione del DoS. Questo infatti non coinvolge solo un host ad eseguire l'attacco ma è distribuito, cioè è performato da centinaia o migliaia di PC attaccanti su cui è installato un Daemon che ha lo scopo di coordinarlo. I target di un attacco DDoS sono normalmente siti di grandi dimensioni, motori di ricerca, dipartimenti governativi.

In confronto al tradizionale DoS, quest'ultima tipologia possiede un maggior numero di sorgenti e di conseguenza è più efficace. In più, sono più difficili da rilevare ed è più difficile difendersi.

Ora che gli attacchi DDoS tendono ad essere sempre più automatizzati, la distruzione di questi ha maggior rilevanza. Gli attacchi fanno uso, tra le altre, delle seguenti tecniche:

- uso di cluster di PC per performare attacchi più intensi;
- producono pacchetti con sorgenti IP random (spoofed) per nascondere le tracce;
- sfruttano i bug dei protocolli di rete e dei sistemi operativi;
- inviano i pacchetti in maniera più veloce senza apparente intenzione di attacco.

Di seguito viene mostrata la tassonomia degli attacchi DDoS.

2.5.1 Tassonomia

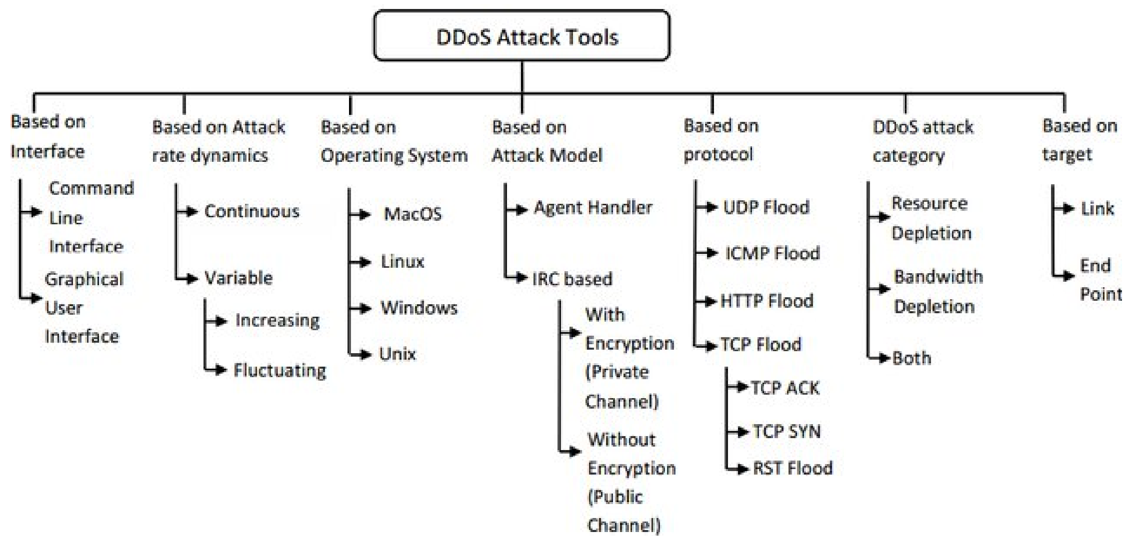

Figura 2.5: Tassonomia attacchi DDoS

2.5.2 Tassonomia delle vulnerabilità nel SDN

A seguire, viene mostrata la classificazione delle vulnerabilità new-flow nell'architettura SDN.

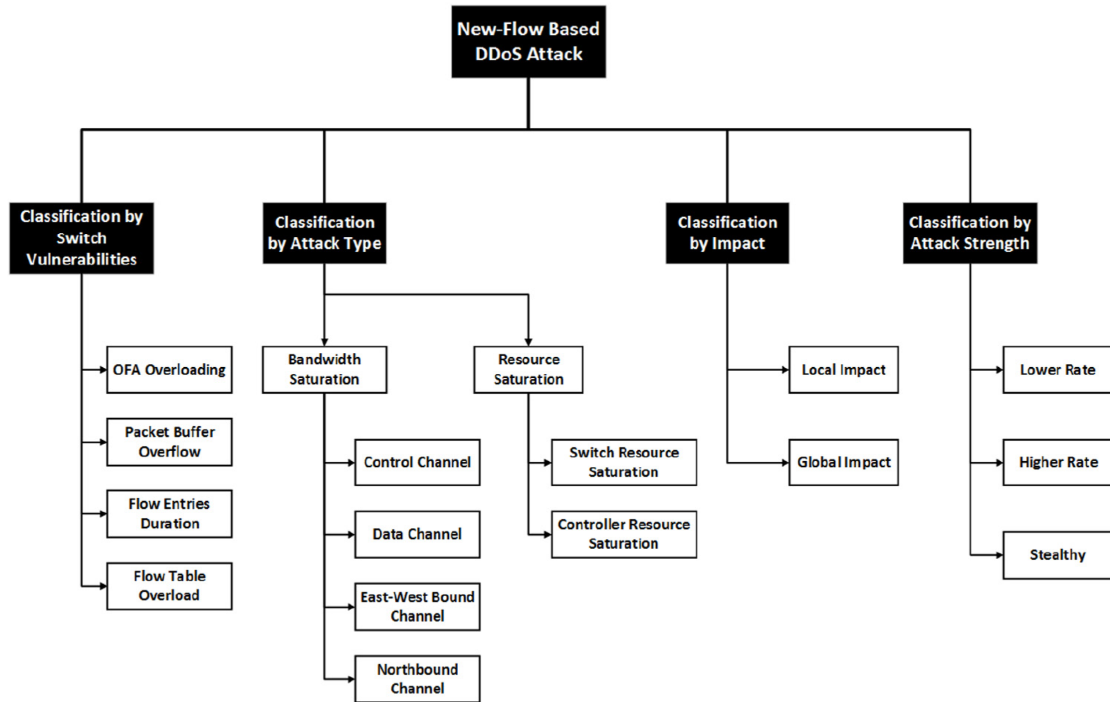

Figura 2.6: Tassonomia delle vulnerabilità new-flow nel SDN [3]

Nella classificazione basata sulle vulnerabilità di uno switch Openflow vengono mostrate quattro vulnerabilità, strettamente interconnesse tra di loro a tal punto che un attacco new-flow può puntare a una o più di queste.

A seguire vengono descritte alcune di queste insieme all'attacco basato sulla saturazione del Control Channel, strettamente legato a quello che mira all'esaurimento delle risorse del controller.

2.5.3 OFA Overloading

OFA (Open Flow Agent) è un software agent che si trova all'interno degli switch Openflow. Il suo scopo è fungere da intermediario tra switch e controller: all'ingresso di un pacchetto, questo controlla le flow entries per verificare che ne sia installata una. In caso negativo, invia un pacchetto *Packet In* al controller via una comunicazione sicura e alla ricezione del *Packet Out* si occupa di installare ove necessario le flow entries all'interno dello switch.

Un attaccante può sovraccaricare questo modulo facendo generare tante table-miss, a tal punto da non riuscire a gestire l'incapsulamento di pacchetti *Packet In* destinati al controller.

Questa vulnerabilità è sfruttabile da un attaccante che come risultato riesce ad esaurire le risorse computazionali dello switch. Pertanto, i client connessi allo switch noteranno una perdita di connettività.

2.5.4 Packet Buffer Overflow

Un'altra caratteristica degli switch Openflow è che essi contengono un buffer. Lo scopo di questo buffer è memorizzare parte del pacchetto in ingresso: se questo deve essere incapsulato in un messaggio Packet In, solo l'header viene spedito verso il controller.

Quando questo buffer si riempe, il campo `max_len` in `ofp_action_output` ha valore `OFPCML_NO_BUFFER` e il pacchetto in ingresso viene spedito interamente verso il controller.

Quando lo switch è sotto attacco, come risultato si ha una saturazione del canale southbound e un esaurimento delle risorse del controller.

Questo tipo di attacco porta ad aumentare la latenza ed il tempo di risposta dello switch, facendo di conseguenza perdere la connettività ai client poichè i pacchetti risulteranno persi.

2.5.5 Durata delle table entries

Negli switch Openflow le flow entries sono regolate da un meccanismo di timeout che determina la loro scadenza all'interno delle flow tables.

Esistono due tipi di timeout: *idle_timeout* e *hard_timeout*. Il primo indica per quanto tempo la flow entry rimane attiva se nessun pacchetto in entrata fa match. Se questo timer non è scaduto, all'arrivo di un pacchetto che fa match viene resettato.

Il secondo timout invece indica per quanto tempo la flow entry rimane memorizzata nella flow table, indipendentemente dall'arrivo di pacchetti che ne fanno il match.

Il timeout *idle_timeout* può essere sfruttato da attaccanti che possono inviare pacchetti ad una bassa frequenza allo scopo di mantenere le flow entries attive per un lungo periodo di tempo. Questo attacco che sfrutta il timeout rimane particolarmente nascosto (Stealthy DDoS attack) agli occhi di qualche metodo tradizionale di rilevamento.

In particolare, uno stealthy DDoS attack deve soddisfare i seguenti requisiti:

1. La degradazione delle performance deve essere inferiore alla soglia adottata da qualche meccanismo di rilevazione;

2. È possibile mimetizzare questo attacco facendolo sembrare un flash crowds, cioè un aumento inaspettato di richieste da client che richiedono un servizio;

3. Non punta ad esaurire le risorse del controller SDN per non far scattare altri meccanismi di rilevazione.

Algoritmo di attacco

Sia:

- m il numero di bot attivi;
- $\lambda_a = \frac{1}{m}\sum_{i=1}^{m}\lambda_a^i$ dove $\forall i \in [1, m]$, λ_a^i è il tasso medio di invio dei pacchetti dall'i-esimo bot in un intervallo di T_a secondi;
- T_d durata dell'attacco.

Scegliamo $\lambda_a \sim \lambda_n$ dove λ_n è il tasso di arrivo di pacchetti in condizioni normali di traffico, aggiustiamo T_a in base all'*idle_timeout* τ: $\tau_{min} \geq 1$ allora $T_a = 1$.

Algoritmo 1: Stealthy DDoS attack against SDN flow table based on *idle_timeout* [4]

```
/* Required input parameters */
- Specify m and λ_a^j
- Set the attack period T_a
- Decide a sufficiently large time window T_d
/* Attack burst orchestration */
j = 1
while j ≤ m do
    t = 0
    Send attack flows (θ_1^j, θ_2^j, ..., θ_{λ_a^j}^j) for duration T_a
    t = t + T_a
    while t ≤ T_d do
        Generate different attack packets φ_k^j for each flow θ_k^j
        Send attack flows (φ_1^j, φ_2^j, ..., φ_{λ_a^j}^j) for duration T_a
        t = t + T_a
    end
    j = j + 1
end
```

2.5.6 Flow table overload

La limitata dimensione delle flow tables può essere sfruttata da eventuali attaccanti che possono riempirle di flow entries, danneggiando di conseguenza le risorse di rete.

2.5.7 Saturazione del Control channel ed esaurimento delle risorse del controller

La Southbound API, anche detto Control Channel, rappresenta il canale di comunicazione tra uno switch Openflow e il suo controller. Per permettere al controller di essere sempre connesso ad ogni switch all'interno della rete, è cruciale che questo canale risulti sempre libero, al fine di un'installazione corretta e reattiva delle flow entries all'interno degli switch.

Il Control Channel può essere sfruttato da eventuali attaccanti che attraverso la generazione di pacchetti con IP spoofed lo rendono saturato di messaggi Packet In provenienti dagli switch. In aggiunta, combinato con il Packet Buffer Overflow 2.5.4 il risultato può essere ancora più devastante.

Nel caso peggiore, questa tipologia di attacco porta all'esaurimento delle risorse CPU e di memoria RAM del controller, e ad una conseguente perdita di connettività da parte degli hosts connessi alla rete SDN.

2.6 Strategie di rilevamento di attacchi DDoS

I principali metodi per il rilevamento di attacchi DDoS in reti SDN sono basati su analisi statistiche, machine learning ed information entropy [3].

2.6.1 Analisi statistiche

In condizioni normali di traffico vengono raccolti dati di profilo: se un pacchetto in ingresso non soddisfa i requisiti di traffico legittimo viene marcato come malevolo.

Per mitigare l'attacco viene limitato il traffico a questi pacchetti o vengono completamente scartati.

Gli autori in [5] hanno proposto un metodo dove vengono analizzati quanti pacchetti per connessione vengono spediti da una sorgente e quante connessioni questa ha effettuato. I pacchetti di un attacco DDoS mantengono un numero relativamente basso di connessioni con lo switch dovuto all'utilizzo di pacchetti spoofed IP e viene riscontrato anche un basso numero di pacchetti transitati per connessione rispetto ad un utente legittimo connesso allo switch.

Lo schema del metodo che hanno proposto è il seguente:

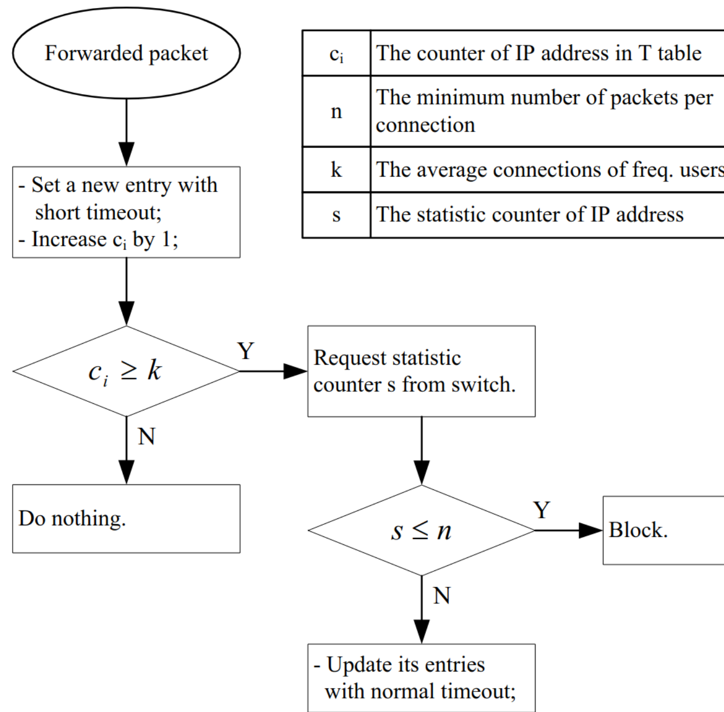

Figura 2.7: Workflow [5]

- Viene mantenuta una tabella T nel controller che associa ad ogni indirizzo IP il numero di pacchetti c_i arrivati al controller;
- All'arrivo di un nuovo pacchetto questo viene considerato malevolo, viene incrementato c_i e viene creata una flow entry con dei tempi di timeout per *idle_timeout* e *hard_timeout* molto brevi cosicchè rimanga attiva per poco tempo;
- Se $c_i \geq k$, quindi il numero di connessioni da una sorgente supera k, viene analizzato il numero medio s di pacchetti per connessione: se supera n allora la sorgente è considerata legittima e vengono aggiornati i tempi di timeout della flow entry in valori "normali".

Un altro metodo proposto in [6] permette di proteggere il canale di comunicazione tra switch e controller da attacchi di tipo flooding DDoS tramite l'installazione di un framework direttamente installato nello switch Openflow.

Il framework è composto da due moduli:

- **Connection migration**. L'idea che sta alla base di questa tecnica di mitigazione è quella di permettere la comunicazione con il controller ai soli pacchetti che hanno completato il TCP handshake con lo switch, scartando

di conseguenza tutti gli altri: lo scopo di questo modulo è proprio quello di effettuare tale distinzione.

Si suddivide in quattro fasi, qui sotto riportate nel diagramma a stati:

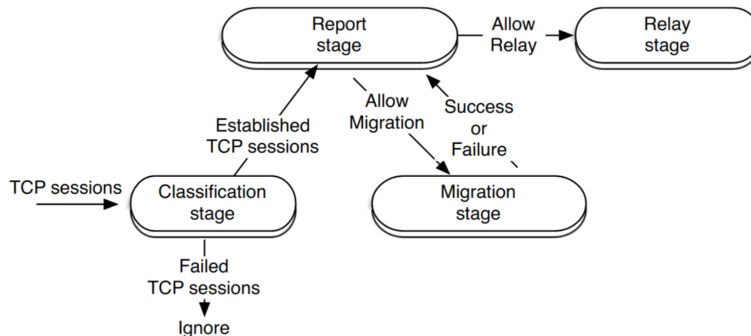

Figura 2.8: Connection migration diagramma a stati [6]

- **Actuating triggers**: questo modulo permette di spedire in modo asincrono informazioni riguardo lo stato di rete e payloads dallo switch al controller e, in condizioni particolari, ha lo scopo di inserire dinamicamente le flow rules per permettere al controller di gestire con miglior efficienza i flow quando si è sotto attacco.

Le tecniche basate su analisi statistiche, anche se molto veloci ed intuitive, non risultano particolarmente adatte ad attacchi DDoS massivi e non escludono la possibilità di falsi positivi cioè pacchetti considerati malevoli ma che in realtà sono provenienti da utenti legittimi.

Inoltre, la seconda tecnica presentata non copre la vasta gamma di attacchi performabili in reti SDN.

2.6.2 Machine Learning

Il machine learning fornisce sistemi di difesa capaci di auto apprendere partendo da set di dati di grandi dimensioni. Alcune tecniche basate sul machine learning sono le neural networks, Bayesian networks, Self-organizating map (SOP), Naïve Bayes e Support Vector Machines (SVM) con le quali è possibile classificare il traffico di rete, segregando quello generato da attacco da quello legittimo.

La classificazione è eseguita sulla base di alcune caratteristiche di rete catturate dagli switch Openflow tra cui: velocità di arrivo dei pacchetti, deviazione standard del numero di pacchetti e dei bytes transitati nelle flow entries, velocità di creazione delle flow entries, numero delle flow entries interattive rispetto al numero totale.

Support Vector Machines (SVM)

Le Support Vector Machines (SVM) sono dei modelli di apprendimento supervisionato con algoritmi che analizzano i dati per la classificazione, l'analisi della regressione e il rilevamento di anomalie. Sviluppato da Vladimir Vapnik e i suoi colleghi nei laboratori AT&T Bell sono diventate note internazionalmente solamente dal 1992.

Definite principalmente per la classificazione binaria, cioè per istanze appartenenti in modo esclusivo a una tra due classi, hanno come obiettivo l'individuazione di una separazione lineare ottimale tra le istanze delle due classi.

Le Support Vector Machines sono:

- Efficaci per domini di grandi dimensioni;
- Alla stessa maniera efficaci nei casi in cui il numero di dimensioni supera il numero delle istanze;
- Usa un set di vettori tra le istanze per la funzione di classificazione chiamati support vectors, quindi permette di risparmiare memoria;
- Versatile: permette l'utilizzo di diverse funzioni Kernel.
- Ha numerose applicazioni tra cui riconoscimento del testo, classificazione delle immagini e bioinformatica;

D'altra parte, presenta degli svantaggi:

- Se il numero dei parametri supera di gran lunga il numero di istanze, per evitare l'overfitting è cruciale scegliere in modo opportuno la funzione Kernel e il termine di regolarizzazione;
- Non forniscono stime di probabilità.

Definizione del problema

Sia $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)\}$ l'insieme delle m istanze di addestramento dove $x_i = (x_1, \ldots, x_n) \in \mathbb{R}^n, y_i \in \{-1, 1\}$ $\forall i = 1, \ldots, n$ sono rispettivamente i vettori delle istanze e la classi di appartenenza.

Il problema consiste nel determinare l'iperpiano di separazione $w^T \cdot x + b = 0$ ottimale cioè tale da massimizzare il margine di separazione tra le due classi, in particolare la distanza tra i support vectors delle due classi (punti più vicini all'iperpiano).

La massimizzazione del margine permette di ottenere la classificazione di dati futuri con maggior precisione minimizzando gli errori.

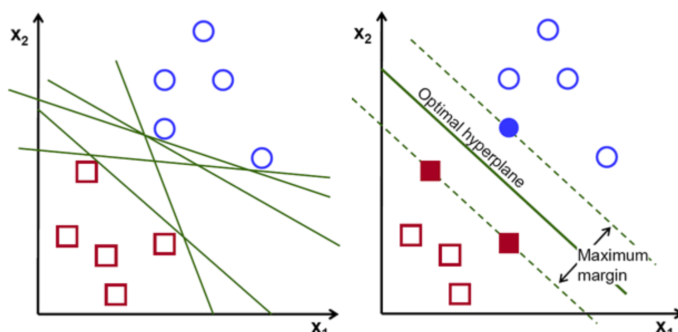

Figura 2.9: Insieme di possibili iperpiani vs iperpiano ottimale [7]

Per determinare in quale lato dell'iperpiano un campione $\vec{u}$ giace, viene calcolata la proiezione ortogonale di $\vec{u}$ nel vettore $\vec{w}$, dove $\vec{w}$ è perpendicolare all'iperpiano. Se il prodotto scalare è maggiore di b (bias) il vettore appartiene alla classe + (**decision rule**):

$$\vec{w} \cdot \vec{u} + b \geq 0 \implies f(u) = +1 \tag{2.1}$$

dove f è la funzione di classificazione.

Per calcolare $\vec{w}, b$ imponiamo le seguenti condizioni:

$$\vec{w} \cdot \vec{x_+} + b \geq 1 \tag{2.2}$$

$$\vec{w} \cdot \vec{x_-} + b \leq -1 \tag{2.3}$$

dove per i support vectors l'uguaglianza è soddisfatta, e introduciamo una nuova variabile per ottenere un solo limite invece che due.

$$y_i = \begin{cases} +1 \text{ se } \vec{x_+} \\ -1 \text{ se } \vec{x_-} \end{cases}$$

e partendo dalla precedente otteniamo:

$$y_i(\vec{w} \cdot \vec{x_i} + b) - 1 \geq 0 \tag{2.4}$$

Se $\vec{x_i}$ è un support vector:

$$y_i(\vec{w} \cdot \vec{x_i} + b) - 1 = 0 \tag{2.5}$$

Il problema originario consiste nel massimizzare la distanza tra i support vectors. Partendo da due possibili support vectors è possibile ottenere la distanza sottraendo i due vettori e moltiplicandoli per il vettore unitario perpendicolare all'iperpiano, ($\vec{w}$ vettore perpendicolare):

$$d = (\vec{x_+} - \vec{x_-}) \cdot \frac{\vec{w}}{\|\vec{w}\|} \tag{2.6}$$

Esplicitando $\vec{x_i}$ in 2.5 semplifichiamo la precedente equazione e otteniamo:

$$d = \frac{1-b}{\|\vec{w}\|} - \frac{-b-1}{\|\vec{w}\|} = \frac{2}{\|\vec{w}\|} \tag{2.7}$$

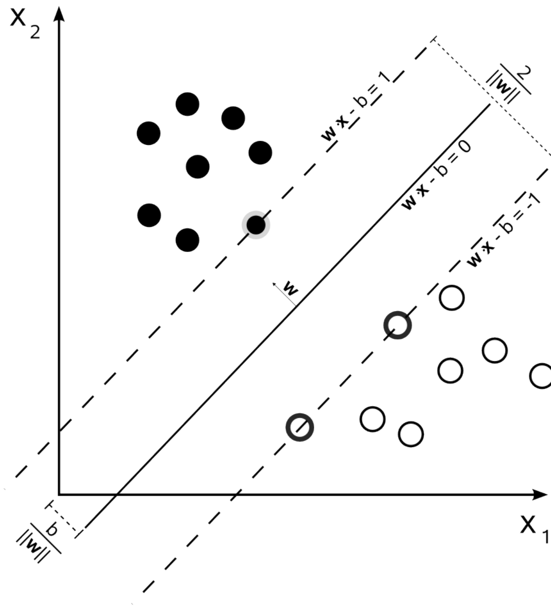

Figura 2.10: 2.2, 2.3, 2.7

Quindi il problema originario consiste in:

$$\max \frac{2}{\|\vec{w}\|} \rightarrow \min \frac{1}{2}\|\vec{w}\|^2 \tag{2.8}$$

Metodo dei moltiplicatori di Lagrange

Data la funzione obiettivo 2.8 ed il vincolo 2.5, attraverso il metodo dei moltiplicatori di Lagrange è possibile formulare il problema in modo diverso, riducendo i punti stazionari della funzione obiettivo f vincolata da g a quelli della funzione langragiana Λ senza vincoli:

$$\Lambda(\vec{x}, \vec{\lambda}) = f(\vec{x}) + \vec{\lambda} \cdot \vec{g}(\vec{x}) = f(\vec{x}) + \sum_{j=1}^{J} \lambda_j g_j(\vec{x}) \tag{2.9}$$

dove f è in I variabili e J è il numero di vincoli g, $\vec{\lambda}$ è il vettore dei moltiplicatori di Lagrange.

Sostituendo f e g la langragiana da minimizzare ha la seguente forma:

$$\Lambda(\vec{x}, \vec{\lambda}) = \frac{1}{2}\|\vec{w}\|^2 - \sum_{i=1}^{m} \lambda_i [y_i(\vec{w} \cdot \vec{x_i} + b) - 1] \tag{2.10}$$

Per trovare i punti di minimo poniamo le derivate parziali rispetto a $\vec{w}$ e b uguali a zero:

$$\frac{\partial \Lambda}{\partial \vec{w}} = \vec{w} - \sum_{i=1}^{m} \lambda_i y_i \vec{x_i} = 0 \implies \vec{w} = \sum_{i=1}^{m} \lambda_i y_i \vec{x_i} \tag{2.11}$$

$$\frac{\partial \Lambda}{\partial b} = \sum_{i=1}^{m} \lambda_i y_i = 0 \tag{2.12}$$

e sostituiamo $\vec{w}$ alla langragiana di partenza:

$$\begin{aligned} \Lambda(\vec{x}, \vec{\lambda}) = \frac{1}{2} &\left(\sum_{i=1}^{m} \lambda_i y_i \vec{x_i} \right) \cdot \left(\sum_{j=1}^{m} \lambda_j y_j \vec{x_j} \right) \\ &- \sum_{i=1}^{m} \lambda_i y_i \vec{x_i} \cdot \left(\sum_{j=1}^{m} \lambda_j y_j \vec{x_j} \right) - \cancel{\sum_{i=1}^{m} \lambda_i y_i b} + \sum_{i=0} m \lambda_i \end{aligned} \tag{2.13}$$

$$= \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j \vec{x_i} \cdot \vec{x_j} \tag{2.14}$$

Grazie a quest'ultimo risultato 2.14 possiamo notare che la massimizzazione del margine dipende dal prodotto scalare di due vettori tra le istanze.

Infine, sostituendo $\vec{w}$, la decision rule in 2.1 si esprime in:

$$\sum_{i=1}^{m} \lambda_i y_i \vec{x_i} \cdot \vec{u} + b \geq 0 \implies f(u) = +1 \tag{2.15}$$

Margine hard e margine soft

La formulazione del problema in 2.8 non ammette errori nel training set, cioè il margine viene scelto in modo tale da non avere punti al suo interno, perciò si dice che il margine è **hard**.

Introducendo delle variabili slack ζ, possiamo spostare gli errori trasformandoli in support vectors: il problema deve essere formulato in modo diverso poichè massimizzando solamente il margine senza minimizzare la somma delle ζ otterremmo un margine che conterrebbe tutte le istanze.

Di conseguenza minimizzando la somma delle variabili slack ζ, 2.8 prende la seguente forma ed il margine viene chiamato **soft**:

$$\min \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{m}\zeta_i \text{ dove } y_i(\vec{w}\cdot\vec{x}+b) \geq 1-\zeta_i, \quad \zeta_i \geq 0\ \forall i = 1,\ldots,m \tag{2.16}$$

C è il parametro di regolarizzazione inverso che permette di controllare l'overfitting.
Aumentando C aumenta il peso delle variabili slack ζ e come risultato si ha una riduzione del margine ed un aumento dell'overfitting, cioè una minore accuratezza sulle nuove istanze. Al contrario, diminuendo C aumenta il margine e di conseguenza gli errori di fitting, ma aumenta l'accuratezza per le nuove istanze poichè c'è una maggior separazione tra le due classi.

Kernel trick

Possiamo trovarci in una situazione in cui lo spazio non è linearmente separabile:

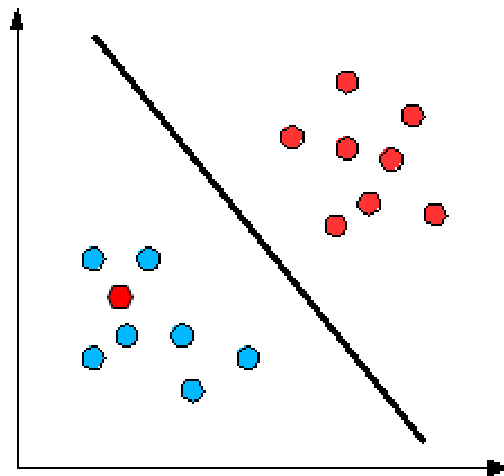

Figura 2.11: Spazio non separabile [8]

Introduciamo una trasformazione $\phi(\vec{x_i})$. Tale trasformazione permette di eseguire una mappatura non lineare del vettore in uno spazio separabile, in particolare

il vettore $\vec{x_i}$ viene mappato in uno spazio di dimensione superiore. Esempio:

$$\phi(x) = \phi\left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix} \tag{2.17}$$

mappatura polinomiale del vettore in 2-d in 3-d.

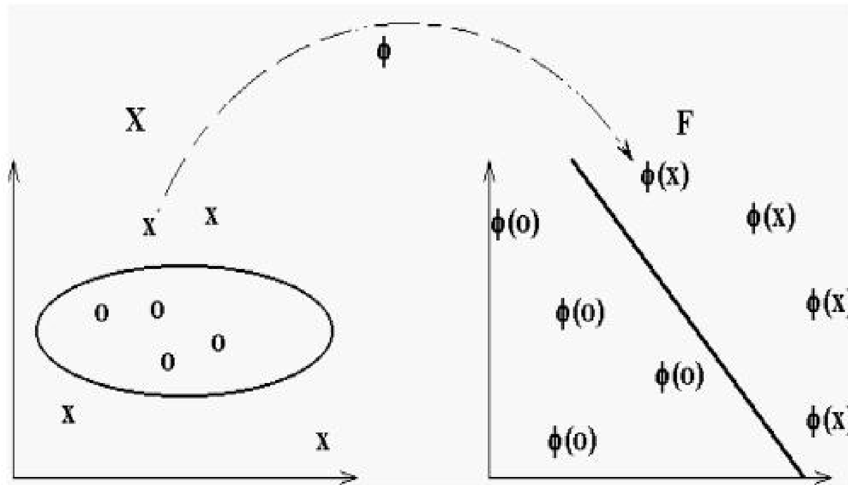

Figura 2.12: Kernel polinomiale [9]

Di conseguenza la funzione langragiana 2.14 si esprime in:

$$\Lambda(\vec{x}, \lambda) = \sum_{i=1}^{m} \lambda_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \lambda_i\lambda_j y_i y_j \phi(\vec{x_i}) \cdot \phi(\vec{x_j}) \tag{2.18}$$

Applicare le trasformazioni per ottimizzare la funzione obiettivo, in situazioni reali cioè quando le dimensioni sono elevate, è computazionalmente costoso perciò entra in gioco il kernel trick.

Definiamo la funzione Kernel K come una funzione che prende due vettori nello spazio originario ed esegue il prodotto scalare dei due vettori nello spazio di dimensione superiore:

$$K(\vec{x}, \vec{y}) = \langle \phi(\vec{x}), \phi(\vec{y}) \rangle \tag{2.19}$$

Funzione Kernel per mappatura polinomiale di secondo grado:

$$\begin{aligned}\phi(\vec{x})^T \cdot \phi(\vec{y}) &= \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} y_1^2 \\ \sqrt{2}y_1y_2 \\ y_2^2 \end{pmatrix} = x_1^2y_1^2 + 2x_1y_1x_2y_2 + x_2^2y_2^2 \\ &= (x_1y_1 + x_2y_2)^2 = \left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}^T \cdot \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right)^2 = (\vec{x}^T \cdot \vec{y})^2\end{aligned} \tag{2.20}$$

Ora è evidente che è sufficiente sostituire il risultato ottenuto nella funzione obiettivo 2.21: senza la necessità di eseguire le trasformazioni su tutti i vettori includiamo direttamente la funzione Kernel (**Kernel trick**):

$$\Lambda(\vec{x}, \lambda) = \sum_{i=1}^{m} \lambda_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \lambda_i \lambda_j y_i y_j K(\vec{x_i}, \vec{x_j}) \tag{2.21}$$

Esempi di funzioni Kernel:

- lineare: $K(\vec{x}, \vec{y}) = \langle \vec{x}, \vec{y} \rangle$
- polinomiale $K(\vec{x}, \vec{y}) = (\vec{x} \cdot \vec{y} + r)^d$
- Gauss Radial Basis: $K(\vec{x}, \vec{y}) = \exp -\gamma \|\vec{x} - \vec{y}\|^2$, $\gamma > 0$
- Sigmoidale $K(\vec{x}, \vec{y}) = \tanh k\vec{x} \cdot \vec{y} + r$

2.6.3 Entropia dell'informazione

L'entropia dell'informazione o entropia di Shannon è uno tra i più comuni e adottati metodi di rilevamento di attacchi DDoS. Molti ricercatori negli anni l'hanno scelta per la sua semplicità ed efficacia.

Nell'ambito della teoria dell'informazione, l'entropia rappresenta la media di informazione o incertezza contenuta in un messaggio emesso da una sorgente.

Il concetto è stato introdotto da Claude Shannon nell'articolo "A Mathematical Theory of Communication" nel 1948.

Autoinformazione

L'idea che sta alla base dell'informazione contenuta in un messaggio (autoinformazione) è che questa dipende dal grado di improbabilità che questo venga comunicato da una sorgente.

Se un evento è probabile che accada, non c'è sorpresa pertanto non è interessante. Di conseguenza, il messaggio che viene trasportato contiene meno informazione rispetto a un evento meno probabile: l'autoinformazione è inversamente proporzionale alla probabilità P che un certo valore x è assegnato ad una variabile X.

Autoinformazione:

$$I(x) = \log_b \left(\frac{1}{P(x)} \right) = -\log_b(P(x)) \tag{2.22}$$

Il logaritmo in base b indica che l'autoinformazione esprime il numero di cifre in base b necessarie per distinguere l'evento accaduto da tutti gli altri equiprobabili. Nella notazione posizionale possiamo infatti esprimere N eventi equiprobabili con $\log_b(N)$ cifre in base b.

Entropia

Data una variabile random X che può assumere valori $x_1, x_2, \ldots, x_n$ che si verificano con probabilità $P(x_1), P(x_2), \ldots, P(x_n)$ rispettivamente, l'entropia dell'informazione è definita come il valore atteso dell'autoinformazione:

$$H(X) = E[I(X)] = E(-\log_b(P(X)) \tag{2.23}$$

cioè il numero medio di cifre in base b da utilizzare per memorizzare un messaggio prodotto dalla sorgente X.

Se la sorgente X è una variabile aleatoria discreta il valore atteso è l'informazione contenuta in un messaggio x_i pesata con la probabilità $P(x_i)$:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_b(P(x_i)) \tag{2.24}$$

Esempio:

Sia X una sorgente binaria che ha p probabilità di produrre 1 e q probabilità di produrre 0 ($p + q = 1$), allora l'entropia è uguale a:

$$H(X) = -(p \log_2 p + q \log_2 q) = -[p \log_2 p + (1 - p) \log_2(1 - p)]$$

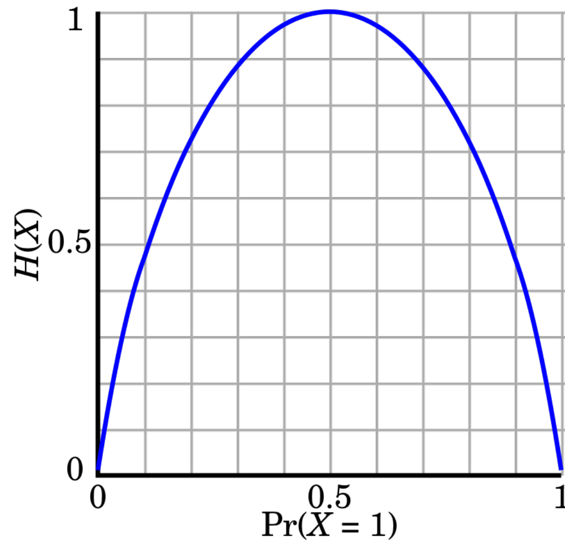

Figura 2.13: Entropia di una sorgente binaria [10]

Se $p = q = 0.5$ l'entropia vale 1 bit mentre nel caso in cui $p = 1$ o $q = 1$ la sorgente è completamente prevedibile (non c'è informazione) e di conseguenza vale 0.

Attraverso l'entropia, sotto certe condizioni, è possibile identificare quando si è soggetti ad un attacco in breve tempo.

Per il rilevamento, l'entropia utilizza due componenti principali: una finestra temporale o basata sulla dimensione dei pacchetti ed il valore di soglia che se superato fa scattare l'allarme [3].

I parametri del traffico di rete che vengono utilizzati per applicare l'entropia variano in base alla tipologia di attacco DDoS e possono includere: IP sorgente, porta sorgente, IP destinazione, porta destinazione, protocollo. Maggiore l'entropia, maggiore la randomicità del traffico di rete.

Per esempio, un parametro rilevante è l'IP destinazione. Quando una vittima si trova sotto attacco viene bombardata da pacchetti con lo stesso IP destinazione, di conseguenza l'entropia diminuisce velocemente e quando supera un valore di soglia l'attacco viene rilevato.

Gli autori in [11], propongono un sistema di rilevamento basato proprio sull'IP destinazione.

Date ϵ finestre di osservazione $W = \{W_1, W_2, \ldots, W_\epsilon\}$ ognuna delle quali si riempie quando n pacchetti hanno transitato la rete, la soglia (threshold) viene calcolata prendendo la media delle entropie delle ϵ finestre di osservazione quando il traffico è “normale”:

$$\delta = \frac{1}{\epsilon} \sum_{i=1}^{\epsilon} H(W_i) \tag{2.25}$$

Al riempimento di una nuova finestra viene calcolata l'entropia. Se questa supera l'entropia media δ l'attacco viene rilevato.

Capitolo 3

Analisi e progettazione

In questo capitolo verrà descritto il sistema progettato, partendo dall'analisi fino alla progettazione.

3.1 Requisiti ed analisi

Il progetto si pone come obiettivi:

- lo sviluppo di un'applicazione che ha come requisiti la rilevazione di anomalie nel traffico di una rete SDN (in particolare relative ad attacchi di tipo DDoS), e la loro mitigazione;
- la simulazione di uno scenario SDN, mettendo a punto una topologia virtuale con switch Openflow e uno o più controllers per sperimentare il funzionamento dell'applicazione.

In particolare, l'applicazione deve:

- collezionare le flow entries relative ad uno switch Openflow allo scopo di analizzare il traffico;
- determinare se il traffico è normale o anomalo utilizzando una Support Vector Machine appropriatamente addestrata;
- applicare/eliminare flow entries nel relativo switch quando un attacco viene rilevato;
- mantenere la connessione attiva per utenti legittimi in caso di attacco;
- mantenere le flow entries attive per il tempo necessario fino all'estinzione dell'attacco.

3.2 Tecnologie utilizzate

Per la simulazione della rete si è utilizzato Mininet, attraverso il quale è possibile istanziare un Open vSwitch collegandolo ad un controller esterno e creare gli host aggiungendo le interfacce virtual ethernet.

Il controller invece è un Ryu controller. Questo permette di aggiungere le flow entries agli switch e dispone di una rest API pubblica (https://ryu.readthedocs.io/en/latest/app/ofctl_rest.html) per collezionare/applicare flow entries agli switch.

Quest'ultima verrà utilizzata dall'applicazione per interfacciarsi con Open vSwitch e classificare il traffico utilizzando il modulo python scikit-learn.

3.2.1 Mininet

Mininet è un emulatore di reti che permette la creazione di host, switch, routers, controllers e link virtuali.

Fa uso della *lightweight virtualization* per rendere l'intero sistema visibile come una rete completa, i cui componenti vengono eseguiti in uno stesso kernel e user space.

Attraverso i *network namespaces*, disponibili dalla versione 2.2.26 del kernel Linux, ogni host ha un proprio stack di rete, quindi delle proprie interfacce di rete, una propria ARP table e routing table. La feature della *process-based virtualization* di Linux, che permette la separazione dei processi dei diversi host, invece, non è implementata in Mininet.

Gli switch di rete virtuali possono essere della tipologia di default, il Linux Bridge, oppure Open vSwitch.

Attraverso Mininet, è possibile effettuare il deployment di una Software-Defined Network. Infatti, gli switch appartenenti alla seconda categoria (Open vSwitch) sono programmabili da controller SDN.

Per quanto riguarda la creazione delle reti, Mininet fornisce un'API in Python, dove è possibile orchestrare la rete, definendo la sua topologia, e permette la sperimentazione di quest'ultima, fornendo all'utente una CLI e dei semplici comandi per testare la connettività.

In breve, i vantaggi nell'utilizzo di Mininet sono:

- È veloce e scalabile: il boot avviene in pochi istanti e supporta migliaia di host;
- Si possono creare topologie custom;
- Si possono lanciare programmi in ciascun host;

- Il packet forwarding è customizzabile attraverso l'utilizzo del protocollo Openflow ed il design della rete SDN può essere facilmente trasferito a switch hardware Openflow;

- Facilmente replicabile e facile da utilizzare.

Mininet presenta anche degli svantaggi. Per esempio:

- Risorse limitate: lanciare una topologia in un singolo host fisico significa spartire le risorse tra le diverse componenti di rete, quindi il rate per il forwarding dei pacchetti negli switch è spartito, come la risorse CPU tra i diversi host virtuali;

- Gli host condividono lo stesso process space (PID) e file system dell'host fisico.

Creazione di topologie di rete

Attraverso poche righe di codice in Python, è possibile creare una topologia di rete.

A seguire viene mostrato il codice di una semplice topologia, composta da n host collegati ad uno stesso switch.

Listato 3.1: Esempio di topologia

```
1  #!/user/bin/python
2  from mininet.topo import Topo
3  from mininet.net import Mininet
4  from mininet.util import dumpNodeConnections
5  from mininet.log import setLogLevel
6  import sys
7
8  class MyTopology(Topo):
9      "Single switch connected to n hosts."
10     def build(self, n=3):
11         # Adding switch
12         switch = self.addSwitch('sw1')
13         # Adding hosts, connecting them to the switch
14         for h in range(n):
15             host = self.addHost('h%s' % (h + 1))
16             self.addLink(host, switch)
17
18 def test(n=3):
19     "Create and test a simple network"
```

```
20      topo = MyTopology(n)
21      net = Mininet(topo)
22      net.start()
23      print("Dumping host connections")
24      dumpNodeConnections(net.hosts)
25      print("Testing network connectivity")
26      net.pingAll()
27      net.stop()
28 
29  if __name__ == '__main__':
30      # If specified, read the number of hosts from
           arguments
31      if len(sys.argv) > 1:
32          n = int(sys.argv[1])
33      else:
34          # Standard number of hosts
35          n = 3
36      # Tell mininet to print useful information
37      setLogLevel('info')
38      test(n)
```

Le classi e i metodi principali sono:

- `Topo`: la classe base di cui bisogna fare l'override del metodo `build()`. Attraverso quest'ultimo viene creato un template, utilizzato da `Mininet` per l'istanziazione vera e propria della rete, composto dai nomi delle componenti di rete e da un database con le informazioni di configurazione;

- `Mininet`: la classe principale il cui scopo è creare e gestire la rete;

- `addSwitch()`, `addHost()`, `addLink()`: tre metodi per rispettivamente aggiungere uno switch, un host, un collegamento bidirezionale tra switch e host alla topologia;

3.2.2 Ryu Controller

Ryu è un component-based framework per il SDN.

Fornisce un'API in Python per creare applicazioni di gestione e controllo della rete e supporta vari protocolli per l'interfacciamento con gli switch, tra cui: OpenFlow, Netconf, OF-config, etc..

Composizione di un App

Un'applicazione Ryu è single-threaded.

Mantenendo una coda FIFO, il thread ha lo scopo di gestire gli eventi, che vengono processati in ordine chiamando il rispettivo handler: proprio per questo motivo la logica del Controller è basata sulla programmazione ad eventi.

Poichè l'event handler viene chiamato nel context del thread che processa gli eventi, se la funzione è bloccante degli eventi potrebbero non essere catturati e andrebbero persi.

L'applicazione è definita da una classe Python che:

- Estende la classe `appManager.RyuApp`;
- Contiene i propri campi e metodi;
- Definisce degli handler per rispondere agli eventi interessati.

Per ogni tipologia di evento che si vuole gestire è necessario definire una callback, registrata attraverso un decorator. La sintassi di base è la seguente:

```
@set_ev_cls(OpenFlowEvent, DISPATCHER)
```

dove il primo parametro descrive il tipo di evento: per convenzione gli eventi Openflow vengono indicati con `ofp_event.EventOFPxxx` dove `xxx` rappresenta il messaggio Openflow. Ad esempio: `ofp_event.EventOFPPacketIn` per messaggi Openflow di tipo Packet In.

Per quanto riguarda il secondo parametro, `DISPATCHER`, questo indica in che specifica fase di negoziazione tra Controller e Switch l'evento deve essere generato:

1. `HANDSHAKE_DISPATCHER`: invio e attesa di messaggi di tipo HELLO;
2. `CONFIG_DISPATCHER`: versione negoziata ed invio messaggio features-request effettuato;
3. `MAIN_DISPATCHER`: feature dello switch ricevute, invio messaggio set-config effettuato;
4. `DEAD_DISPATCHER`: disconnessione dallo switch, anche per casi di errore.

3.2.3 scikit-learn

scikit-learn è un modulo Python open source per il machine learning distribuito sotto licenza BSD a 3 clausole.

Il suo progetto è stato sviluppato da David Cournapeau ed attualmente è mantenuto da un team di volontari.

Contiene vari algoritmi di classificazione, regressione e clustering ed utilizza NumPy per le operazioni di algebra lineare ed array di elevate prestazioni.

Le classi che consentono di eseguire la classificazione binaria o multiclasse su un dataset sono: `SVC`, `NuSVC`, `LinearSVC` [12].

Di seguito viene mostrato uno script che carica un dataset sul cancro. I parametri corrispondono a caratteristiche del nucleo di una cellula campionata a cui viene associata la tipologia di tumore (classe): benigno o maligno.

Al classificatore viene passato come parametro la funzione kernel, in questo caso lineare, ma è possibile associarne di altre tipologie tra quelle descritte precedentemente ('polynomial', 'rfb', 'sigmoid') ed è possibile specificare il parametro di regolarizzazione C, il cui valore di default è 1.

Infine vengono previste le classi partendo da un test set e viene calcolata l'accuratezza.

Listato 3.2: scikit-learn SVM

```
1  from sklearn import datasets
2  from sklearn.svm import SVC
3  from sklearn.model_selection import train_test_split
4  from sklearn import metrics
5
6  # Load cancer dataset
7  cancer = datasets.load_breast_cancer()
8
9  # Print the feature and target names
10 print("Features: ", cancer.feature_names, "\nLabels: ",
       cancer.target_names)
11
12 # Get values
13 X = cancer.data
14 y = cancer.target
15
16 # Split dataset into training set and test set
17 X_train, X_test, y_train, y_test = train_test_split(X, y
       , test_size=0.25, random_state=0)
18
19 """ Create classifier with:
20     - linear kernel
21     - C = 1 (default) as regularization parameter
22 """
23 clf = SVC(kernel='linear', random_state=0)
24
```

```
# Train the model
clf.fit(X_train, y_train)

# Predict test set and calculate accuracy
y_pred = clf.predict(X_test)
print("Accuracy: ", metrics.accuracy_score(y_test,
    y_pred))
```

3.3 Progetto finale

Dopo aver parlato delle tecnologie utilizzate, mi occuperò ora di illustrare brevemente le varie parti del sistema, descrivendo il loro ruolo:

3.3.1 Rete virtuale

La rete virtuale Mininet ha come ruolo appunto la simulazione della rete. Ogni host è collegato allo switch Openflow, in questo caso un Open vSwitch attraverso cui viene generato del traffico classificabile come "normale". Sempre servendoci degli host in questa rete verrà performato l'attacco e verificato se l'algoritmo di mitigazione funziona correttamente.

3.3.2 Controller Ryu

Il controller Ryu sviluppato funge da switch. Comunica con Open vSwitch e contiene il decorator per pacchetti Packet In allo scopo di installare flow entries negli switch.

3.3.3 App Python

L'applicazione in Python si interfaccia con l'API REST ofct_rest, rileva e mitiga gli attacchi DDoS e mostra i grafici dei parametri della SVM al variare del traffico di rete.

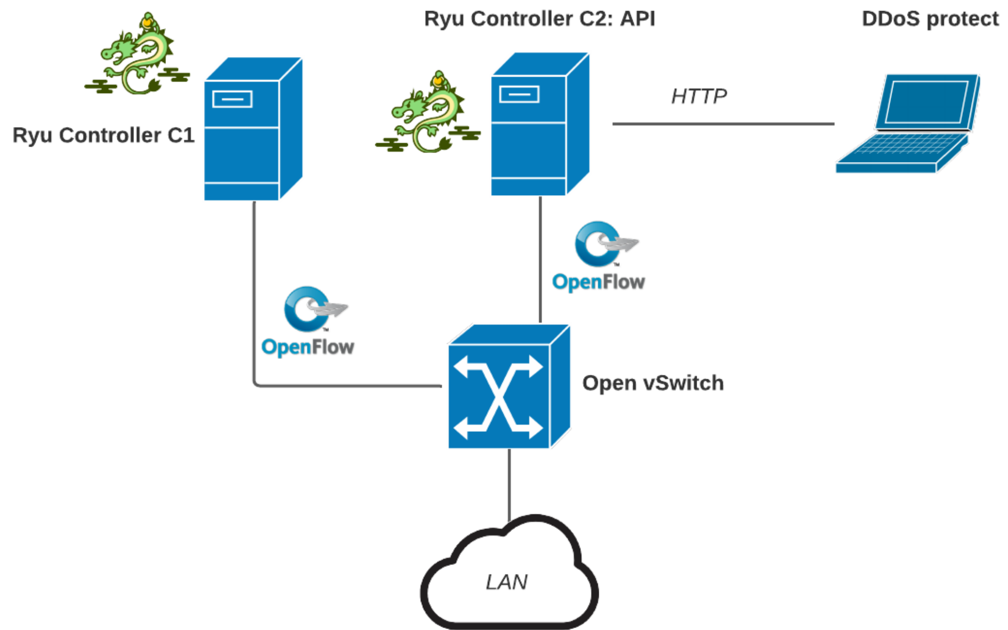

Figura 3.1: Architettura semplificata

Capitolo 4

Design

In questo capitolo viene descritto il design del progetto. Partendo dai parametri scelti per l'addestramento della Support Vector Machine, si mostra successivamente la topologia di rete adottata e l'architettura dell'applicazione.

4.1 Parametri di addestramento della Support Vector Machine

L'addestramento della Support Vector Machine è effettuato sulla base di alcuni parametri che assumono valori diversi in situazioni di traffico normale ed anormale.

I parametri utilizzati per l'addestramento sono [13]:

4.1.1 Velocità degli indirizzi IP sorgente

$$SSIP = \frac{\sum IP_{src}}{T} \tag{4.1}$$

dove $\sum IP_{src}$ è il numero di sorgenti IP e T è il periodo di campionamento.

$SSIP$ (Speed of source IP) rappresenta la velocità degli indirizzi IP sorgente nell'unità di tempo. In situazioni di attacco, specialmente quando l'IP sorgente è generato da un pool random, questo parametro aumenta rapidamente.

4.1.2 Deviazione standard del numero di pacchetti nelle flow entries

$$SDFP = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\text{packets}_i - \text{mean_packets})} \tag{4.2}$$

dove $\text{mean_packets} = \frac{1}{N}\sum_{i=1}^{N} \text{packets}_i$ è il numero medio di pacchetti transitati nelle flow entries i nel periodo T, N è il numero di flow entries.

$SDFP$ (Standard deviation of flow packets) rappresenta la deviazione standard del numero di pacchetti nelle flow entries. Quando l'attacco è generato utilizzando sorgenti IP random, il numero di pacchetti transitati presso la flow entry precedentemente creata dal controller è pari a 0, di conseguenza questo parametro diminuisce velocemente.

4.1.3 Deviazione standard del numero di bytes nelle flow entries

$$SDFB = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\text{bytes}_i - \text{mean_bytes})} \tag{4.3}$$

dove $\text{mean_bytes} = \frac{1}{N}\sum_{i=1}^{N} \text{bytes}_i$ è il numero medio di bytes transitati nelle flow entries i nel periodo T, N è il numero di flow entries.

$SDFB$ (Standard deviation of flow bytes) rappresenta la deviazione standard del numero di bytes nelle flow entries. Quando un attacco è in corso, i pacchetti "malevoli" hanno generalmente minori dimensioni rispetto a quelli "normali": questo permette all'attaccante di inviarne un maggior numero nell'unità di tempo. Detto ciò $SDFB$ diminuisce in caso di attacco.

4.1.4 Velocità delle flow entries

$$SFE = \frac{N}{T} \tag{4.4}$$

dove N è il numero di flow entries, T è il periodo di campionamento.

SFE (Speed of flow entries) rappresenta la velocità delle flow entries nell'unità di tempo. In una situazione di attacco, il numero delle flow entries aumenta drasticamente, di conseguenza SFE.

4.1.5 Rapporto tra flow interattivi e flow totali

$$RPF = \frac{Int_IP}{N} \tag{4.5}$$

dove:

- Int_IP è il numero di flow interattive, cioè le flow entries i dove esiste una flow entry j t.c.:

$$Src_IP_i = Dst_IP_j$$
$$Dst_IP_i = Src_IP_j$$

- N è il numero totale di flow entries

RPF (Ratio of pair-flow) è il rapporto tra flow interattivi e flow totali.

In normali condizioni di traffico, l'IP sorgente dell'i-esimo flow è uguale all'IP destinazione del j-esimo e l'IP destinazione dell'i-esimo uguale all'IP sorgente del j-esimo, perciò $RPF = 1$.

In situazioni di attacco, la vittima si trova bombardata di pacchetti alla quale non riesce a rispondere. Aumentando N e rimanendo costante il numero di Int_IP RPF diminuisce bruscamente.

Utilizzando questi parametri, la SVM viene addestrata allo scopo di classificare il traffico come normale o anomalo.

4.2 Topologia di rete

La topologia è costituita da:

- Rete virtuale Mininet composta da 1 switch, 25 hosts ed 1 host target (utilizzato per l'addestramento) connessi nella rete 137.204.0.0/24.

  Questa viene lanciata in una VM per limitare le risorse degli host e dello switch. Quando l'attacco è in esecuzione stiamo facendo un flood della rete perciò viene utilizzata un'elevata percentuale di CPU. Di conseguenza, se non utilizzassimo una VM, la situazione porterebbe ad una degradazione delle performance dei controller, in particolar modo verrebbe colpita l'API che non riuscirebbe a rispondere alle richieste HTTP, e così la rilevazione dell'attacco non andrebbe a buon fine.

  La VM che ho utilizzato dispone di 1 core di CPU e di 4 MB di RAM ed è collegata alla rete in modalità bridged (nel mio caso con IP: 192.168.1.39).

  Lo switch è collegato ai due controller remoti C1 e C2 in modalità EQUAL (default).

- Controller C1. Questo controller funge da switch e ad un nuovo pacchetto Packet-In installa un flow entry basato su IP sorgente ed IP destinazione. In questo modo è possibile collezionare le statistiche dei flow entries dall'API.

- Controller C2. C2 è di fatto la REST API `ofctl_rest` che si interfaccia con lo switch.

La peculiarità sta nel fatto che `ofctl_rest` costituisce un unico controller. Non viene cioè eseguita insieme a C1 (in questo caso avremmo un solo controller) ma in un nuovo processo `ryu-manager`.

Infatti, considerato che `ryu-manager` utilizza un solo thread, se avessimo un unico controller in caso di attacco l'elevato consumo di CPU da parte di C1 risulterebbe in un'API non reattiva alle richieste HTTP dell'applicazione.

In aggiunta, in C2 viene configurato lo switch con un messaggio asincrono con l'indicazione di non ricevere messaggi da quest'ultimo (es. messaggi Packet-In), poichè C2 installa nuovi flow tramite l'applicazione. Il vantaggio principale è che il canale risulterà libero dai messaggi Packet-In quando la vittima è sotto attacco.

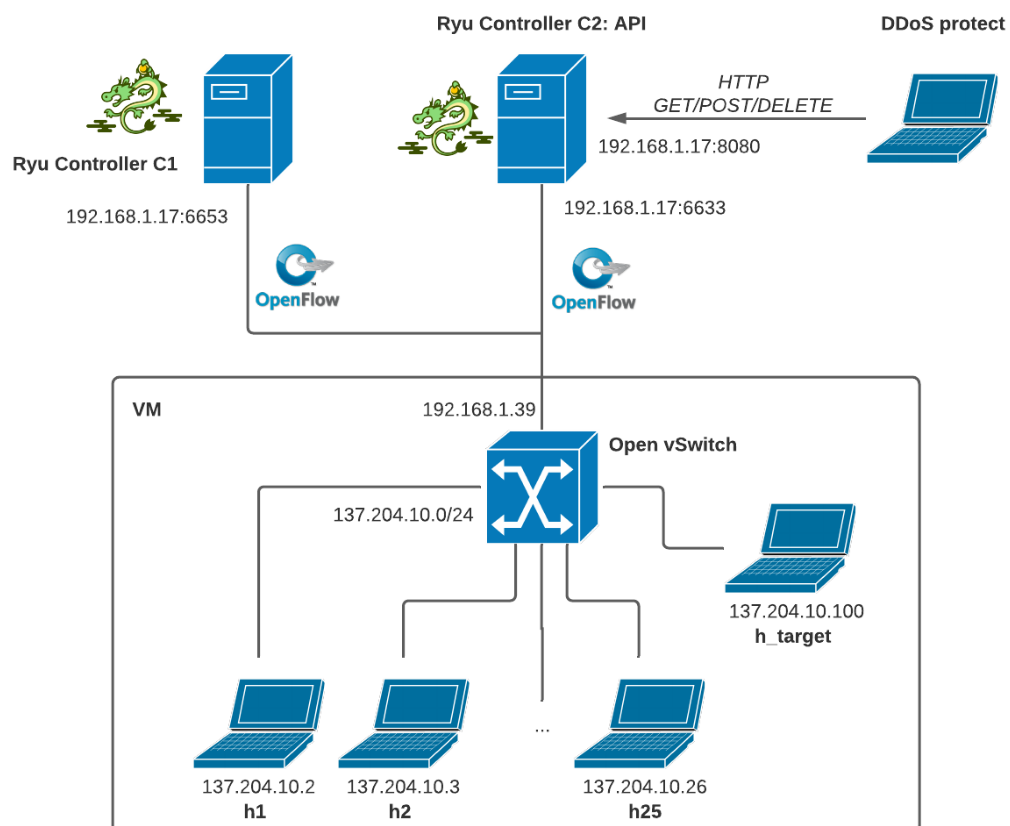

Figura 4.1: Architettura dettagliata

4.3 Architettura dell'applicazione

Per la realizzazione dell'applicazione è stato utilizzato il pattern MVC (Model-View-Controller) nella quale il Controller aggiorna la View servendosi del Model.

Di seguito vengono illustrate le varie componenti:

- **Model**. Contiene le classi tramite le quali è possibile modellare le flow entries e costruire oggetti di aggiornamento della View;
- **View**. Contiene gli oggetti per plottare i grafici e metodi per aggiornarli quando comunicato dal controller.

  È composta da 2 threads. Il primo rappresenta il `mainloop()` di Tkinter mentre il secondo viene runnato per ricevere nuovi dati dal controller. In particolare i nuovi dati sono aggiunti dal controller in una coda bloccante condivisa;
- **Controller**. Contiene tutta la logica dell'applicazione, dalla lettura delle flow entries al calcolo dei parametri della SVM e alla mitigazione di un attacco. È composto da 3 classi: `DDoSControllerThread`, `FeaturesController` e `SVMController`.

  La prima (`DDoSControllerThread`) è di fatto un thread il cui loop è implementato usando una Async Final State Machine (Async FSM).

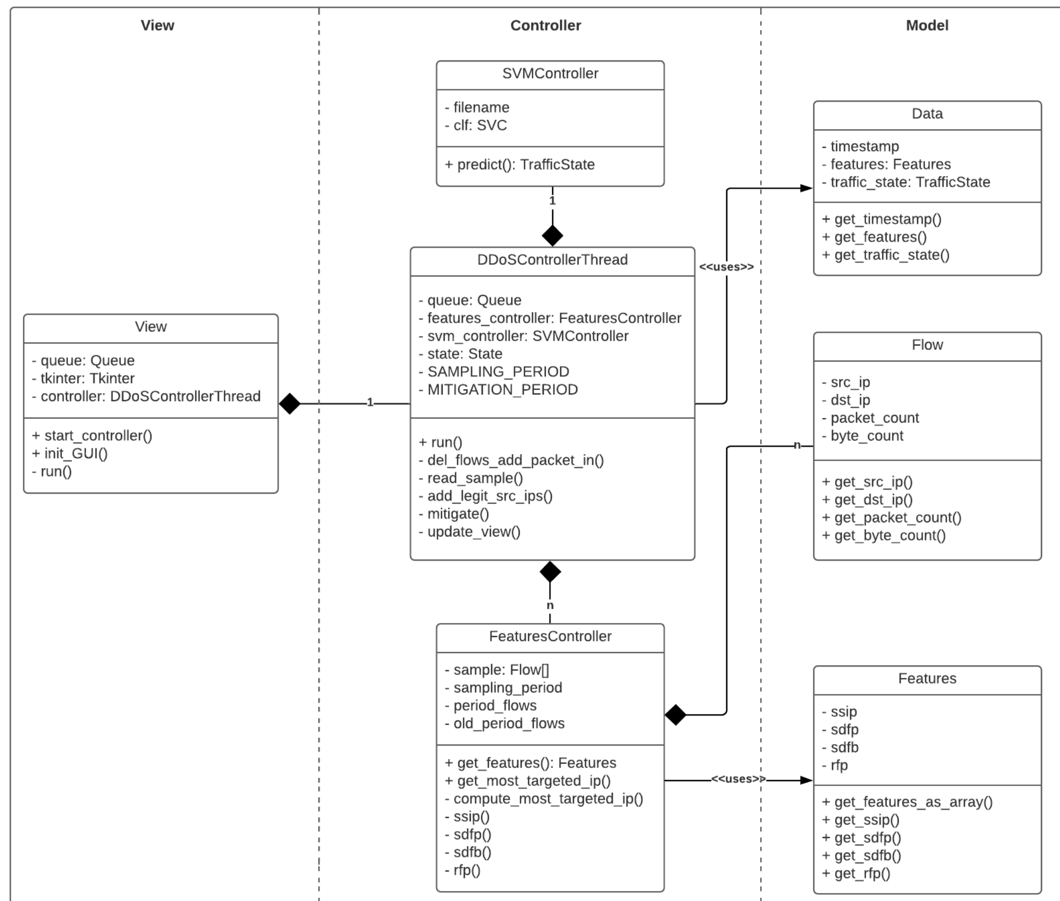

Figura 4.2: Schema UML dell'architettura

4.3.1 Modellazione di DDoSControllerThread

La maggior parte della logica dell'applicazione è contenuta in questa classe. Questa esegue le chiamate HTTP all'API `ofctl_rest`, determina lo stato del traffico servendosi di `SVMController` ed applica la mitigazione quando necessario.

Inoltre, utilizzando `FeaturesController` aggiorna il Model e la View attraverso la coda bloccante.

Il metodo `run` è implementato usando una Async FSM e si identificano 3 stati principali derivati dal tipo di traffico transitante nella rete:

1. `UNCERTAIN`. La macchina a stati finiti si trova in questo stato nell'arco temporale che intercorre tra due campionamenti. Infatti in questo intervallo non è possibile stabilire che tipo di traffico è presente nella rete.

Le azioni che svolge sono:

1.1. pulisce le flow entries installando la regola per i messaggi Packet In;

1.2. esegue la funzione di `sleep` per `SAMPLING_PERIOD` secondi;

1.3. determina lo stato successivo della FSM campionando le flow entries, determinando i parametri della SVM e facendo il `predict()`.

2. `NORMAL`. Quando ci troviamo in questo stato i valori dei parametri della SVM e lo stato del traffico devono essere graficati perciò vengono inseriti dati nella `Queue`.

   In aggiunta, viene aggiornato l'array degli indirizzi IP sorgente "legittimi" relativamente alle flow entries che hanno come IP destinazione l'host correntemente più preso di mira.

   Una volta terminate queste due funzioni lo stato ritorna in `UNCERTAIN`

3. `ANOMALOUS`. Quando la Support Vector Machine classifica il traffico come malevolo lo stato della FSM è `ANOMALOUS`.

   In questo stato la FSM:

   3.1. comunica alla View di graficare i parametri della SVM e lo stato del traffico;

   3.2. mitiga l'attacco eliminando tutte le flow entries ed installandone una che fa match con l'IP dell'host target (IP destinazione più preso di mira), specificando come azione: DROP;

   3.3. installa flow entries per gli indirizzi IP legittimi che comunicavano con tale host per permettere a loro di continuare a scambiare pacchetti;

   3.4. mantiene la flow entry DROP per `MITIGATION_PERIOD`;

   3.5. cambia lo stato in `UNCERTAIN` per campionare la rete e verificare se l'attacco si è estinto.

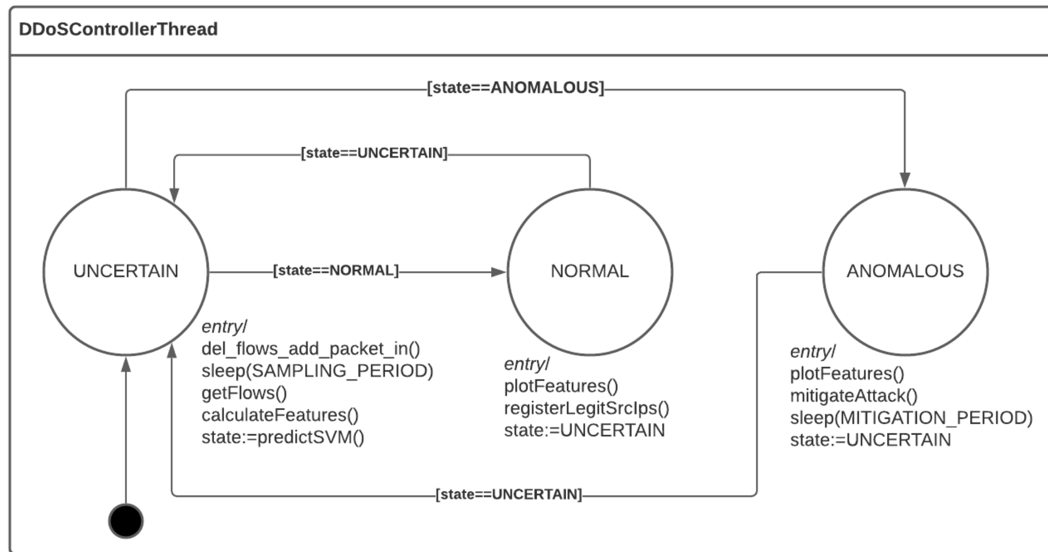

Figura 4.3: Diagramma a stati di `DDoSControllerThread`

4.3.2 Modellazione di FeaturesController

La classe FeaturesController ha lo scopo di calcolare i parametri utilizzati per l'addestramento partendo da campionamenti effettuati da `DDoSControllerThread`.

Come vedremo, diversamente dal processo di addestramento della SVM, questa classe non elimina tutte le flow entries ad ogni periodo `SAMPLING_PERIOD`, ma calcola i parametri sottraendo le informazioni ottenute con quelle del campionamento precedente.

Questo perchè se nella realtà eliminassimo ad ogni `SAMPLING_PERIOD` secondi le flow entries il delay per ogni connessione aumenterebbe, poichè tempo verrebbe sprecato per comunicare di nuovo con il controller e nell'installazione della flow entry. In più, se l'host target ha normalmente un numero elevato di connessioni, il canale di comunicazione con il controller risulterebbe appesantito.

Per il calcolo dei parametri viene mantenuta una struttura dati `period_flows` dove vengono memorizzate le flow entries nelle quali è transitato del traffico dal periodo t a $t+$ `SAMPLING_PERIOD`. Se del traffico non è transitato allora la flow entry non viene considerata per il calcolo dei parametri. Il calcolo dei parametri è così effettuato:

- SSIP. Dato l'IP destinazione più preso di mira viene calcolato il numero di sorgenti IP da `period_flows` e viene diviso per `SAMPLING_PERIOD`;
- SDFP. Nel momento in cui `period_flows` è aggiornato si calcola il numero di pacchetti transitati presso ciascuna connessione attiva facendo la differenza con il campionamento precedente;
- SDFB. Uguale a SDFP ma sui bytes transitati;
- SFE. Lunghezza di `period_flows` / `SAMPLING_PERIOD`;
- RFP. Rapporto dei flow interattivi di `period_flows` con la sua lunghezza.

Capitolo 5

Sviluppo

Terminata l'analisi e il design mi occuperò ora di illustrare i vari script e le classi che compongono il sistema, partendo dalla realizzazione della topologia all'implementazione dell'applicazione. Il codice completo è disponibile nella repository pubblica https://github.com/w-disaster/svm-ddos-sdn.

5.1 Realizzazione della topologia di rete

Come detto precedentemente la topologia virtuale è implementata utilizzando Mininet. Di seguito viene mostrata la classe utilizzata per l'addestramento dove viene assegnato un IP diverso all'host target. La topologia viene eseguita con il comando:

```
sudo mn --custom mn_ddos_topology.py --switch ovsk \
--controller=remote,ip=192.168.1.17:6653 \
--controller=remote,ip=192.168.1.17:6633 --topo ddostopo
```

Listato 5.1: DDoS experimentation topology

```
from mininet.topo import Topo

N_HOSTS = 25
DPID = "1"
TARGET_IP = "137.204.10.100"

class DDoSTopo(Topo):
    "DDoS experimentation topology"

    def build(self):
        # Add one switch
```

```
12          s0 = self.addSwitch("s0", dpid=DPID)
13
14          # Add hosts and connect them to the switch
15          for i in range(1, N_HOSTS + 1):
16              h = self.addHost("h" + str(i), ip="
                    137.204.10." + str(i + 1) + "/24")
17              self.addLink(h, s0, bw=10, delay="10ms")
18
19          # Add target host
20          h_target = self.addHost("h_target", ip=TARGET_IP
                + "/24")
21          self.addLink(h_target, s0, bw=10, delay="10ms")
22
23  topos = { 'ddostopo': ( lambda: DDoSTopo() ) }
```

Diversamente dalla topologia che è attiva nel guest (IP 192.168.1.39), i controller vengono eseguiti nell'host. Il controller C1 è in ascolto sulla porta 6653 (`ryu-manager --ofp-tcp-listen-port 6653 c1.py`) e si comporta come un normale switch, installando una flow entry che fa match in base alla coppia IP sorgente, IP destinazione.

A seguire il decorator per gli eventi di tipo Packet In.

Listato 5.2: Controller C1

```
1   @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
2   def _packet_in_handler(self, ev):
3       # If you hit this you might want to increase
4       # the "miss_send_length" of your switch
5       if ev.msg.msg_len < ev.msg.total_len:
6           self.logger.debug("packet truncated: only %s of
                %s bytes", ev.msg.msg_len, ev.msg.total_len)
7       msg = ev.msg
8       datapath = msg.datapath
9       ofproto = datapath.ofproto
10      parser = datapath.ofproto_parser
11      in_port = msg.match['in_port']
12
13      pkt = packet.Packet(msg.data)
14      eth = pkt.get_protocols(ethernet.ethernet)[0]
15
16      if eth.ethertype == ether_types.ETH_TYPE_LLDP:
```

```
17            # ignore lldp packet
18            return
19        dl_dst = eth.dst
20        dl_src = eth.src
21
22        dpid = datapath.id
23        self.mac_to_port.setdefault(dpid, {})
24
25        self.logger.info("packet in %s %s %s %s", dpid,
             dl_src, dl_dst, in_port)
26        # learn a mac address to avoid FLOOD next time.
27        self.mac_to_port[dpid][dl_src] = in_port
28
29        if dl_dst in self.mac_to_port[dpid]:
30            out_port = self.mac_to_port[dpid][dl_dst]
31        else:
32            out_port = ofproto.OFPP_FLOOD
33
34
35        actions = [parser.OFPActionOutput(out_port)]
36
37
38        # install a flow to avoid packet_in next time
39        if out_port != ofproto.OFPP_FLOOD:
40
41            # check IP protocol
42            if eth.ethertype == ether_types.ETH_TYPE_IP:
43                ip = pkt.get_protocol(ipv4.ipv4)
44                src_ip = ip.src
45                dst_ip = ip.dst
46                protocol = ip.proto
47
48                # match all packets with src_ip as source IP
                     , dst_ip as
49                # destination IP
50                match = parser.OFPMatch(eth_type=ether_types
                     .ETH_TYPE_IP,
51                        ipv4_src=src_ip,
52                        ipv4_dst=dst_ip,
53                        eth_src=dl_src,
```

```
54                          eth_dst=dl_dst,
55                          in_port=in_port,
56                          ip_proto=protocol)
57
58              # verify if we have a valid buffer_id, if
                   yes avoid to send both
59              # add flow
60              if msg.buffer_id != ofproto.OFP_NO_BUFFER:
61                  self.add_flow(datapath, 1, match,
                       actions, msg.buffer_id)
62                  return
63              else:
64                  self.add_flow(datapath, 1, match,
                       actions)
65
66      data = None
67      if msg.buffer_id == ofproto.OFP_NO_BUFFER:
68          data = msg.data
69
70      # packet out
71      out = parser.OFPPacketOut(datapath=datapath,
           buffer_id=msg.buffer_id,
72                                   in_port=in_port, actions=
                                        actions, data=data)
73      datapath.send_msg(out)
```

Per quando riguarda il controller C2, questo è composto da due file. Il primo è `ofctl_rest.py` ed il secondo `c2.py`. Attraverso quest'ultimo viene inviato un messaggio asincrono allo switch facendo uso del metodo `OFPSetAsync` per stabilire che tipo di messaggi vuole ricevere dallo switch.

Nel codice a seguire ogni maschera ha valore 0, perciò C2 non riceve nessun tipo di messaggio dallo switch (è possibile verificarlo utilizzando Wireshark).

C2 viene lanciato insieme a `ofctl_rest` con il comando:
`ryu-manager --ofp-tcp-listen-port 6633 c2.py ofctl_rest.py`

Listato 5.3: Controller C2

```
1 class C2(app_manager.RyuApp):
2     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
3
4     def __init__(self, *args, **kwargs):
```

```
5         super(C2, self).__init__(*args, **kwargs)
6 
7     @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
       CONFIG_DISPATCHER)
8     def switch_features_handler(self, ev):
9         datapath = ev.msg.datapath
10        ofproto = datapath.ofproto
11        parser = datapath.ofproto_parser
12
13        # Do not receive Packet In messages
14        packet_in_mask = (1 << ofproto.OFPR_NO_MATCH
15                & 1 << ofproto.OFPR_ACTION
16                & 1 << ofproto.OFPR_INVALID_TTL)
17
18        # Do not receive port status messages
19        port_status_mask = (1 << ofproto.OFPPR_ADD
20                        & 1 << ofproto.OFPPR_DELETE
21                        & 1 << ofproto.OFPPR_MODIFY)
22
23        # Do not receive flow removed messages
24        flow_removed_mask = (1 << ofproto.
           OFPRR_IDLE_TIMEOUT
25                         & 1 << ofproto.
                              OFPRR_HARD_TIMEOUT
26                         & 1 << ofproto.OFPRR_DELETE)
27
28        # Build request and send msg
29        req = parser.OFPSetAsync(datapath, [
           packet_in_mask, 0],
30                 [port_status_mask, 0], [
                    flow_removed_mask, 0])
31        datapath.send_msg(req)
```

5.2 Addestramento della SVM

Ora che l'intera topologia è messa a punto l'addestramento della SVM può iniziare.

5.2.1 Simulazione di traffico "normale"

Prima di tutto viene simulato traffico "normale", classificabile con la classe 0. Per automatizzarne la generazione vengono resi visibili i namespaces degli host di Mininet e successivamente eseguito il seguente script in ognuno di essi:

Listato 5.4: Normal traffic

```
#!/bin/bash
# If no argument supplied
if [ $# -eq 0 ]; then
    echo "usage: sudo bash traffic.sh <target_ip>"
else
    TARGET_IP="$@"
    while true; do
        N_PACKETS=$(shuf -i 10-20 -n 1)
        N_BYTES=$(shuf -i 150-200 -n 1)
        PAUSE=$(shuf -i 1-10 -n 1)
        hping3 -c $N_PACKETS -d $N_BYTES --icmp "
            $TARGET_IP"
        sleep $PAUSE
    done
fi
```

Il traffico è generato dal comando `hping3` dove è possibile specificare sia quanti pacchetti devono essere inviati che la loro dimensione in bytes. Ad ogni ciclo questi due parametri vengono scelti randomicamente tra un range di valori insieme al periodo dove l'host non comunica con il target.

Per simulare i picchi di traffico è stato utilizzato un altro script `traffic_peak.sh` dove al comando `hping3` viene aggiunta la direttiva `--fast`.

Nel seguente script viene mostrato come vengono resi visibili i namespaces ed eseguiti i comandi in ciascun host Mininet.

Listato 5.5: Traffic generator

```
#!/bin/bash
# If no argument supplied
if [ $# -eq 0 ]; then
  echo "usage: sudo bash gen_traffic.sh <target_ip>"
else
    TARGET_IP="$@"
    PIDS=$(ps aux | grep "mininet:h" | cut -c5-16 | tail
        -n+2)
```

```
8       N_PIDS=$(echo $PIDS | wc -w)
9
10      mkdir -p /var/run/netns
11
12      trap "exit" INT TERM ERR
13      trap "rm -rf /var/run/netns && kill 0" EXIT
14
15      I=1
16      while read PID; do
17          ln -sf /proc/$PID/ns/net /var/run/netns/$PID
18          if [ $I -ge $(($N_PIDS / 2)) ]; then
19          ip netns exec $PID bash ./traffic.sh $TARGET_IP
                &
20          else
21              ip netns exec $PID bash ./traffic_peak.sh
                    $TARGET_IP &
22          fi
23          I=$((I+1))
24      done <<< "$PIDS"
25      wait
26  fi
```

5.2.2 Simulazione di traffico "anomalo"

L'attacco DDoS è simulato anche questo attraverso il comando `hping3` e in questo caso si identifica come un attacco di tipo ICMP flood. Gli indirizzi IP sorgenti vengono scelti in modo random.

```
hping3 --rand-source --flood --icmp <target_ip>
```

Facendo uso di questo comando, oltre ad essere colpito l'host target anche il controller C1 risulta bombardato di pacchetti Packet In, poichè nessun pacchetto fa match con flow entries installate nello switch. Pertanto, se l'attacco è abbastanza potente, potrebbe portare alla saturazione del Control Channel e ad un esaurimento delle risorse del controller 2.5.7.

La soluzione di mitigazione adottata gestisce anche questo tipo di problematica, installando infatti la DROP flow entry il Control Channel di C1 risulterà libero.

5.2.3 Calcolo dei parametri

Lo script `train.py` calcola i parametri di traning ogni `SAMPLING_PERIOD` secondi, eliminando successivamente tutte le flow entries ed installando la regola per i messaggi Packet In: in questo modo ad ogni periodo è possibile stabilire con esattezza quanti pacchetti e bytes sono passati per ogni flow entry, e di conseguenza tutti i parametri.

Un oggetto di tipo FeaturesCalculator viene istanziato ogni `SAMPLING_PERIOD` secondi, passando nel costruttore il campionamento effettuato tramite l'API.

Listato 5.6: FeaturesCalculator

```
class FeaturesCalculator:
    def __init__(self, sample, target_ip,
       sampling_period):
        self.sample = sample
        self.target_ip = target_ip
        self.sampling_period = sampling_period

    def get_features(self):
        return Features(self.__get_ssip(), self.
           __get_sdfp(), self.__get_sdfb(),
                         self.__get_sfe(), self.__get_rfp
                            ())

    def __get_ssip(self):
        count_src_ips = 0
        for flow in self.sample:
            if flow.get_dst_ip() == self.target_ip:
                count_src_ips += 1
        return count_src_ips / self.sampling_period

    def __get_sdfp(self):
        packet_count = []
        for flow in self.sample:
            packet_count.append(flow.get_packet_count())
        return np.std(packet_count)

    def __get_sdfb(self):
        byte_count = []
        for flow in self.sample:
```

```
27                 byte_count.append(flow.get_byte_count())
28             return np.std(byte_count)
29 
30         def __get_sfe(self):
31             return len(self.sample) / self.sampling_period
32 
33         def __get_rfp(self):
34             n_int_flows = 0
35             for uni_flow in self.sample:
36                 for bi_flow in self.sample:
37                     if uni_flow.get_src_ip() == bi_flow.
                          get_dst_ip() and uni_flow.get_dst_ip
                          () == bi_flow.get_src_ip():
38                         n_int_flows += 1
39             return float(n_int_flows) / len(self.sample)
```

Facendo uso di Tkinter è possibile verificare il comportamento di ciascun parametro graficando i loro valori.

Nelle figure successive è evidente la loro variazione quando l'attacco è in corso rispetto alla situazione di traffico "normale".

Figura 5.1: Variazione dei parametri di addestramento

Il dataset generato è composto da 1500 campionamenti di cui 1000 di traffico "normale" e 500 di traffico "anomalo" poichè in quest'ultimo caso i parametri non variano in modo considerevole. Nella figura seguente è mostrato il set utilizzato per il testing (50% dei campionamenti totali) graficato in 2 dimensioni: deviazione standard del numero di pacchetti, deviazione standard del numero di bytes. La figura è uno zoom di quella originale per evidenziare la separazione delle due classi.

Il testing presenta il 100% di accuratezza.

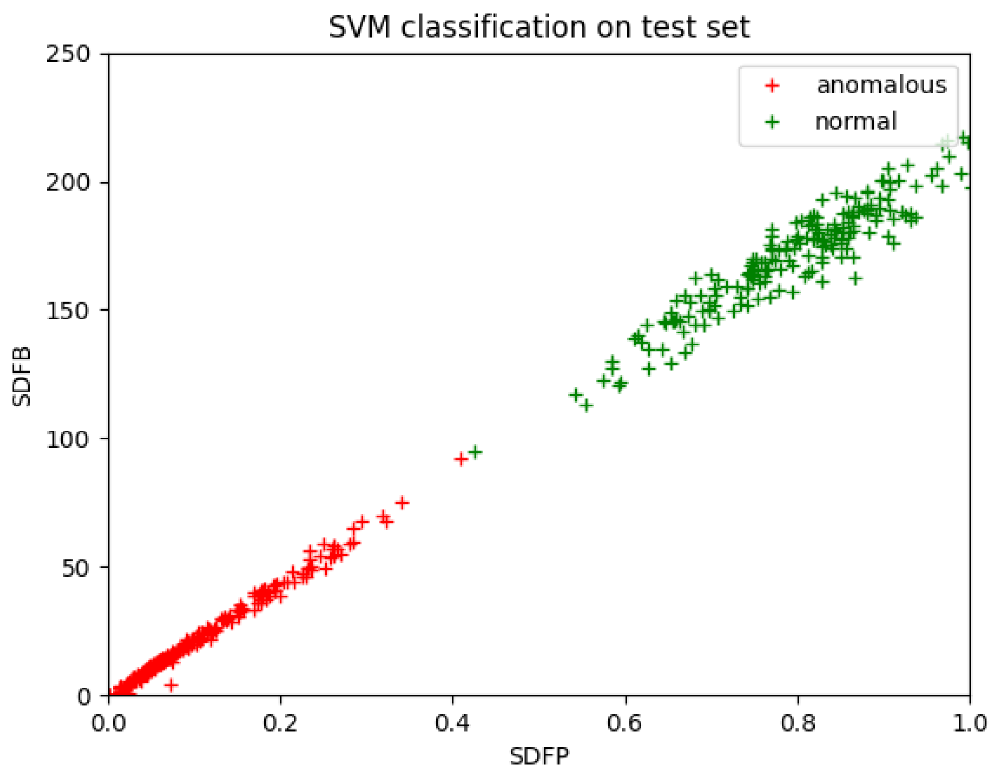

Figura 5.2: Risultati della classificazione

5.3 Sviluppo dell'applicazione

5.3.1 Controllers

Terminata la generazione del dataset e l'addestramento della SVM, l'oggetto `SVC`, cioè il classificatore, viene salvato nella memoria persistente attaverso il modulo `joblib`. Questo ci permetterà di ricostruire l'oggetto senza rieffettuare l'adde-

stramento ad ogni riavvio dell'applicazione. Tale oggetto è contenuto nella classe `SVMController` che all'istanziazione lo carica dalla memoria.

Per quanto riguarda la classe `DDoSControllerThread` a seguire è mostrato il codice della macchina a stati finiti e del metodo `__mitigate`.

Come periodo di campionamento si è scelto `SAMPLING_PERIOD`$= 3s$ per non appesantire eccessivamente lo switch di richieste e viene settato `MITIGATION_PERIOD`$= 30s$ per il periodo di tempo nella quale la DROP flow entry rimane installata, relativamente breve per testare la corretta eliminazione dallo switch.

Il metodo `__mitigate` itera tutte le flow entries relative all'ultimo campionamento e reinstalla quelle che hanno come IP sorgente un IP legittimo ed IP destinazione il target dell'attacco.

Listato 5.7: DDoSControllerThread

```
1  def run(self):
2      while True:
3          if self.state == State.UNCERTAIN:
4              """ Delete all flows, add Packet In flow and
                   create new FeaturesController
5              in order to calculate features from scratch
                 """
6              if self.prec_state == State.UNCERTAIN or
                 self.prec_state == State.ANOMALOUS:
7                  self.del_flows_add_packet_in()
8                  self.features_controller =
                       FeaturesController(SAMPLING_PERIOD)
9              self.prec_state = self.state
10
11             # After waiting SAMPLING_PERIOD sec get flow
                   entries
12             time.sleep(SAMPLING_PERIOD)
13             conn = http.client.HTTPConnection(API_IP,
                 API_PORT)
14             conn.request("GET", "/stats/flow/" + DPID)
15             response = conn.getresponse()
16
17             # Read response
18             if response.status == 200:
19                 sample_as_json = json.loads(response.
                       read())
20             else:
```

```
21                  sample_as_json = []
22
23              if len(sample_as_json) > 0:
24                  # Read flow entries
25                  sample = self.__read_sample(
                       sample_as_json)
26                  # If there are flows based on src ip
27                  if len(sample) > 0:
28                      # Get Features object
29                      self.features_controller.add_sample(
                           sample)
30                      # Get most targeted IP in order to
                           install a drop rule if the next
                           state is ANOMALOUS
31                      self.most_targeted_ip = self.
                           features_controller.
                           get_most_targeted_ip()
32
33                      self.f = self.features_controller.
                           get_features()
34                      # Get features
35                      features = [self.f.get_ssip(), self.
                           f.get_sdfp(), self.f.get_sdfb(),
                           self.f.get_sfe(),
36                                  self.f.get_rfp()]
37
38                      # Predict class
39                      self.state = State(self.
                           svm_controller.predict([features
                           ]).value)
40
41          elif self.state == State.NORMAL:
42              print("traffic is NORMAL")
43              # Update View
44              self.__update_view()
45              # Add src ips as legitimate
46              self.__add_legit_src_ips(sample_as_json)
47              # Change state
48              self.prec_state = self.state
49              self.state = State.UNCERTAIN
```

```
50
51         elif self.state == State.ANOMALOUS:
52             print("traffic is ANOMALOUS")
53             # Update View
54             self.__update_view()
55             # Mitigate attack
56             print("mitigating attack: drop packets with
                  destination IP " + self.most_targeted_ip)
57             self.__mitigate(sample_as_json)
58             time.sleep(MITIGATION_PERIOD)
59             print("delete drop flow rule")
60             self.prec_state = self.state
61             self.state = State.UNCERTAIN
```

Listato 5.8: Mitigazione

```
1  def __mitigate(self, sample_as_json):
2      conn = http.client.HTTPConnection(API_IP, API_PORT)
3      conn.request("DELETE", "/stats/flowentry/clear/" +
          DPID)
4
5      drop_flow = json.dumps({
6          "dpid": DPID,
7          "priority": MEDIUM_PRIORITY,
8          "match": {
9              "ipv4_dst": self.most_targeted_ip,
10             "dl_type": 2048,
11         },
12         # DROP
13         "actions": []
14     })
15     conn = http.client.HTTPConnection(API_IP, API_PORT)
16     conn.request("POST", "/stats/flowentry/add",
          drop_flow)
17
18     packet_in_flow = json.dumps({
19         "dpid": DPID,
20         "table_id": 0,
21         "match": {},
22         "priority": LOW_PRIORITY,
23         "actions": [{
```

```
24              "type": "OUTPUT",
25              "port": "CONTROLLER"
26          }]
27      })
28      conn = http.client.HTTPConnection(API_IP, API_PORT)
29      conn.request("POST", "/stats/flowentry/add",
           packet_in_flow)
30
31      # In conclusion keep connection for legitimate users
32      for k in range(0, len(sample_as_json[DPID]) - 1):
33          if not (sample_as_json[DPID][k]["match"].get('
               nw_src') is None):
34              if sample_as_json[DPID][k]["match"]["nw_src"
                   ] in self.legit_src_ips and \
35                      sample_as_json[DPID][k]["match"]["
                           nw_dst"] == self.most_targeted_ip
                           :
36                  action = sample_as_json[DPID][k]["
                       actions"][0].split(":")
37                  add_flow = json.dumps({
38                      "dpid": DPID,
39                      "priority": HIGH_PRIORITY,
40                      "match": {
41                          "ipv4_src": sample_as_json[DPID
                               ][k]["match"]["nw_src"],
42                          "ipv4_dst": sample_as_json[DPID
                               ][k]["match"]["nw_dst"],
43                          "dl_type": sample_as_json[DPID][
                               k]["match"]["dl_type"],
44                      },
45                      "actions": [
46                          {
47                              "type": action[0],
48                              "port": action[1],
49                          }
50                      ]
51                  })
52                  conn = http.client.HTTPConnection(API_IP
                       , API_PORT)
53                  conn.request("POST", "/stats/flowentry/
```

```
            add", add_flow)
```

5.3.2 View

La View è implementata utilizzando Tkinter, una libreria di Python che consente la creazione di interfacce grafiche. All'interno della finestra principale `Tkinter` viene aggiunta:

- una label, attraverso la quale verrà visualizzato lo stato del traffico;
- 5 sottografici che graficheranno l'andamento dei parametri di addestramento ad ogni campionamento.

Per far si che i grafici si aggiornino, come specificato nel capitolo 4 viene mantenuto un thread che ad ogni evento (cioè messaggio da parte del controller) aggiorna i dati correnti li grafica.

A seguire le immagini della GUI.

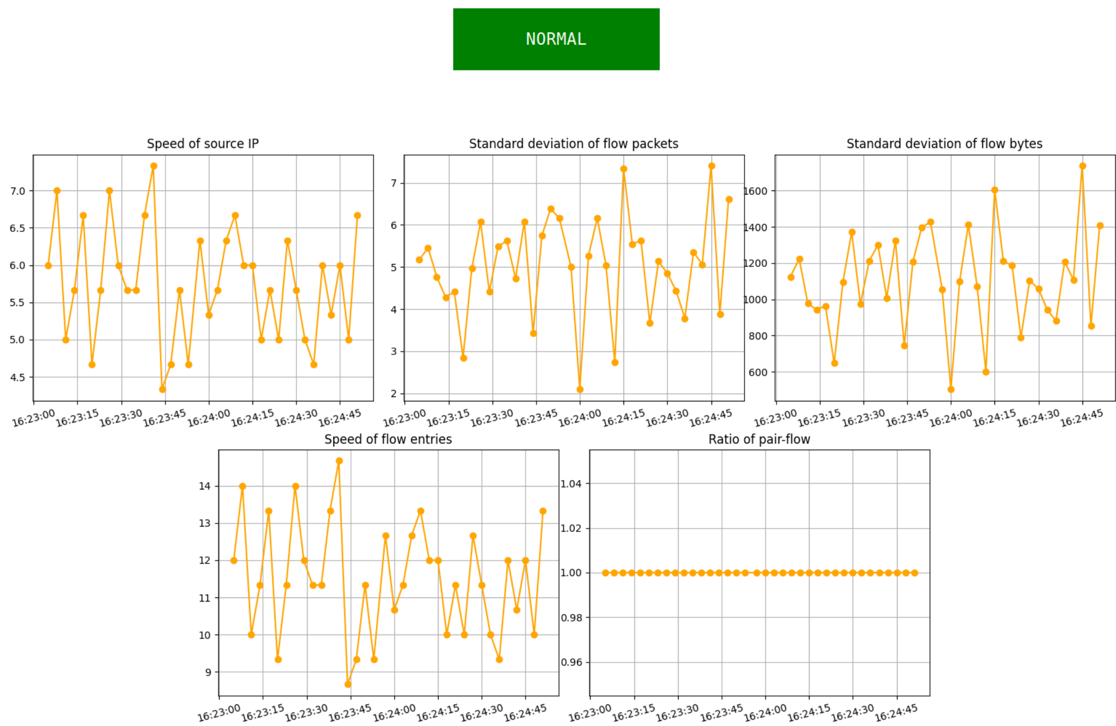

Figura 5.3: Traffico "normale"

Figura 5.4: Traffico "anomalo"

Capitolo 6

Conclusioni

Alla luce dei risultati ottenuti gli obiettivi prefissati dal progetto sono stati pienamente raggiunti. L'architettura implementata è stata in grado di soddisfare tutti i requisiti posti in fase di analisi. Il sistema infatti riesce a rilevare e mitigare l'attacco in breve tempo limitando di conseguenza i danni arrecati alla vittima ed al controller.

Avendo utilizzato una rete virtuale come scopi futuri è auspicabile generare il dataset con dati reali e non simulati, al fine di avere una classificazione più precisa e ad hoc per ogni tipologia di rete.

L'utilizzo delle Support Vector Machines per il rilevamento di questo tipo di attacco è ancora in fase di sperimentazione e ciò è evidenziato dalla scarsità di progetti presenti in rete a riguardo. Perciò, sotto questo aspetto, spero di aver lasciato un contributo positivo che possa essere d'aiuto per lavori futuri.

Ringraziamenti

Vorrei iniziare ringraziando il Prof. Franco Callegati, il relatore della tesi, che mi ha permesso di presentare questi argomenti che reputo estremamente interessanti e grazie al quale sono riuscito ad accrescere le mie conoscenze nell'ambito della sicurezza informatica.

Dei ringraziamenti speciali vanno alla mia famiglia, soprattutto ai miei genitori, che mi hanno permesso con i loro sacrifici di continuare gli studi, ai miei amici senza il quale non avrei trovato la forza di andare avanti nel mio percorso e alla mia ragazza che mi ha sempre sopportato e supportato, in particolar modo in questo periodo.

Un ringraziamento anche a tutti i miei compagni di corso che, anche se non ho sentito particolarmente in quest'ultimo anno di pandemia, mi hanno sempre motivato a dare il massimo di me stesso.

Bibliografia

[1] Open Networking Foundation (ONF). *OpenFlow Switch Specification*. URL: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf.

[2] Alaitz Mendiola Jasone Astorga Eduardo Jacob Marivi Higuero. *A Survey on the Contributions of Software-Defined Networking to Traffic Engineering*. URL: https://www.researchgate.net/publication/311338103_A_Survey_on_the_Contributions_of_Software-Defined_Networking_to_Traffic_Engineering.

[3] Abhinav Bhandari Maninder Pal Singh. *New-flow based DDoS attacks in SDN: Taxonomy, rationales, and research challenges*. URL: https://www.sciencedirect.com/science/article/abs/pii/S0140366419313830.

[4] Wanchun Dou Xiaotong Wu MengLiu e Shui Yu. *DDoS attacks on data plane of software-defined network: are they possible?* URL: https://onlinelibrary.wiley.com/doi/epdf/10.1002/sec.1709.

[5] Minho Park Nhu-Ngoc Dao Junho Park e Sungrae Cho. *A Feasible Method to combat against DDoS Attack in SDN Network*. URL: https://ieeexplore.ieee.org/document/7057902.

[6] Seungwon Shin Vinod Yegneswaran Phillip Porras Guofei Gu. *AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks*. URL: http://www.cs.columbia.edu/~lierranli/coms6998-10SDNFall2014/papers/Avant-CCS2013.pdf.

[7] Rohith Gandhi. *Support Vector Machine — Introduction to Machine Learning Algorithms*. URL: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

[8] Eberly College of Science. *10.3 - When Data is NOT Linearly Separable*. URL: https://online.stat.psu.edu/stat508/lesson/10/10.3.

[9] Wikipedia. *Polynomial kernel*. URL: https://en.wikipedia.org/wiki/Polynomial_kernel.

[10] Wikipedia. *Entropia (teoria dell'informazione)*. URL: https://it.wikipedia.org/wiki/Entropia_(teoria_dell%5C%27informazione).

[11] Jacir L. Bordim Ranyelson N. Carvalho e Eduardo A. P. Alchieri. *Entropy-Based DoS Attack Identification in SDN*. URL: https://ieeexplore.ieee.org/abstract/document/8778219.

[12] scikit-learn. *1.4. Support Vector Machines*. URL: https://scikit-learn.org/stable/modules/svm.html#svm-classification.

[13] Jin Ye Xiangyang Cheng Jian Zhu Luting Feng e Ling Song. *A DDoS Attack Detection Method Based on SVM in Software Defined Network*. URL: https://downloads.hindawi.com/journals/scn/2018/9804061.pdf.