Università Politecnica delle Marche

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

---

Tesi di Laurea Magistrale

Rilevamento della presenza di malware analizzando il traffico di rete in tempo reale con modelli di Deep Learning

Malware detection by analyzing real-time network traffic with Deep Learning models

Relatore:

Prof. Claudia Diamantini

Tesi di laurea magistrale di:

Daniele Marzetti

Correlatore:

Prof. Luca Spalazzi

Anno Accademico 2020-2021

Indice

1 Introduzione 11

2 Background 13
2.1 Stato dell'arte su Intrusion e malware detection 13
2.2 Architettura Transformer 14

3 Materiali e metodi 17
3.1 Dataset 17
3.2 CICFlowMeter 22
3.3 Criticità 27
3.3.1 Chiusura dovuta a FlowTimeout 28
3.3.2 Chiusura dovuta a flag FIN 29
3.3.3 Flusso che inizia con RST o FIN 30
3.3.4 Pacchetti invertiti 30
3.4 Pipeline 31

4 Data Preprocessing 33
4.1 Generazione del Dataset 33
4.1.1 Creazione del grouped packets dataset 33
4.1.2 Risoluzione dei bug individuati 35
4.2 Etichettatura del dataset 39
4.2.1 Flussi con label dubbia 40
4.2.2 Script labellatore 42
4.2.3 Flows Dataset etichettato 45
4.3 Analisi descrittiva 45
4.4 Data Selection 52
4.5 Feature Engineering 54

5 Esperimenti 61
5.1 Suddivisione del dataset 61
5.2 Esperimenti con la rete LSTM 62
5.3 Esperimenti con Transfomer 65
5.3.1 Positional Encoding 66
5.3.2 Transformer Model 67

5.3.3 Earliness 70

6 Risultati 73
6.1 Risultati LSTM 73
6.1.1 Classificazione binaria 73
6.1.2 Classificazione ensemble 75
6.2 Risultati Transformer 77

7 Conclusioni e sviluppi futuri 79

A Appendice 81
A.1 Distribuzioni delle lunghezze dei flussi maligni 81
A.2 Direct Follower Graph 83
A.3 Petri Net 87

B Appendice 91
B.1 Risultati ottenuti con l'architettura LSTM 91
B.2 Risultati ottenuti con l'architettura Transformer 95

Riferimenti bibliografici 99

Elenco delle figure

2.1 Architettura Transformer [1] 15
2.2 Meccanismo di attenzione [1] 16

3.1 Infrastruttura di rete [2] 19
3.2 BasicPacketInfo 23
3.3 BasicFlow 24
3.4 addpacket pt.1 26
3.5 addpacket pt.2 26
3.6 addpacket pt.3 27
3.7 addpacket pt.4 28
3.8 dumpLabeledCurrentFlow 29
3.9 Pacchetti invertiti 30
3.10 Pacchetti duplicati 31
3.11 Pipeline 31

4.1 toArray 34
4.2 flowtoCSV 35
4.3 addPacket mod pt.1 36
4.4 addPacket mod pt.2 37
4.5 addPacket mod pt.3 37
4.6 addPacket mod pt.4 38
4.7 addPacket mod pt.5 38
4.8 addPacket mod pt.6 39
4.9 Labellatore pt.1 43
4.10 Labellatore pt.2 44
4.11 Labellatore pt.3 44
4.12 Labellatore pt.4 44
4.13 Protocolli di rete 46
4.14 Distribuzione lunghezza flussi pt.1 47
4.15 Distribuzione lunghezza flussi pt.2 49
4.16 Direct Follower Graph Botnet 50
4.17 Direct Follower Graph XSS 50
4.18 Direct Follower Graph DoS Hulk 51

4.19 Petri Net Botnet 52
4.20 Petri Net XSS 52
4.21 Petri Net DoS Hulk 52
4.22 Data selection 54
4.23 Data Preprocess 56
4.24 Distribuzione feature 57
4.25 Feature Selection 58
4.26 Distribuzione feature 59

6.1 Risultati LSTM con dropout 74
6.2 Loss al variare del dropout 74
6.3 Risultati LSTM senza dropout 75
6.4 Risultati LSTM ensemble 76
6.5 Risultati per classe 77

A.1 Distribuzione lunghezza flussi pt.3 81
A.2 Distribuzione lunghezza flussi pt.4 82
A.3 Direct Follower Graph DoS Goldeneye 83
A.4 Direct Follower Graph DoS Slowhttptest 84
A.5 Direct Follower Graph DoS Slowloris 84
A.10 Direct Follower Graph Port Scan 84
A.6 Direct Follower Graph DDoS 85
A.7 Direct Follower Graph Brute Force 85
A.8 Direct Follower Graph FTP-Patator 86
A.9 Direct Follower Graph SSH-Patator 86
A.11 Petri Net DoS Goldeneye 87
A.12 Petri Net DoS Slowhttptest 87
A.13 Petri Net DoS Slowloris 88
A.14 Petri Net DDoS 88
A.15 Petri Net Brute Force 88
A.16 Petri Net FTP Patator 88
A.17 Petri Net SSH Patator 89
A.18 Petri Net Port Scan 89

B.1 Variazione delle metriche con WS fino a 10 92
B.2 Precision e recall per classe al variare degli iperparametri 93
B.3 Precision per classe 94
B.4 Recall per classe 95

Elenco delle tabelle

3.1 Caratteristiche dei dataset 18
3.2 Classi per giorno 18
3.3 Distribuzione delle classi nel dataset 21

4.1 Campi del grouped packets dataset 35
4.2 Confronto periodo degli attacchi tra sito web e dataset 40
4.3 Flussi anomali DoS slowloris 41
4.4 Flussi anomali Port Scan 41
4.5 Flussi anomali Botnet 41
4.6 Flussi anomali DDoS 42
4.7 Distribuzione delle classi nel flows dataset 45
4.8 Statistiche lunghezza flussi 48
4.9 Lunghezza massima dei flussi per classe 53
4.10 Distribuzione delle classi nel flows dataset dopo data selection 55
4.11 feature estratte dal grouped packets dataset 56
4.12 Statistiche feature 56
4.13 Numero di pacchetti per label 60

5.1 Configurazioni LSTM 65
5.2 Configurazione Transformer 71

6.1 Risultati classificazione binaria Transformer 77
6.2 Risultati classificazione binaria Transformer 78

B.1 Precision multi-classe Transformer 96
B.2 Recall multi-classe Transformer 97

Elenco dei listati

4.1 Undersampling benigni 54
5.1 Preprocess 61
5.2 Stratified K-Fold Split 62
5.3 Create windows for LSTM 63
5.4 Build LSTM Network 64
5.5 Create windows for Transformer 66
5.6 Positional Encoding 66
5.7 Multi-Head Self Attention 67
5.8 Transformer Encoder 68
5.9 Build Transformer 69
5.10 Compute Earliness 70

1

Introduzione

Gli attacchi informatici sono sempre più frequenti al giorno d'oggi, inoltre prima di accorgersi di un attacco possono passare anche mesi. Il progetto di questa tesi nasce dall'idea di creare una tecnica basata su algoritmi di deep learning per i SIEM (Security Information and Event Management), permettendo quindi di riconoscere eventuali attacchi in corso a partire dai log presenti in un'infrastruttura di rete. Il primo passo è stato la ricerca di un dataset contenente sia comportamenti normali che comportamenti anomali, con particolare attenzione al fatto che il dataset fosse etichettato e recente. Non trovando datset di soli log si è optato per dataset di IDS (Intrusion Detection System). L'obiettivo che ci si è posto è quello di classificare i flussi di rete contenuti in questa tipologia in real-time, tramite algoritmi di deep learning come reti LSTM (Long Short-Term Memory) e Transformer. In particolare sarà eseguita prima una classificazione binaria e successivamente una classificazione multi-classe sui flussi classificati come maligni. Per eseguire la classificazione in real-time bisogna classificare sequenze di pacchetti, per fare ciò è stato necessario raggruppare tutti i pacchetti per ogni relativo flusso. I lavori precedenti in questo campo sono molto pochi, e si vuole quindi scoprire se è possibile classificare correttamente attacchi tramite il flusso di rete utilizzando solamente informazioni contenute nei pacchetti di rete. Nel capitolo 2 sono descritti i lavori allo stato dell'arte nell'IDS e l'architettura Transformer. Nel capitolo 3 è presentato il dataset e tutti gli strumenti utilizzati per ricavare i pacchetti dal dataset, le problematiche e la pipeline. Nel capitolo 4 è descritto l'intero processo di data preprocess, data selection e feature engineering. Nel capitolo 5 sono descritti gli esperimenti realizzati, mentre nel capitolo 6 sono mostrati i risultati ottenuti con le reti. Infine nel capitolo 7 sono presenti le conclusioni ed i futuri sviluppi.

2

Background

In questo capitolo è riportato lo stato dell'arte dell'intrusion detection, inoltre è descritte l'architettura Transformer, illustrando le differenze con le reti ricorrenti.

2.1 Stato dell'arte su Intrusion e malware detection

Nel corso degli anni sono stati sviluppati vari approcci basati su algoritmi di machine learning per i sistemi di Intrusion Detection (IDS). Un survey di interesse è di Ferrag et al. [3] dove vengono valutate le performance di sette modelli di deep learning in classificazione binaria e multi-classe sui dataset CICIDS2018 [2] e Bot-IoT [4]. Altri studi comparativi sul tema sono di Ghurab et al. [5] e di Leevy et al. [6] dove il primo descrive le caratteristiche dei vari dataset di IDS riepilogandone per ognuno i risultati ottenuti da molteplici lavori, mentre il secondo illustra svariati modelli di classificazione allo stato dell'arte relativi al dataset CICIDS2018. Si sono selezionate le tecniche di maggior interesse che hanno prodotto i migliori risultati, ad esempio Zhu et al. [7] hanno sviluppato una rete LSTM con meccanismo di attenzione multi-flusso superando i risultati ottenuti con classificatori come Naive Bayes, AdaBoost, SVM, MLP e LSTM sul dataset CICIDS2017 [2]. Anche Lin et al. [8] hanno implementato una rete LSTM con meccanismo di attenzione utilizzando l'algoritmo SMOTE [9] per bilanciare il dataset CICIDS2018 effettuando oversampling, ottenendo risultati migliori rispetto a reti MLP (Multilayer Perceptron) e LSTM. Zhoua et al. [10] hanno addestrato i classificatori C4.5, RF e Forest PA sviluppando un classificatore ensemble che è stato testato sui dataset CICIDS2017, AWID [11] e NSL-KDD [12] ottenendo degli ottimi risultati. Kim et al. [13] hanno sviluppato una rete CNN (Convolutional Neural Network) multi-classe sperimentata sui dataset KDDCUP99 e CICIDS2018 superando le performance ottenute con una rete ricorrente.

Sono stati sviluppati anche dei modelli per il funzionamento in real-time come quello creato da Zhang et al. [14] che sfrutta il calcolo distribuito con Spark per classificare il traffico di rete dei dataset KDDCUP99 e CICIDS2017 tramite un random forest. Un altro modello real-time è SwiftIDS di Jin et al. [15] che si basa sul LightGBM (light gradient boosting machine) tramite il calcolo parallelo ed è stato utilizzato sui dataset KDDCUP99, NSL-KDD e CICIDS2017 ottenendo degli

ottimi risultati. Anche AI-IDS di Kim et al. [16] è stato pensato per l'uso in real-time, tale modello ha la particolarità di utilizzare anche il payload del traffico di rete. In particolare sono state confrontate sui dataset CISC2010 e CICIDS2017 una rete CNN e una rete LSTM provando varie configurazioni, ma i risultati ottenuti sono inferiori rispetto ad altri lavori.

Sfortunatamente in molti articoli mancano i particolari su come è stata svolta la fase di pulizia, selezione e bilanciamento dei dati, quindi i risultati degli studi non sono facilmente confrontabili tra loro. Per quanto riguarda il rilevamento in real-time i lavori presenti sono pochi e si basano quasi esclusivamente su dataset obsoleti.

2.2 Architettura Transformer

Il Transformer [1] è una architettura allo stato dell'arte per il NLP (Natural Language Processing). A differenza delle RNN (Recurrent Neural Network) dove il numero di operazioni per associare elementi distanti nella stessa sequenza dipende dalla distanza, compromettendo l'apprendimento di dipendenze tra posizioni distanti, il Transformer si basa sul meccanismo di attenzione. Questo permette una maggiore parallelizzazione dei calcoli che si traduce in una complessità computazionale minore e quindi in un minor tempo di addestramento. Inoltre, il meccanismo di attenzione permette di ricavare associazioni anche tra elementi molto distanti, teoricamente anche con una distanza infinita, senza incorrere in problemi di esplosione o azzeramento del gradiente. In figura 2.1 è riportata l'architettura completa di un modello Transformer, che è composto principalmente dai blocchi:

- Encoder: formato da 6 layer identici composti a loro volta da due sub-layer: multi-head self-attention e feed forward entrambi con residual connection e un layer di normalizzazione in output.
- Decoder: formato da 6 layer identici composti a loro volta dai due sub-layer che compongono un encoder, dove la multi-head self-attention riceve come input l'output dell'encoder. Inoltre è presente un ulteriore sub-layer chiamato masked multi-head self-attention che è un multi-head self-attention con una maschera per permettere all'output di dipendere solo dagli output precedenti.

L'input prima di essere passato all'encoder viene trasformato tramite un layer di word embedding che mappa le parole dette token in vettori di dimensione $\mathrm{d_{model}}$ chiamati vettori di embedding.

Dato che non sono presenti ricorrenze o convoluzioni nel modello, ai vettori di embedding viene sommato l'output dei positional encoding. Il positional encoding viene utilizzato per iniettare le informazioni sulle posizioni relative o assolute nelle sequenze di vettori di embedding. Possono essere utilizzate varie funzioni come encoding, la funzione utilizzata normalmente è riportata nella formula 2.1.

$$\begin{aligned} PE(pos, 2i) &= sin\Big(\frac{pos}{10000^{2i/dmodel}}\Big) \\ PE(pos, 2i+1) &= cos\Big(\frac{pos}{10000^{2i/dmodel}}\Big) \end{aligned} \tag{2.1}$$

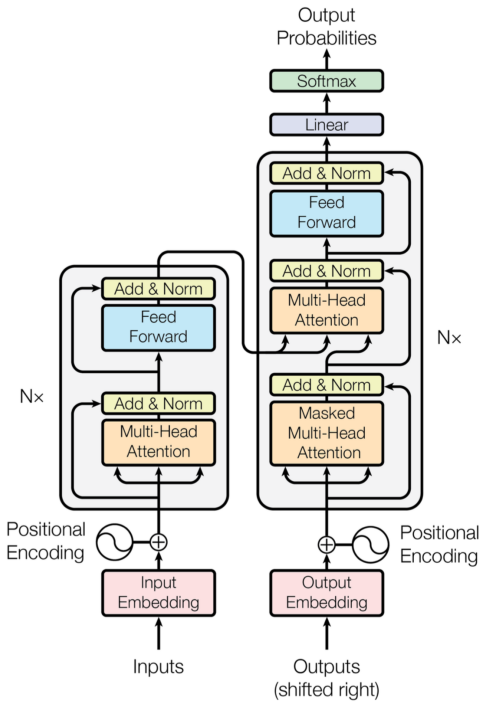


Figure 1: The Transformer - model architecture.

Figura 2.1: Architettura Transformer [1]

Nella formula 2.1 *pos* rappresenta la posizione di un vettore di embedding nella sequenza e i è la dimensione i-esima del vettore di embedding. Quindi se si vuole ottenere ad esempio il positional encoding della seconda dimensione di un vettore di embedding in posizione 3 rispetto alla sequenza a cui appartiene: *pos* è pari a 2 e i è pari a 3.

La funzione seno è usata per il calcolo dei positional encoding per le dimensioni pari, mentre il coseno è utilizzato nel calcolo delle dimensioni dispari.

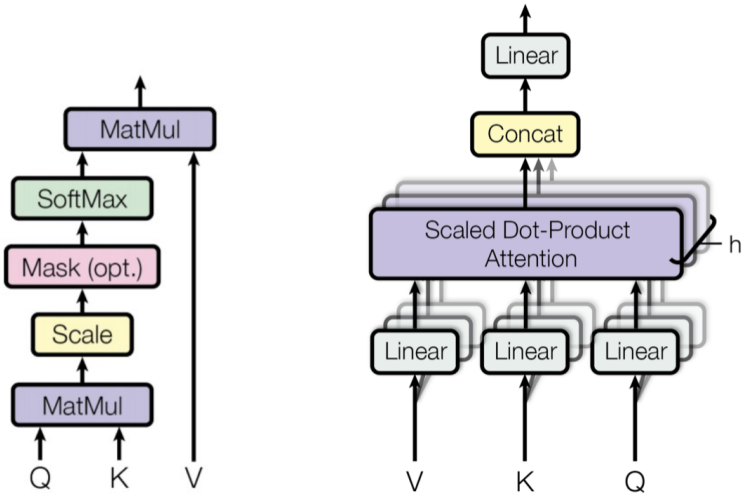
Il meccanismo di attenzione (figura 2.2a) che è alla base del multi-head self-attention si basa sulla formula 2.2. A differenza delle RNN o delle LSTM, il meccanismo di attenzione teoricamente permette di ricavare associazioni tra elementi anche con sequenze di lunghezza infinita.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.2}$$

Nella formula 2.2 Q è la matrice delle query, K è la matrice delle key e V è la matrice dei value. Nella *self* attention queste matrici sono le stesse, ovvero l'input dell'encoder.

Il meccanismo di attention permette di confrontare ogni singolo elemento nella sequenza con tutti gli altri, focalizzandosi solo sugli elementi con maggiore importanza. Quindi pesa l'input sulla base dell'input stesso.

Aggiungendo h proiezioni lineari diversi alle matrici Q, K, V si ottiene la multi-head self-attention. In particolare vengono utilizzati h self-attention, chiamate teste o heads. Ogni testa è differente dalle altre perché le matrici delle proiezioni (QW_i, QK_i, QV_i) sono imparate durante la fase di training. I meccanismi di attenzione posono quindi essere eseguiti in parallelo, e presentano il vantaggio che l'input, tramite le proiezioni lineari, viene trasformato e ogni testa si specializza in un determinato aspetto dell'input. Gli output della multi-head self-attention vengono poi concatenati e passati al feed forward sub-layer.



(a) Attention (b) Multi-head Attention

Figura 2.2: Meccanismo di attenzione [1]

Il layer decoder viene utilizzato soltanto se si è in una problematica del tipo seq2seq. Ad esempio nei problemi di NLP o di traduzione automatica, dove l'output deve essere anch'esso una sequenza di token e può avere dimensione diversa dalla sequenza di input. Infine è presente un layer softmax di output, questo layer può essere utilizzato anche direttamente dopo l'encoder se non si è in un problema seq2seq. Ovviamente il numero di encoder e decoder non deve essere fisso a 6. Esistono vari modelli di transformer che sono stati costruiti nel corso degli anni che rispondo a esigenze diverse, come computer vision, traduttori automatici, ecc...

3

Materiali e metodi

In questo capitolo sono illustrate le caratteristiche del dataset cercato, successivamente viene descritto nei dettagli come è formato il dataset scelto e come funziona il tool da cui è stato generato. Inoltre sono spiegati i bug trovati nel tool e le implicazioni che vengono a crearsi. Infine è spiegata l'intera pipeline del progetto fornendo la nomenclatura dei vari datset e spiegando gli step del progetto.

3.1 Dataset

Come anticipato nell'introduzione, non trovando un buon dataset contenente log di un'infrastruttura di rete si sono cercati dataset di IDS (Intrusion detection system)/IPS (Intrusion prevention system) contenenti sia comportamenti normali che comportamenti anomali. Si è posta particolare attenzione al fatto il dataset sia etichettato e recente. Inoltre, per usare algoritmi di deep learning come LSTM (Long Short-Term Memory) e Transformer per sintetizzare uno schema di processo di attacco è necessario che gli elementi nel dataset possano essere raggruppati in base al processo di appartenenza. Ovvero che il dataset sia un log di eventi, e che tramite uno o più campi (Case ID) si possano raggruppare tutti gli eventi riferiti ad una specifica traccia.

Nella ricerca di un dataset con le caratteristiche descritte, è stato d'aiuto il survey di Ring et al. [17] che fornisce una panoramica esaustiva riguardo i dataset di network intrusion detection.

La scelta è ricaduta sui dataset riportati in tabella 3.1, dato che la maggior parte dei dataset trovati non corrispondevano a quanto cercato. In particolare alcuni non sono completamente pubblici, molti non sono etichettati e la maggior parte sono obsoleti e non contengono nemmeno flussi HTTPS. Invece è necessario che il dataset sia recente perché gli attacchi cambiano molto in fretta nel tempo, a causa della scoperta di nuovi exploit e delle risoluzioni delle vulnerabilità.

Questi dataset contengono feature statistiche relative ai singoli flussi del traffico di rete, inoltre hanno la particolarità di disporre delle catture dei pacchetti di rete (file pcap) e del tool utilizzato per ottenere le statistiche.

Nome	Anno	Formato	Flussi	Durata	Traffico benigno
CICIDS2017	2017	packet, flow	2.8M	5 giorni	simulato
CTU-13	2011	packet, flow	23.7M	142 ore	reale
CICIDS2018	2018	packet, flow, log	16.2M	10 giorni	simulato

Tabella 3.1: Caratteristiche dei dataset

Quindi a partire dalle catture dei pacchetti di rete è possibile associare i pacchetti ai relativi flussi. Utilizzando il concetto di flusso come traccia e il pacchetto come evento.

Il dataset CTU-13 [18] contiene solo traffico benigno, background e relativo a vari tipi di attacchi Botnet. Inoltre questo dataset non contiene i payload dei pacchetti relativi a flussi benevoli, ma solo gli header per questioni di privacy perché i flussi benigni sono reali. Siccome questo dataset non è molto recente e soffre dei limiti appena descritti è stato escluso dalla scelta.

Per quanto riguarda i dataset del CIC (Canadian Institute for Cybersecurity) sono entrambi recenti, quindi contengono dei flussi di rete similari a quelli odierni, inoltre hanno una maggiore varietà di tipologie d'attacco rispetto a CTU-13.

La differenza principale tra loro è il numero di flussi, mentre le tipologie di attacco sono le medesime. Infatti il dataset CICIDS2018 [2] è stato creato utilizzando un'infrastruttura di rete molto maggiore rispetto a quella utilizzata per il dataset CICIDS2017 [2]. Un'altra differenza è che il dataset del 2018 contiene anche i log dei sistemi operativi di ogni macchina presente nell'infrastruttura. Il problema di questo dataset è che ha dimensioni compresse maggiori di 220 GB, appunto perché sono state utilizzate molte più macchine e per il doppio dei giorni rispetto alla versione del 2017. Inoltre nel dataset del 2018 è presente un file pcap per ogni computer per ogni giorno, mentre nel dataset del 2017 è contenuto un solo file pcap per ogni giorno. Per questi motivi è stato scelto il dataset CICIDS2017 [2].

Tale dataset è di una fonte autorevole, ovvero il Canadian Institute for Cybersecurity (CIC) e l'University of New Brunswick (UNB). Il dataset contiene le catture dei pacchetti di rete di 5 giorni, dalle 09:00 di lunedì 03/07/2017 alle ore 17:00 di venerdì 07/07/2017. In tabella 3.2 sono riportate le classi presenti nel dataset per ogni giorno di acquisizione. Come si può notare solo i flussi benigni sono stati catturati in vari giorni, mentre tutti i flussi maligni sono stati catturati in uno specifico arco temporale di un determinato giorno.

Day	Classi
Monday	Benign
Tuesday	Benign, FTP-Patator, SSH-Patator
Wednesday	Benign, Heartbleed, DoS slowloris, DoS Slowhttptest DoS Hulk, DoS GoldenEye
Thursday	Benign, Infiltration, Web Attack - Brute Force Web Attack - XSS, Web Attack - Sql Injection
Friday	Benign, Botnet, Port Scan, DDoS

Tabella 3.2: Classi per giorno

Il dataset contiene due diversi formati di file:

- pcap: contengono una cattura completa dei pacchetti di rete transitati nell'infrastruttura di rete in un determinato giorno
- csv: contengono le informazioni, le feature statistiche e la classe relativamente ad ogni flusso

I file csv sono stati generati a partire dal tool CICFlowMeter [19]. Il dataset contiene 8 file csv perché alcuni giorni sono stati suddivisi in più file. Riguardo i file csv è presente sia una versione completa contenente anche indirizzi IP, numeri di porta, numero del protocollo di trasporto e ID per ogni flusso, sia una versione sprovvista di tali informazioni già pronta per essere usata come input per un classificatore.

L'infrastruttura di rete, riportata in figura 3.1, è formata da una rete vittima e da una rete attaccante. La rete vittima è composta da dieci computer con sistemi operativi: Windows, Macintosh, Linux, inoltre sono presenti anche: un firewall con NAT (Network Address Translation), uno switch, un router e tre server. Mentre la rete attaccante è composta da uno switch, un router e quattro computer attaccanti con i sistemi operativi: Kali Linux e Windows. I pacchetti di rete inviati o ricevuti sono stati catturati tramite una porta dello switch nella rete vittima che è stata configurata come mirror port.

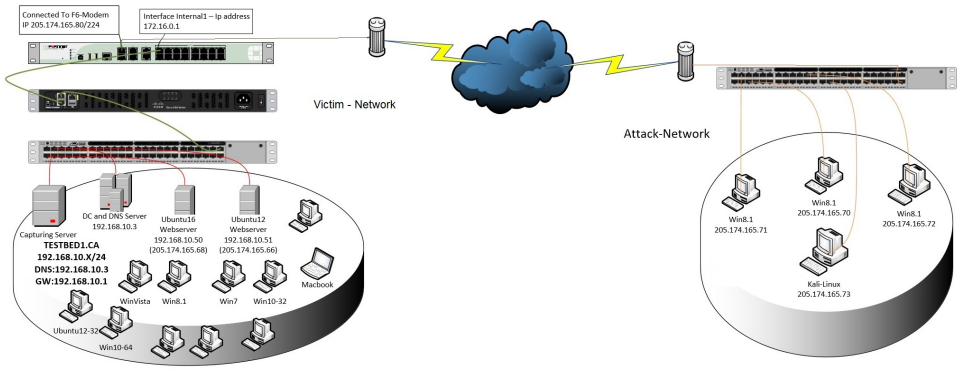


Figura 3.1: Infrastruttura di rete [2]

Il flussi benevoli sono stati ottenuti tramite una simulazione usando degli agenti java che si sono scambiati traffico sulla rete basato su dei profili di connessione ottenuti tramite algoritmi di machine learning. In particolare il profilo benigno utilizzato (B-profile) è basato sul comportamento di 25 utenti e genera flussi HTTP, HTTPS, FTP, SSH e email. Gli algoritmi utilizzati per ottenere il B-profile sono K-Means, Random Forest, SVM e J48, utilizzando feature come la distribuzione della dimensione dei pacchetti per protocollo, il numero dei pacchetti per flusso, la distribuzione del tempo di richiesta e specifici pattern nei payload.

I dati malevoli invece sono stati ottenuti eseguendo dei veri attacchi sulla rete vittima da parte dalla rete attaccante . Di seguito sono descritte le tipologie degli attacchi che sono stati eseguiti:

- Botnet: l'attacco è stato eseguito utilizzando il RAT (Remote Access Tool) Ares [20] scritto in Python, che permette di eseguire comandi tramite una shell remota ed effettuare: key logging, screenshot, upload e download di file sulle macchine infettate all'insaputa degli utenti. Ares è composto da due programmi distinti:
  - server C&C (Comando e Controllo): funge da interfaccia web con la quale controllare gli agenti
  - agente: viene eseguito dal computer vittima ed assicura la connessione con il server C&C

  L'interfaccia web, ovvero il server C&C, può esser eseguito su qualsiasi server Python, mentre gli agenti, dopo essere stati compilati, vengono installati sui computer delle vittime come eseguibili.
- Patator [21]: è un programma multi-thread scritto in Python che supporta oltre 30 protocolli, tra cui FTP, SSH, SMTP, HTTP, DNS. Per il corretto funzionamento del tool si deve creare un file contenente gli username che si vogliono attaccare e un file contenente le varie password da provare. Nel dataset gli attacchi effettuati sono stati solo sui protocolli FTP e SSH.
- Web attack: gli attacchi web (Brute force, XSS, SQLi) sono stati svolti attaccando una Damn Vulnerable Web App (DVWA) [22]. Una DVWA è un'applicazione web in PHP e MySQL piena di vulnerabilità di sicurezza, che viene usata dai professionisti di sicurezza per studiare gli attacchi che vi vengono condotti oppure può essere usata come allenamento o per scopi didattici. Gli attacchi XSS e Brute Force sono stati eseguiti in modo automatico tramite il framework Selenium, ma non sono riportati ulteriori dettagli.
- DoS Goldeneye [23]: è un tool scritto in Python che permette di eseguire un attacco DoS (Denial of service). Il traffico HTTP generato è perfettamente legittimo ed il vettore d'attacco è: HTTP Keep Alive + NoCache. Quindi per rendere indisponibile un server il tool invia centinaia di richieste di connessione HTTP con le opzioni Keep Alive e NoCache ad intervalli regolari di tempo. Una caratteristica di questo attacco è che non richiede troppa banda per aver successo.
- DoS Slowloris e DoS Slowhttptest: il protocollo HTTP è progettato in modo tale che le richieste devono esser completamente ricevute prima di poter essere elaborate. Se una richiesta HTTP è incompleta, ad esempio per una bassa larghezza di banda, il server web manterrà le risorse occupate in attesa del completamento della richiesta. Se il server mantiene troppe risorse in attesa si può generare un DoS. Slowhttptest [24] è un tool che fa proprio questo: invia delle richieste HTTP incomplete lasciando aperta la connessione con il server con una banda minima. Slowloris è uno degli attacchi che si possono selezionare dal tool.
- DoS Hulk: la maggior parte dei tool DoS utilizzano dei pattern che si ripetono rendendo così l'attacco facilmente prevedibile da IDS. Hulk (Http Unbearable Load King) [25] è un tool in Python che ad ogni richiesta genera un modello unico, bypassando i cache engine ed impattando direttamente sulle risorse del server. Come tecniche di elusione utilizza:
  - Offuscamento del client: l'user agent viene scelto in maniera casuale da una lista di user agent.
  - HTTP Keep Alive con una finestra temporale variabile.
  - NoCache, ma se un server utilizza un servizio di caching non avrà effetto.

  - Trasformazione univoca dell'URL: l'URL viene trasformato aggiungendo dei parametri e dei valori al fine di evitare l'utilizzo di una pagina memorizzata nella cache, eludendo così anche i servizi di caching.
- DDoS: è stato utilizzato un tool GUI molto user-friendly chiamato LOIC (Low Orbit Ion Cannon) [26] che permette di effettuare attacchi DoS e DDoS. Loic apre molteplici connessioni sfruttando i flood: TCP, UDP e HTTP GET finché il server si sovraccarica e smette di rispondere alle richieste.
- Port Scan: per eseguire l'attacco è stato utilizzato il tool NMap, in particolare le opzioni: sS, sT, sF, sX, sN, sP, sV, sU, sO, sA, sW, sR,sL e B.
- Heartbleed: è un bug di sicurezza nella libreria crittografica OpenSSL scoperto nelle versioni minori della 1.0.1g. Questa vulnerabilità permette un over-read del buffer durante la richiesta heartbeat, consentendo così di ottenere la secret key. In particolare, viene inviata una richiesta heartbeat malformata con un payload piccolo ma un campo molto grande per il server vittima. Il tool utilizzato per sfruttare il bug Heartbleed si chiama Heartleech [27].
- Infiltration: per infiltrarsi all'interno della rete vittima è stato utilizzato il tool Metasploit che permette di eseguire scansioni di vulnerabilità, consentendo anche di sfruttarle per effettuare exploit e penetration test. L'exploit, dopo esser stato scaricato dalla vittima da dropbox o tramite una chiavetta usb, installerà una backdoor e inizierà una scansione delle porte internamente alla rete vittima.

Nella tabella 3.3 è riportata la distribuzione delle classi del dataset. Come si può notare il dataset è fortemente sbilanciato verso i flussi benigni. Questo aspetto sarà approfondito nel paragrafo 3.3.

Label	**Conteggio flussi**
BENIGN	2.273.097
DoS Hulk	231.073
PortScan	158.930
DDoS	128.027
DoS GoldenEye	10.293
FTP-Patator	7.938
SSH-Patator	5.897
DoS slowloris	5.796
DoS Slowhttptest	5.499
Bot	1.966
Web Attack – Brute Force	1.507
Web Attack – XSS	652
Infiltration	36
Web Attack – Sql Injection	21
Heartbleed	11
Totale	**2.830.743**

Tabella 3.3: Distribuzione delle classi nel dataset

Un ultimo aspetto importante di questo dataset, ma anche del CICIDS2018, è che rispettano gli 11 criteri stilati da Sharafaldin et al. [28] per la generazione di un corretto dataset di IDS. I criteri sono i seguenti:

1. Completa configurazione della rete: l'infrastruttura di rete deve essere realistica e deve includere: switch, router, firewall e vari sistemi operativi, perché gli attacchi possono avere comportamenti differenti in base al sistema.
2. Traffico completo.
3. Dataset etichettato
4. Iterazione completa: avere delle connessioni esterne ma anche interne alla LAN.
5. Cattura completa: il traffico di rete deve essere catturato nella sua interezza, per ottenere una valutazione accurata dagli IDS.
6. Disponibilità di protocolli: è importante che vengano utilizzati i vari protocolli di comunicazione e che ci sia traffico di tipo interattivo e traffico congestionato.
7. Diversità degli attacchi.
8. Anonimato: gli indirizzi IP e i payload sono utili per meccanismi di rilevamento come il deep packet inspection (DPI), quindi non dovrebbero essere rimossi.
9. Eterogeneità: gli IDS possono avere varie fonti, un dataset eterogeneo dovrebbe contenere ad esempio dump di memoria o log.
10. Insieme delle feature definito.
11. Metadati: il dataset dovrebbe sempre essere documentato con tutti i dettagli di realizzazione come l'infrastruttura di rete, i sistemi operativi delle macchine, la tipologia di attacchi, ecc...

3.2 CICFlowMeter

Il tool CICFlowMeter [19] o ISCXFlowMeter è un generatore di flussi di traffico di rete bidirezionali. Come visto nel precedente paragrafo è stato utilizzato per la generazione dei flussi, ovvero i file csv, nei dataset CICIDS2017 e CICIDS2018. CICFlowMeter è disponibile in due versioni differenti su due repository, la trattazione si baserà sulla versione più aggiornata del tool, anche se per la creazione del dataset CICIDS2017 è stata utilizzata una versione ancora più datata rispetto alla versione più datata disponibile. Ogni flusso viene caratterizzato con oltre 80 feature statistiche relative al traffico di rete come: durata, numero di pacchetti scambiati, numero di byte scambiati, lunghezza media dei pacchetti, deviazione standard della lunghezza dei pacchetti. Ogni feature è calcolata sia rispetto al flusso forward che al flusso backward. Il tool può generare i flussi direttamente in real time, oppure a partire dalle catture dei pacchetti di rete (file pcap). CICFlowMeter può essere installato sia sui sistemi Windows che sui sistemi Linux e può essere utilizzato sia tramite CLI che tramite GUI, quindi è user-friendly. Il programma è scritto nel linguaggio di programmazione Java e si basa sulla libreria jNetpcap che è un wrapper di libpcap/winpcap e funge da interfaccia verso i file pcap. Come prerequisiti il tool ha bisogno di Java JDK, di un IDE per Java, come Eclipse, e di Maven. Per essere installato bisogna clonare il seguente repository GitHub `https://github.com/ahlashkari/CICFlowMeter` e digitare il comando:

```
mvn install:install-file -Dfile=jnetpcap.jar -DgroupId=org.jnetpcap -DartifactId=jnetpcap
-Dversion=1.4.1 -Dpackaging=jar
```

all'interno della cartella jnetpcap-1.4.r1425. Successivamente bisogna aprire Eclipse, importare il progetto come progetto Maven e creare una nuova run configuration inserendo come nuovo VM argument:

```
-Djava.library.path= <path della cartella jnetpcap-1.4.r1425>
```

Il tool è formato da varie classi Java, la seguente trattazione si concentrerà solo sulle classi principali tralasciando le classi che gestiscono la parte grafica, il multithreading o altri aspetti secondari. Inoltre, sono presenti varie classi quasi identiche tra loro perché alcune vengono eseguite tramite CLI e altre tramite GUI.

Lo strumento legge un file pcap pacchetto per pacchetto, ovvero sequenzialmente tramite la classe ReadpcapFileWorker. I pacchetti di rete sono rappresentati dalla classe BasicPacketInfo, in figura 3.2 sono riportati gli attributi della classe. In realtà la classe non rappresenta propriamente un pacchetto di rete ma un pacchetto in modo più astratto, dato che contiene informazioni dei livelli: data link, rete e trasporto.

```
7  public class BasicPacketInfo {
8
9  /*  Basic Info to generate flows from packets   */
10     private    long id;
11     private    byte[] src;
12     private    byte[] dst;
13     private    int srcPort;
14     private    int dstPort;
15     private    int protocol;
16     private    long   timeStamp;
17     private    long   payloadBytes;
18     private    String  flowId = null;
19 /* ****************************************** */
20     private    boolean flagFIN = false;
21     private    boolean flagPSH = false;
22     private    boolean flagURG = false;
23     private    boolean flagECE = false;
24     private    boolean flagSYN = false;
25     private    boolean flagACK = false;
26     private    boolean flagCWR = false;
27     private    boolean flagRST = false;
28     private    int TCPWindow=0;
29     private    long headerBytes;
30     private int payloadPacket=0;
```

Figura 3.2: BasicPacketInfo

Ogni pacchetto viene caratterizzato con indirizzo IP sorgente e destinazione, porta sorgente e destinazione, campo protocollo dell'header di livello tre, timestamp, flag trasportati, valore del campo TCPWindow, numero di byte contenuti nel payload e nell'header di livello quattro. Ovviamente alcuni attributi come TCPWindow e i flag sono caratterizzati solo se il pacchetto contiene un segmento TCP. Inoltre ogni pacchetto ha un Flow ID, ovvero una stringa che segue il pattern:

```
Source IP - Destination IP - Source Port - Destination Port - Protocol
```

Il Flow ID non è univoco, dato che a distanza di tempo possono esistere flussi che transitano tra le stesse macchine e sulle stesse porte, utilizzando lo stesso protocollo di trasporto. Sono di particolare importanza i metodi *fwdFlowId* e *bwdFlowId* presenti nella classe BasicPacketInfo, il primo genera il Forward Flow ID mentre il secondo genera il Backward Flow ID. Il Forward Flow ID è definito come il Flow ID, mentre il Backward Flow ID è l'opposto ovvero:

```
Destination IP - Source IP - Destination Port - Source Port - Protocol
```

I vari Flow ID servono al tool per identificare il flusso al quale appartiene il pacchetto e allo stesso tempo per sapere se un pacchetto è in backward o in forward rispetto al flusso. La classe PacketReader permette di effettuare il mapping tra i pacchetti contenuti nel file pcap e la classe BasicPacketInfo, infatti utilizzando la libreria jNetpcap popola gli oggetti della classe BasicPacketInfo.

Il concetto di flusso è invece rappresentato dalla classe BasicFlow. In figura 3.3 sono riportati gli attributi della classe BasicFlow.

```
10 public class BasicFlow {
11
12     private final static String separator = ",";
13     private     SummaryStatistics       fwdPktStats = null;
14     private     SummaryStatistics       bwdPktStats = null;
15     private     List<BasicPacketInfo>   forward = null;
16     private     List<BasicPacketInfo>   backward = null;
17
18     private     long forwardBytes;
19     private     long backwardBytes;
20     private     long fHeaderBytes;
21     private     long bHeaderBytes;
22
23     private     boolean isBidirectional;
24
25     private     HashMap<String, MutableInt> flagCounts;
26
27     private     int fPSH_cnt;
28     private     int bPSH_cnt;
29     private     int fURG_cnt;
30     private     int bURG_cnt;
31     private     int fFIN_cnt;
32     private     int bFIN_cnt;
33
34     private     long Act_data_pkt_forward;
35     private     long min_seg_size_forward;
36     private     int Init_Win_bytes_forward=0;
37     private     int Init_Win_bytes_backward=0;
38
39
40     private     byte[] src;
41     private     byte[] dst;
42     private     int    srcPort;
43     private     int    dstPort;
44     private     int    protocol;
45     private     long   flowStartTime;
46     private     long   startActiveTime;
47     private     long   endActiveTime;
48     private     String flowId = null;
49
50     private     SummaryStatistics flowIAT = null;
51     private     SummaryStatistics forwardIAT = null;
52     private     SummaryStatistics backwardIAT = null;
53     private     SummaryStatistics flowLengthStats = null;
54     private     SummaryStatistics flowActive = null;
55     private     SummaryStatistics flowIdle = null;
56
57     private     long   flowLastSeen;
58     private     long   forwardLastSeen;
59     private     long   backwardLastSeen;
60     private     long   activityTimeout;
```

Figura 3.3: BasicFlow

Come si può notare alcuni attributi sono gli stessi della classe BasicPacketInfo, però sono relativi al flusso e non ad un singolo pacchetto. Un flusso è caratterizzato quindi da: indirizzo IP sorgente e destinazione, porta sorgente e destinazione, numero del protocollo di trasporto, timestamp di inizio del flusso, contatori dei vari flag TCP e una serie di attributi che servono al tool per calcolare le varie feature statistiche. Ogni flusso ha anche: una lista contenente tutti gli oggetti BasicPacketInfo in forward e un'altra lista contenente tutti gli oggetti BasicPacketInfo in backward relativi al flusso. Il Flow ID è definito come per i pacchetti e viene istanziato con il valore del Flow ID del primo pacchetto del flusso.

La classe più importante è FlowGenerator e permette di generare i flussi a partire dalle catture dei pacchetti di rete. In questa classe è presente il metodo addPacket che aggiunge al flusso tutti i pacchetti relativi al flusso stesso, oppure crea un nuovo flusso se il pacchetto scansionato è il primo del flusso, cioè se il flusso si è appena aperto. Per generare i flussi FlowGenerator usa l'attributo currentFlows che è un oggetto HashMap contenente come key i Flow ID dei flussi e come value gli oggetti BasicFlow. Questo oggetto, durante l'esecuzione del tool, contiene tutti i flussi che sono stati generati e non sono ancora giunti a conclusione in un certo istante di tempo della scansione dei pacchetti. Quindi currentFlows cresce nel tempo con l'arrivo di pacchetti che instaurano nuove connessioni. All'interno di currentFlows tutti i flussi hanno Flow ID univoco. Il metodo addPacket prima di tutto controlla se il forward Flow ID o il backward Flow ID del pacchetto, che si sta attualmente scansionando, sia già presente all'interno delle chiavi di currentFlows. Se l'esito è negativo viene creato un nuovo flusso, contenente il pacchetto, che viene aggiunto ai currentFlows e il Flow ID del flusso sarà il forward Flow ID del pacchetto. Se invece il pacchetto fa parte di un flusso che si trova nei flussi correnti, il programma controlla se il pacchetto rientra tra le condizioni di chiusura di un flusso, ovvero:

1. FlowTimeout esaurito, come mostrato in figura 3.4
2. pacchetto contenente flag FIN, come mostrato nelle figure 3.5 e 3.6
3. pacchetto contenente flag RST, come mostrato in figura 3.7

Tali condizioni di chiusura sono state definite dai creatori del tool, ma come illustrato nel paragrafo 3.3 non sono delle condizioni sufficienti per ottenere dei flussi che rispecchino fedelmente i flussi delle catture dei pacchetti di rete.

L'attributo flowTimeout indica la massima durata che può avere un flusso e ha 2 minuti come valore impostato di default, ma può essere modificato a piacimento. Nel caso 1 un flusso viene chiuso se il flowTimeout è scattato, successivamente il flusso viene salvato nel file csv ed eliminato dai currentFlows. Il pacchetto che si sta attualmente scansionando sarà inserito come primo pacchetto di un nuovo flusso, che avrà lo stesso Flow ID di quello eliminato. Quindi il flusso appena eliminato ed il nuovo flusso avranno lo stesso Flow ID, che non sarà necessariamente il Forward Flow ID del pacchetto, come nel caso di creazione di un nuovo flusso, ma potrebbe anche essere il Backward Flow ID.

Per quanto riguarda il caso 2 il metodo distingue se il pacchetto con il flag FIN è in backward o in forward rispetto al flusso. Il comportamento è speculare: in entrambi i casi viene incrementato di uno il valore del contatore FwdFINFlag o BwdFINFlag del flusso e se la somma di questi attributi è pari a due allora il flusso viene salvato nel file csv e rimosso dai currentFlows. Altrimenti il pacchetto viene aggiunto al flusso e currentFlows viene aggiornato. Quindi un flusso viene chiuso solo se riceve un flag FIN in forward e un flag FIN in backward, indifferentemente dall'ordine di ricezione. Ma se per qualunque motivo arrivassero multipli flag FIN da un lato della connessione, prima che l'altro lato invii un flag FIN, il flusso non si potrebbe mai chiudere, a meno del verificarsi della condizione 1. Inoltre, il tool scarta dal flusso eventuali pacchetti contenenti ulteriori flag FIN ricevuti.

Nel caso 3, ovvero pacchetto contenente il flag di TCP RESET, il flusso viene aggiornato con l'inserimento dell'ultimo pacchetto e successivamente viene salvato nel file csv e rimosso dai currentFlows. In tutti gli altri casi, ovvero se il pacchetto

```
81  public void addPacket(BasicPacketInfo packet) {
82      if (packet == null) {
83          return;
84      }
85
86      BasicFlow flow;
87      long currentTimestamp = packet.getTimeStamp();
88      String id;
89
90      if (this.currentFlows.containsKey(packet.fwdFlowId()) || this.currentFlows.containsKey(packet.bwdFlowId())) {
91
92          if (this.currentFlows.containsKey(packet.fwdFlowId())) {
93              id = packet.fwdFlowId();
94          } else {
95              id = packet.bwdFlowId();
96          }
97
98          flow = currentFlows.get(id);
99          // Flow finished due flowtimeout:
100         // 1.- we move the flow to finished flow list
101         // 2.- we eliminate the flow from the current flow list
102         // 3.- we create a new flow with the packet-in-process
103         if ((currentTimestamp - flow.getFlowStartTime()) > flowTimeOut) {
104             if (flow.packetCount() > 1) {
105                 if (mListener != null) {
106                     mListener.onFlowGenerated(flow);
107                 } else {
108                     finishedFlows.put(getFlowCount(), flow);
109                 }
110                 // flow.endActiveIdleTime(currentTimestamp,this.flowActivityTimeOut,
111                 // this.flowTimeOut, false);
112             }
113             currentFlows.remove(id);
114             currentFlows.put(id, new BasicFlow(bidirectional, packet, flow.getSrc(), flow.getDst(),
115                     flow.getSrcPort(), flow.getDstPort(), this.flowActivityTimeOut));
```

Figura 3.4: addpacket pt.1

```
126         } else if (packet.hasFlagFIN()) {
127             //
128             // Forward Flow
129             //
130             if (Arrays.equals(flow.getSrc(), packet.getSrc())) {
131                 // How many forward FIN received?
132                 if (flow.setFwdFINFlags() == 1) {
133                     // Flow finished due FIN flag (tcp only)?:
134                     // 1.- we add the packet-in-process to the flow (it is the last packet)
135                     // 2.- we move the flow to finished flow list
136                     // 3.- we eliminate the flow from the current flow list
137                     if ((flow.getBwdFINFlags() + flow.getBwdFINFlags()) == 2) {
138                         logger.debug("FlagFIN current has {} flow", currentFlows.size());
139                         flow.addPacket(packet);
140                         if (mListener != null) {
141                             mListener.onFlowGenerated(flow);
142                         } else {
143                             finishedFlows.put(getFlowCount(), flow);
144                         }
145                         currentFlows.remove(id);
146                         // Forward Flow Finished.
147                     } else {
148                         logger.info("Forward flow closed due to FIN Flag");
149                         flow.updateActiveIdleTime(currentTimestamp, this.flowActivityTimeOut);
150                         flow.addPacket(packet);
151                         currentFlows.put(id, flow);
152                     }
153                 } else {
154                     // some error
155                     // TODO: review what to do with the packet
156                     logger.warn("Forward flow received {} FIN packets", flow.getFwdFINFlags());
157                 }
158                 //
159                 // Backward Flow
160                 //
```

Figura 3.5: addpacket pt.2

```
161                 } else {
162                     // How many backward FIN packets received?
163                     if (flow.setBwdFINFlags() == 1) {
164                         // Flow finished due FIN flag (tcp only)?:
165                         // 1.- we add the packet-in-process to the flow (it is the last packet)
166                         // 2.- we move the flow to finished flow list
167                         // 3.- we eliminate the flow from the current flow list
168                         if ((flow.getBwdFINFlags() + flow.getBwdFINFlags()) == 2) {
169                             logger.debug("FlagFIN current has {} flow", currentFlows.size());
170                             flow.addPacket(packet);
171                             if (mListener != null) {
172                                 mListener.onFlowGenerated(flow);
173                             } else {
174                                 finishedFlows.put(getFlowCount(), flow);
175                             }
176                             currentFlows.remove(id);
177                             // Backward Flow Finished.
178                         } else {
179                             logger.info("Backwards flow closed due to FIN Flag");
180                             flow.updateActiveIdleTime(currentTimestamp, this.flowActivityTimeOut);
181                             flow.addPacket(packet);
182                             currentFlows.put(id, flow);
183                         }
184                     } else {
185                         // some error
186                         // TODO: review what to do with the packet
187                         logger.warn("Backward flow received {} FIN packets", flow.getBwdFINFlags());
188                     }
189                 }
```

Figura 3.6: addpacket pt.3

non contiene flag FIN o RST e se il flowtimeout non è scattato, il pacchetto viene aggiunto al flusso e currentFlows viene aggiornato. Ma se il pacchetto dovesse arrivare da un lato della connessione da cui è stato inviato in precedenza un flag FIN, il pacchetto verrebbe scartato come riportato in figura 3.6.

Non tutti i flussi rientrano tra le regole di chiusura previste per vari motivi, il più semplice è dovuto al fatto che una volta che il tool raggiunge la fine del file pcap possono esserci ancora dei flussi attivi. Quindi nella classe FlowGenerator è presente anche il metodo dumpLabeledCurrentFlow che salva nel csv i flussi rimanenti in currentFlows, una volta raggiunta la fine del file pcap. In figura 3.8 è riportato il codice del metodo.

I flussi vengono scritti nel file csv solo se contengono almeno due pacchetti, indipendentemente dal fatto che il flusso sia terminato o meno.

3.3 Criticità

Durante lo svolgimento della tesi sono emerse varie criticità nel dataset e nel tool CICFlowMeter. Per quanto riguarda il dataset, come mostrato nella tabella 3.3, è presente un grande sbilanciamento tra le classi. Infatti, i flussi benigni sono circa 2.270.000 mentre i flussi maligni sono solo circa 557.000, ovvero meno di 1/4 rispetto ai benigni. Lo sbilanciamento è ancora più accentuato tra le classi maligne, infatti si passa dagli 11 flussi Heartbleed agli oltre 230.000 flussi DoS Hulk, con una differenza di quattro ordini di grandezza. Ma la criticità principale del dataset è il fatto che i flussi nei csv (flows dataset) non corrispondono ai flussi generati con Wireshark.

```
194             } else if (packet.hasFlagRST()) {
195                 logger.debug("FlagRST current has {} flow", currentFlows.size());
196                 flow.addPacket(packet);
197                 if (mListener != null) {
198                     mListener.onFlowGenerated(flow);
199                 } else {
200                     finishedFlows.put(getFlowCount(), flow);
201                 }
202                 currentFlows.remove(id);
203             } else {
204                 //
205                 // Forward Flow and fwdFIN = 0
206                 //
207                 if (Arrays.equals(flow.getSrc(), packet.getSrc()) && (flow.getFwdFINFlags() == 0)) {
208                     flow.updateActiveIdleTime(currentTimestamp, this.flowActivityTimeOut);
209                     flow.addPacket(packet);
210                     currentFlows.put(id, flow);
211                     //
212                     // Backward Flow and bwdFIN = 0
213                     //
214                 } else if (flow.getBwdFINFlags() == 0) {
215                     flow.updateActiveIdleTime(currentTimestamp, this.flowActivityTimeOut);
216                     flow.addPacket(packet);
217                     currentFlows.put(id, flow);
218                     //
219                     // FLOW already closed!!!
220                     //
221                 } else {
222                     logger.warn("FLOW already closed! fwdFIN {} bwdFIN {}", flow.getFwdFINFlags(),
223                             flow.getBwdFINFlags());
224                     // TODO: we just discard the packet?
225                 }
226             }
227         } else {
228             currentFlows.put(packet.fwdFlowId(), new BasicFlow(bidirectional, packet, this.flowActivityTimeOut));
229         }
230     }
```

Figura 3.7: addpacket pt.4

Come accennato nella sezione 3.2, le condizioni di chiusura dei flussi stabilite dai creatori del tool non sono precise perché possono generare flussi diversi dai reali. Di seguito sono descritti i bug che sono stati individuati nella generazione dei flussi. I primi due sono stati individuati durante l'analisi del codice del tool, mentre gli altri sono stati individuati durante la fase di preprocessing e analisi del dataset, perché sono associati a comportamenti più insoliti.

3.3.1 Chiusura dovuta a FlowTimeout

Come descritto nella sezione 3.2, una condizione per la chiusura di un flusso è data dal superamento del FlowTimeout che di default è pari a due minuti. In realtà un flusso può avere una durata arbitraria e non limitata a soli due minuti. Nel TCP l'unico timeout che fa cadere la connessione, dopo il passare di un certo periodo, è dovuto al timer keepalive che scatta solo dopo che un flusso è rimasto inattivo per 2 ore. Ma il FlowTimeout considera il tempo totale e non il solo tempo di inattività, inoltre ha una durata di gran lunga inferiore rispetto al timer keepalive. Questo meccanismo divide un flusso con durata maggiore di due minuti in tanti flussi, generando un nuovo flusso ogni due minuti. Dato che in realtà il flusso è unico tutti i flussi creati sono spuri e non rispecchiano la situazione reale che è nella cattura dei pacchetti di rete.

```
299    public long dumpLabeledCurrentFlow(String fileFullPath, String header) {
300        if (fileFullPath == null || header == null) {
301            String ex = String.format("fullFilePath=%s,filename=%s", fileFullPath);
302            throw new IllegalArgumentException(ex);
303        }
304
305        File file = new File(fileFullPath);
306        FileOutputStream output = null;
307        int total = 0;
308        try {
309            if (file.exists()) {
310                output = new FileOutputStream(file, true);
311            } else {
312                if (file.createNewFile()) {
313                    output = new FileOutputStream(file);
314                    output.write((header + LINE_SEP).getBytes());
315                }
316            }
317
318            for (BasicFlow flow : currentFlows.values()) {
319                if (flow.packetCount() > 1) {
320                    output.write((flow.dumpFlowBasedFeaturesEx() + LINE_SEP).getBytes());
321                    total++;
322                } else {
323
324                }
325            }
326
327        } catch (IOException e) {
328            logger.debug(e.getMessage());
329        } finally {
330            try {
331                if (output != null) {
332                    output.flush();
333                    output.close();
334                }
335            } catch (IOException e) {
336                logger.debug(e.getMessage());
337            }
338        }
339        return total;
340    }
```

Figura 3.8: dumpLabeledCurrentFlow

3.3.2 Chiusura dovuta a flag FIN

La seconda condizione per la chiusura di un flusso, come descritto nella sezione 3.2, è dovuta alla ricezione di un pacchetto contenente il flag FIN in backward e di un pacchetto contenente il flag FIN in forward. Come già anticipato precedentemente se vengono ricevuti multipli pacchetti contenenti il flag FIN da un lato della connessione (e.g. duplicazione di pacchetti) il flusso non potrà più chiudersi con questa condizione. Potrà chiudersi solo se arriverà un pacchetto dopo due minuti dall'inizio del flusso. Inoltre, dopo che è arrivato un pacchetto contenente il flag FIN qualsiasi altro pacchetto successivo proveniente dallo stesso lato della connessione sarà scartato. Ma nella realtà un pacchetto che è stato inviato successivamente può anche arrivare prima di un pacchetto precedente, se questi seguono route diverse. Quindi questa procedura prevede solo il funzionamento più semplice del protocollo TCP, non considerando pacchetti duplicati, ritardi sulla rete, route diverse, ecc... In questo modo i flussi generati avranno dei pacchetti mancanti rispetto al flusso reale. In realtà la chiusura della connessione in TCP tramite i flag FIN prevede anche la ricezione di un pacchetto con flag ACK dopo ogni rispettivo FIN. Il tool non controlla neppure questa condizione. Questo causa che flussi con lo stesso Flow ID inizino con un pacchetto contenente il flag ACK relativo al flusso precedente.

3.3.3 Flusso che inizia con RST o FIN

Durante la fase di preprocessing del dataset sono stati individuati alcuni flussi che iniziano con un pacchetto contenente il flag RST o FIN. Analizzando il dataset si è notato che sono presenti vari flussi che includono sia pacchetti contenenti flag FIN, ma anche altri pacchetti contenenti flag RST. Il tool quando arriva il pacchetto contenente il flag RST chiude il flusso e lo rimuove dai currentFlows. Se in realtà il flusso non è ancora chiuso e arriva un pacchetto con flag FIN, quest'ultimo viene erroneamente interpretato come il primo pacchetto di un nuovo flusso. Oppure può capitare che arrivino prima i due flag FIN, quindi il flusso viene chiuso, ma successivamente arrivano anche altri pacchetti contenenti flag RST. Se arrivano multipli pacchetti con RST il tool genererà tanti flussi di dimensione pari a due composti da soli flag RST, perché il primo pacchetto sarà considerato come primo pacchetto di un nuovo flusso ed il secondo come la chiusura di esso. Quindi questi flussi spuri creati avranno il primo pacchetto contenente un flag RST o FIN, eventualmente seguito da altri pacchetti di chiusura contenenti flag RST, FIN o ACK. Se poi sullo stesso Flow ID si instaura un nuovo flusso, il tool crea un flusso che in realtà è l'unione della fine del flusso precedente e del flusso attuale.

3.3.4 Pacchetti invertiti

Nel dataset sono presenti anche dei flussi dove sono presenti dei pacchetti invertiti di ordine, ovvero flussi in cui è arrivato prima un pacchetto che in realtà è successivo al pacchetto che è arrivato prima. Un esempio è riportato in figura 3.9 dove si può notare che è arrivato prima un pacchetto contenente flag RST (termine connessione) e successivamente il pacchetto contenente il flag SYN (richiesta di connessione). Il pacchetto con flag RST è successivo al pacchetto con flag SYN dato che ha sequence number e ack number successivi, ma è arrivato prima. Questo tipo di comportamento è probabilmente dovuto a qualche bug nello switch dato che la differenza tra l'arrivo dei pacchetti è nell'ordine dei μs o al massimo dei ms (e.g. preferenza di una porta rispetto all'altra).

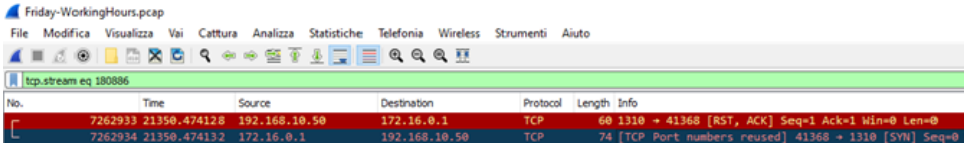


Figura 3.9: Pacchetti invertiti

Un altro esempio è visibile in figura 3.10 dove similmente al caso precedente arriva prima un pacchetto con flag RST e poi un pacchetto con flag SYN. In questo caso però i pacchetti sono duplicati, quindi ci sono due pacchetti con flag RST seguiti da due pacchetti con flag SYN.

Un altro caso incontrato è che arrivi prima la risposta ACK e successivamente la richiesta di connessione SYN, quest'ultimo caso però non crea particolari problemi.

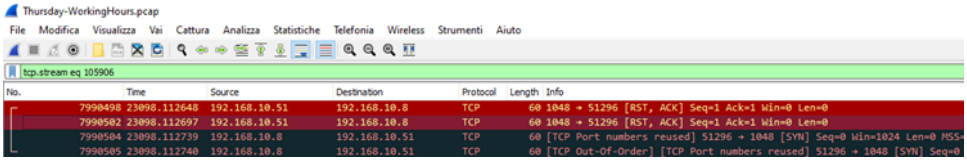


Figura 3.10: Pacchetti duplicati

La maggior parte di questi bug quindi dividono un flusso che in realtà è unico in più flussi, creando dei flussi spuri alle volte anche non conformi al protocollo TCP (e.g. flussi che iniziano con pacchetti con RST). Mentre gli altri bug scartano dei pacchetti dai flussi rendendo il flusso diverso da come è realmente.

3.4 Pipeline

Come descritto precedentemente si vogliono usare i flussi come tracce raggruppando i pacchetti di un determinato flusso come eventi di una certa traccia, tramite un campo chiamato Case ID. Per effettuare questa operazione è stato necessario modificare il tool CICFlowMeter, dato che strumenti come WireShark o Tshark non hanno generato esattamente gli stessi flussi. L'obiettivo è quello di classificare i flussi, tramite la sequenza dei relativi pacchetti, come benevoli e malevoli determinando la tipologia d'attacco nel caso malevolo. Per raggiungere tale obiettivo è stata realizzata la pipeline riportata in figura 3.11.

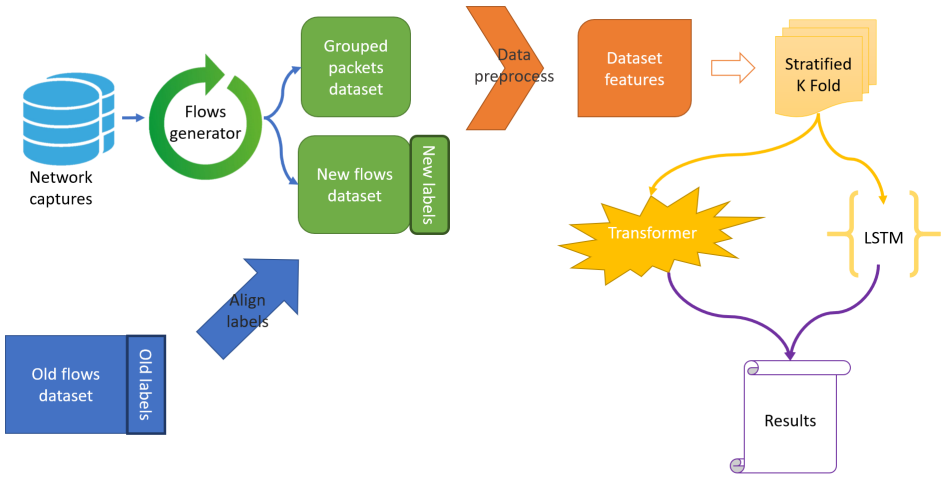


Figura 3.11: Pipeline

Dopo aver modificato il tool CICFlowMeter, per risolvere i bug individuati nel paragrafo 3.3, a partire dalle catture dei pacchetti di rete tramite il tool si otterrà il dataset contenente i flussi corretti chiamato flows dataset. Tra le modifiche apportate al tool è presente la possibilità di creare anche il dataset contenente i pacchetti raggruppati per flusso che sarà chiamato grouped packets dataset o più semplicemente grouped dataset.

Del dataset CICIDS2017 (in figura old flows dataset) originale si vogliono ottenere le classi (o label) ed allinearle al nuovo flows dataset. Quindi si sono svolte delle operazioni per etichettare il nuovo dataset a partire dalle informazioni del vecchio. Come spiegato in precedenza, per la classificazione verranno utilizzati solamente i pacchetti raggruppati per flusso, ovvero il grouped packets dataset. Quindi le classi riallineate verranno aggiunte su quest'ultimo dataset.

Seguiranno le attività di data selection, data preprocessing e feature extraction sul grouped packets dataset. Il dataset risultante, dopo la suddivisone in fold stratificati, verrà utilizzato come input alle reti LSTM(Long Short-Term Memory) e Transformer. I classificatori sono di tipo ensemble, in particolare eseguiranno prima una classificazione binaria per riconoscere come benevolo o malevolo un flusso e successivamente una classificazione multi-label solo sui flussi riconosciuti come attacchi, per riconoscerne la tipologia. A valle della classificazione verranno calcolate le seguenti metriche:

- Accuracy
- Precision
- Recall
- Earliness [29]

4

Data Preprocessing

In questo capitolo viene spiegato come è stato generato il nuovo dataset con particolare enfasi sul come è stato possibile ottenere le classi a partire dal vecchio dataset. Successivamente è presente un'analisi descrittiva del dataset seguita dalla descrizione del preprocess svolto. Infine, è presenta una sezione che affronta come è stato svolto il processo di feature engineering.

4.1 Generazione del Dataset

In questa sezione viene descritta la generazione del dataset, in particolare le modifiche apportate al software CICFlowMeter per generare correttamente i grouped packets.

4.1.1 Creazione del grouped packets dataset

Per quanto riguarda la creazione del grouped packets dataset sono stati aggiunti degli attributi nella classe BasicPacketInfo riguardanti il pacchetto, che nella versione originale del tool non erano presenti:

- *FrameNumber*: identificatore univoco di un frame all'interno della cattura dei pacchetti di rete
- *acknumber*: valore del campo *Acknowledgment number* nell'header TCP
- *seqnumber*: valore del campo *Sequence number* nell'header TCP
- *payload*: payload del segmento TCP o del datagramma UDP

Per valorizzare questi attributi sono stati modificati dei metodi nella classe PacketReader, che come descritto in precedenza, permette di leggere le catture dei pacchetti di rete estraendo i campi desiderati grazie alla libreria jnetpcap. Successivamente, per salvare i pacchetti in formato csv, è stato implementato il metodo toArray all'interno della classe BasicPacketInfo, che crea un array convertendo in stringhe tutti gli attributi di un oggetto BasicPacketInfo e il Case ID associato.

Nello specifico il metodo toArray, mostrato in figura 4.1, necessità di due parametri: il Case ID del flusso e l'indirizzo IP sorgente del flusso. Dopo aver istanziato

```
public String[] toArray(long id, byte[] src) {
    String [] array = new String [24];
    Instant itime = Instant.EPOCH.plus(this.getTimeStamp(), ChronoUnit.MICROS);
    LocalTime time = LocalTime.from(itime.atZone(ZoneId.of("MET")));

    array[0] = Long.toString(id);
    array[1] = flowId;
    array[2] = this.getSourceIP();
    array[3] = this.getDestinationIP();
    array[4] = Integer.toString(srcPort);
    array[5] = Integer.toString(dstPort);
    array[6]= Integer.toString(protocol);
    array[7] = time.toString();
    array[8] = Long.toString(headerBytes);
    array[9] = Long.toString(payloadBytes);
    array[10] =  Integer.toString(payloadPacket);
    array[11] = Integer.toString(TCPWindow);
    array[12] = Integer.toString(Utils.boolToInt(flagFIN));
    array[13] = Integer.toString(Utils.boolToInt(flagPSH));
    array[14] = Integer.toString(Utils.boolToInt(flagURG));
    array[15] = Integer.toString(Utils.boolToInt(flagECE));
    array[16] = Integer.toString(Utils.boolToInt(flagSYN));
    array[17] = Integer.toString(Utils.boolToInt(flagACK));
    array[18] = Integer.toString(Utils.boolToInt(flagCWR));
    array[19] = Integer.toString(Utils.boolToInt(flagRST));
    array[20] = Long.toString(FrameNumber);
    array[21] = Integer.toString(Utils.boolToInt(this.isForwardPacket(src)));
    array[22] = Long.toString(seqnumber);
    array[23] = Long.toString(acknumber);
    //arr[23] = this.getPayload();
    return array;
}
```

Figura 4.1: toArray

l'array di stringhe il metodo recupera il timestamp del pacchetto con precisione del μs e converte in stringhe tutti gli attributi dell'oggetto. Inoltre, confrontando l'indirizzo IP sorgente del flusso con l'indirizzo IP sorgente del pacchetto, calcola se il pacchetto è in forward o in backward rispetto al flusso (campo isForward).

É stata creata una nuova classe chiamata CSVUtils che contiene tutti i metodi implementati per scrivere il grouped dataset e fa uso della libreria opencsv. In particolare il metodo flowtoCSV, dato un flusso e il relativo Case ID, crea un ArrayList contenente tutti i pacchetti del flusso, li ordina per timestamp crescente e successivamente scrive ogni pacchetto su una riga del file csv utilizzando il metodo toArray.

Il metodo flowtoCSV, mostrato in figura 4.2, viene chiamato in tutti i punti dove viene aggiunta una riga nel csv dei flussi, cioè all'interno dei metodi addPacket e dumpLabeledCurrentFlow.

Il Case ID invece, essendo relativo al flusso, è stato aggiunto tra gli attributi della classe basicFlow. L'attributo Case ID viene valorizzato tramite un contatore aggiunto nel metodo addPacket della classse FlowGenerator che viene incrementato ogni volta che un flusso viene chiuso e scritto nel file csv.

Il grouped packets dataset sarà composto quindi del Case ID del flusso, di tutti gli attributi di un oggetto BasicPacketInfo e del campo isForward calcolato nel metodo toArray, come riportato in tabella 4.1.

Non è stato incluso anche l'attributo payload perché avrebbe reso il dataset di dimensioni esagerate, inoltre non sarebbe stato utilizzato per la classificazione essendo una stringa di lunghezza variabile. Comunque nel tool è stata lasciata la

```
45    public void flowToCSV(long id, BasicFlow flow) {
46        ArrayList<BasicPacketInfo> packets = null;
47        packets = new ArrayList<BasicPacketInfo>();
48
49        packets.addAll(flow.getForward());
50        packets.addAll(flow.getBackward());
51
52        Collections.sort(packets, comparing(BasicPacketInfo::getTimeStamp));
53
54        for(BasicPacketInfo packet:packets) {
55            writer.writeNext(packet.toArray(id, flow.getSrc()));
56        }
57    }
```

Figura 4.2: flowtoCSV

Campo	Descrizione
IP Src	indirizzo IP sorgente
IP Dst	indirizzo IP di destinazione
Src Port	porta sorgente
Dst Port	porta di destinazione
Protocol	protocollo di livello trasporto utilizzato
Timestamp	timestamp di arrivo del pacchetto, con precisione del μs
headerBytes	lunghezza dell'header di livello 4 in byte
payloadBytes	lunghezza del payload di livello 4 in byte
payloadPacket	indica se il segmento contiene il payload
TCPWindow	dimensione della finestra TCP
flagFIN	indica se il pacchetto contiene il flag FIN
flagPSH	indica se il pacchetto contiene il flag PSH
flagURG	indica se il pacchetto contiene il flag URG
flagECE	indica se il pacchetto contiene il flag ECE
flagSYN	indica se il pacchetto contiene il flag SYN
flagACK	indica se il pacchetto contiene il flag ACK
flagCWR	indica se il pacchetto contiene il flag CWR
flagRST	indica se il pacchetto contiene il flag RST
FrameNumber	identificatore del frame (livello 2)
isForward	indica se il segmento è forward o backward rispetto al flusso
acknumber	valore del campo *Acknowledgment number* nell'header TCP
seqnumber	valore del campo *Sequence number* nell'header TCP

Tabella 4.1: Campi del grouped packets dataset

possibilità di un futuro utilizzo, de-commentando alcune righe nel codice.

4.1.2 Risoluzione dei bug individuati

Per quanto riguarda le modifiche apportate per risolvere i bug individuati nel paragrafo 3.3 è stata modificata la classe basicFlow aggiungendo, oltre al Case ID, l'attributo *state*: che rappresenta lo stato del flusso. Lo stato del flusso può assumere i seguenti tre valori:

- 0: flusso aperto

- 2: flusso completamente chiuso, sono stati ricevuti 2 flag FIN da entrambi le direzioni o è stato ricevuto un flag RST
- -1: flusso non valido, il primo pacchetto contiene un flag RST o FIN

Anche la classe FlowGenerator è stata modificata per risolvere i bug scovati, in particolare è stato modificato profondamente il funzionamento metodo addPacket.

La condizione di chiusura 1, ovvero durata del flusso che supera il FlowTimeout, è stata mantenuta solo per i flussi con protocollo di trasporto UDP. In questo modo si limita la lunghezza dei flussi UDP e si risolve il problema della suddivisione di un flusso TCP in più flussi TCP. Il codice relativo è riportato in figura 4.3.

```
198             // Flow finished due flowtimeout:
199             // 1.- we move the flow to finished flow list
200             // 2.- we eliminate the flow from the current flow list
201             // 3.- we create a new flow with the packet-in-process
202             else if((packet.getProtocol() != 6) && (currentTimestamp -flow.getFlowStartTime())>flowTimeOut){
203                 if(flow.packetCount()>1){
204                     if (mListener != null) {
205                         flow.setCaseId(count);
206                         mListener.onFlowGenerated(flow);
207                         writer.flowToCSV(count++, flow);
208                     }
209                     else{
210                         finishedFlows.put(getFlowCount(), flow);
211                     }
212                     //flow.endActiveIdleTime(currentTimestamp,this.flowActivityTimeOut, this.flowTimeOut, false);
213                 }
214                 currentFlows.remove(id);
215                 currentFlows.put(id, new BasicFlow(bidirectional,packet,flow.getSrc(),flow.getDst(),flow.getSrcPort(),
216                         flow.getDstPort(), this.flowActivityTimeOut));
217
```

Figura 4.3: addPacket mod pt.1

Per quanto riguarda la chiusura del flusso dovuta alla ricezione dei flag FIN, con la versione originale del tool uno dei problemi è che se un flusso riceve multipli pacchetti contenenti flag FIN dalla stessa direzione, oltre ad impedire la chiusura del flusso con questo criterio, vengono scartati eventuali pacchetti successivi ricevuti da tale direzione. Come si può notare dalla figura 4.4 ora un flusso può ricevere un numero indeterminato di flag FIN da entrambe le direzioni, quando sia il client che il server hanno inviato almeno un flag FIN lo stato del flusso passerà a due. Inoltre non viene scartato nessun pacchetto anche se vengono ricevuti FIN multipli dalla stessa direzione. In questo modo il flusso generato si allinea perfettamente alla cattura dei pacchetti di rete.

Anche la chiusura dovuta al flag RST è stata modificata, come mostrato in figura 4.5

Ora alla ricezione di un pacchetto contenente un flag RST il flusso non viene chiuso immediatamente, ma passa nello stato due. Quindi un flusso può continuare a ricevere eventuali pacchetti, risolvendo il problema della generazione di flussi spuri che iniziano con pacchetti contenenti flag RST o FIN. Entrambi i criteri sono validi se il flusso non inizia con un pacchetto contenente il flag RST o il flag FIN, ovvero se il flusso è in stato diverso da -1 come si vede nelle figure 4.4 e 4.5 Come visto nel paragrafo 3.3, quando l'ordine dei pacchetti viene invertito si può verificare che il primo pacchetto di un flusso contenga un flag RST o FIN. Per risolvere anche quest'ultima problematica sono state aggiunte delle condizioni durante la creazione del flusso, ovvero a seguito della scansione di un pacchetto che non abbia Flow ID

```
112             if((packet.hasFlagFIN()) && (flow.getState() != -1)){
113                 //
114                 // Forward Flow
115                 //
116                 if (Arrays.equals(flow.getSrc(), packet.getSrc())) {
117                     if (flow.setFwdFINFlags() >= 1) {
118                         if (flow.getFwdFINFlags() >= 1 && flow.getBwdFINFlags() >=1) {
119                             flow.updateActiveIdleTime(currentTimestamp,this.flowActivityTimeOut);
120                             flow.addPacket(packet);
121                             flow.setState(2);
122                             currentFlows.put(id,flow);
123                         // Forward Flow Finished.
124                         }
125                         else {
126                             logger.info("Forward flow closed due to FIN Flag");
127                             flow.updateActiveIdleTime(currentTimestamp,this.flowActivityTimeOut);
128                             flow.addPacket(packet);
129                             currentFlows.put(id,flow);
130                         }
131                     }
132                 //
133                 // Backward Flow
134                 //
135                 } else {
136                     if (flow.setBwdFINFlags() >= 1) {
137                         if (flow.getFwdFINFlags() >= 1 && flow.getBwdFINFlags() >=1) {
138                             logger.debug("FlagFIN current has {} flow",currentFlows.size());
139                             flow.updateActiveIdleTime(currentTimestamp,this.flowActivityTimeOut);
140                             flow.addPacket(packet);
141                             flow.setState(2);
142                             currentFlows.put(id,flow);
143                         // Backward Flow Finished.
144                         }
145                         else {
146                             logger.info("Backwards flow closed due to FIN Flag");
147                             flow.updateActiveIdleTime(currentTimestamp,this.flowActivityTimeOut);
148                             flow.addPacket(packet);
149                             currentFlows.put(id,flow);
150                         }
151                     }
152                 }
153             }
```

Figura 4.4: addPacket mod pt.2

```
186             // Flow finished due RST flag (tcp only):
187             // 1.- we add the packet-in-process to the flow (it is the last packet)
188             // 2.- we move the flow to finished flow list
189             // 3.- we eliminate the flow from the current flow list
190             else if((packet.hasFlagRST()) && (flow.getState() != -1)){
191                 logger.debug("FlagRST current has {} flow",currentFlows.size());
192                 count7++;
193                 flow.updateActiveIdleTime(currentTimestamp,this.flowActivityTimeOut);
194                 flow.addPacket(packet);
195                 flow.setState(2);
196                 currentFlows.put(id,flow);
197             }
```

Figura 4.5: addPacket mod pt.3

presente tra i currentFlows. Come mostrato in figura 4.6 se il primo pacchetto del nuovo flusso contiene un flag FIN o RST il flusso viene creato ma lo stato viene settato al valore -1. Se invece il flusso è ancora valido perché l'inversione è solo tra un pacchetto SYN e SYN-ACK il flusso viene creato normalmente, ma il Flow ID viene invertito per tenere traccia della giusta direzione del flusso.

```
223             //State == 1, RST o FIN come primo pacchetto, si scarta il flusso e se sono passati più di 2 secondi si crea un nuovo flusso con il nuovo pacchetto
224             else if((flow.getState() == -1) && (currentTimestamp -flow.getFlowStartTime())>rstTimeOut && (!packet.hasFlagRST()) && (!packet.hasFlagFIN())){
225                 //Il flusso con primo pacchetto RST viene scartato e viene aggiunto un nuovo flusso che non ha come primo pacchetto un RST
226                 currentFlows.remove(id);
227                 currentFlows.put(packet.fwdFlowId(), new BasicFlow(bidirectional,packet, this.flowActivityTimeOut));
228             }
229             else{
230 
231                 flow.updateActiveIdleTime(currentTimestamp,this.flowActivityTimeOut);
232                 flow.addPacket(packet);
233                 currentFlows.put(id,flow);
234             }
235         }
236         else{
237             if(packet.hasFlagSYN() && packet.hasFlagACK()) {
238                 //Il pacchetto contenente SYN-ACK è arrivato prima di quello contenente il SYN, quindi inverto il flusso.
239                 currentFlows.put(packet.bwdFlowId(), new BasicFlow(bidirectional,packet,packet.getDst(),packet.getSrc(),
240                         packet.getDstPort(),packet.getSrcPort(), this.flowActivityTimeOut));
241             }
242 
243             else if(packet.hasFlagRST() || (packet.hasFlagFIN())) {
244                 //Il pacchetto viene "scartato" perché appartiene al flusso precedente (già chiuso), o l'ordine dei pacchetti è invertito. -> state = -1
245                 BasicFlow discarded_flow = new BasicFlow(bidirectional,packet, this.flowActivityTimeOut);
246                 discarded_flow.setState(-1);
247                 currentFlows.put(packet.fwdFlowId(), discarded_flow);
248             }
249 
250             else {
251                 currentFlows.put(packet.fwdFlowId(), new BasicFlow(bidirectional,packet, this.flowActivityTimeOut));
252             }
253         }
254         return id;
255     }
```

Figura 4.6: addPacket mod pt.4

Un flusso che si trova in stato -1 non sarà immediatamente eliminato, perché i pacchetti successivi genererebbero un nuovo flusso, quindi il flusso viene esteso con i nuovi pacchetti ad esso relativi. In questo modo però è presente un problema: qualsiasi altro flusso che abbia lo stesso Flow ID sarà scartato. Per ovviare a questo problema si è inserito un attributo chiamato rstTimeOut con un valore pari a due secondi. In questo modo se dovesse arrivare, dopo due secondi dall'inizio del flusso, un pacchetto con lo stesso Flow ID, ma in realtà appartenente ad un altro flusso, il flusso precedente verrà eliminato dai currentFlows e al suo posto sarà inserito il nuovo flusso, come mostrato in figura 4.7.

```
218             //State == 1, RST o FIN come primo pacchetto, si scarta il flusso e se sono passati più di 2 secondi si crea un nuovo flusso con il nuovo pacchetto
219             else if((flow.getState() == -1) && (currentTimestamp -flow.getFlowStartTime())>rstTimeOut && (!packet.hasFlagRST()) && (!packet.hasFlagFIN())){
220                 //Il flusso con primo pacchetto RST viene scartato e viene aggiunto un nuovo flusso che non ha come primo pacchetto un RST
221                 currentFlows.remove(id);
222                 currentFlows.put(packet.fwdFlowId(), new BasicFlow(bidirectional,packet, this.flowActivityTimeOut));
223             }
```

Figura 4.7: addPacket mod pt.5

Questo però solo se il nuovo pacchetto ovviamente non contiene a sua volta il flag RST o il flag FIN. Quindi se un flusso è in stato -1 non potrà chiudersi e rimarrà nei flussi correnti, a meno del soddisfacimento della condizione appena descritta. Per scartare quindi questi flussi il metodo dumpLabeledCurrentFlow è stato modificato in modo tale da non scrivere i flussi correnti con state -1.

L'ultima modifica apportata è proprio nella chiusura dei flussi, infatti finora si è visto che un flusso cambia il valore dell'attributo state ma comunque non viene

chiuso e salvato nel csv. Un flusso che ha ricevuto due FIN bidirezionali o un RST, può continuare a riceve ulteriori pacchetti anche di acknowledgment, o ritardati o duplicati come visto nel paragrafo 3.3. Il tool originale crea dei flussi spuri contenenti RST, FIN e ACK dovuti ad una chiusura anticipata, perché bastano un RST o 2 FIN bidirezionali per rientrare tra le condizioni di chiusura dei flussi del tool. Inoltre, il tool non prende in considerazione l'ACK finale dopo la ricezione dei due flag FIN bidirezionali. Per risolvere tutti questi problemi si è posticipata la chiusura della connessione il più possibile. In particolare quando un flusso è in stato due, qualsiasi pacchetto che non contenga il flag SYN viene aggiunto al flusso. In questo modo non solo viene aggiunto l'ultimo pacchetto contenente l'acknowledgment, ma anche eventuali pacchetti duplicati o ritardati saranno inclusi all'interno del flusso. Il codice è mostrato in figura 4.8.

```
154             // Aggiunto per fine TCP dopo ultimo ACK
155             else if(flow.getState()==2){
156                 if (packet.hasFlagSYN()){
157                     if (mListener != null) {
158                         flow.setCaseId(count);
159                         mListener.onFlowGenerated(flow);
160                         writer.flowToCSV(count++, flow);
161                     } else {
162                         finishedFlows.put(getFlowCount(), flow);
163                     }
164                     currentFlows.remove(id);
165 
166                     if(packet.hasFlagACK()) {
167                         currentFlows.put(packet.bwdFlowId(), new BasicFlow(bidirectional,packet,packet.getDst(),packet.getSrc(),
168                                 packet.getDstPort(),packet.getSrcPort(), this.flowActivityTimeOut));
169                     }
170                     else {
171                         currentFlows.put(packet.fwdFlowId(), new BasicFlow(bidirectional,packet, this.flowActivityTimeOut));
172                     }
173                 }
174                 else {
175                     flow.updateActiveIdleTime(currentTimestamp,this.flowActivityTimeOut);
176                     flow.addPacket(packet);
177                     currentFlows.put(id,flow);
178                 }
179             }
```

Figura 4.8: addPacket mod pt.6

Se invece il pacchetto contiene il flag di richiesta di una nuova connessione allora il flusso viene chiuso, eliminato dai currentsFlow e sarà creato ed inserito il nuovo flusso. Se il pacchetto oltre al flag SYN contiene anche l'ACK la procedura sarà la stessa, ma il Flow ID sarà invertito come spiegato precedentemente.

Quindi a differenza del tool originale si è completamente cambiato l'approccio della chiusura dei flussi. Ogni flusso viene tenuto tra i flussi correnti anche dopo che è stato chiuso perché possono arrivare ulteriori pacchetti. Tutte queste modifiche risolvono completamente i bug descritti precedentemente e rendono i flussi generati coerenti con i flussi delle catture dei pacchetti di rete. Ovviamente servirà uno spazio di allocazione maggiore per l'attributo currentFlows dato che conterrà più flussi rispetto al caso originale.

4.2 Etichettatura del dataset

La versione utilizzata dai creatori del dataset per generare i flussi è obsoleta rispetto a quella utilizzata e descritta nel paragrafo 3.2, inoltre sono state anche apportate

le modifiche descritte nel paragrafo 4.1 quindi i flussi del flows dataset obsoleto sono differenti rispetto ai flussi del flows dataset generato. Però, come spiegato nel paragrafo 3.4, il flows dataset obsoleto contiene le etichette, quindi in questo paragrafo è illustrato come sono state allineate le classi al nuovo flows dataset.

Inizialmente è stato utilizzato un approccio basato sul left join tra il nuovo dataset ed il vecchio dataset, per ottenere le etichette. Ma questo approccio presentava alcuni problemi in quanto per alcuni flussi nel nuovo dataset non erano presenti i corrispondenti flussi nel vecchio dataset, a causa delle differenze tra i tool.

4.2.1 Flussi con label dubbia

Nel sito web dove è pubblicato il dataset sono riportati tutti gli attacchi effettuati con dettagli relativi agli orari di attacco e agli indirizzi IP delle macchine attaccanti e delle macchine vittime.

Si è verificata la validità di queste informazioni controllandone la corrispondenza con il dataset.

Nella tabella 4.2 sono riportati gli orari di inizio e fine degli attacchi presenti nel sito web e nel dataset.

Attacco	**Orario sito web**	**Orario nel dataset**
Bot	10:02 - 11:02	09:34 - 12:59
PortScan	01:55 - 03:29	01:05 - 03:23
DDoS	03:56 - 04:16	03:56 - 04:16
Web Attack - Brute Force	9:20 - 10:00	09:15 - 10:00
Web Attack - XSS	10:15 - 10:35	10:15 - 10:35
Web Attack - Sql Injection	10:40 - 10:42	10:40 - 10:42
FTP-Patator	9:20 - 10:20	09:17 - 10:30
SSH-Patator	02:00 - 03:00	02:09 - 03:11
DoS slowloris	9:47 - 10:10	02:24 - 10:11
DoS Slowhttptest	10:14 - 10:35	10:15 - 10:37
DoS Hulk	10:43 - 11:00	10:43 - 11:07
DoS GoldenEye	11:10 - 11:23	11:10 - 11:19
Heartbleed	03:12 - 03:32	03:12 - 03:32
Infiltration	02:19 - 03:45	02:19 - 03:45

Tabella 4.2: Confronto periodo degli attacchi tra sito web e dataset

Come è possibile notare la maggior parte degli orari coincidono, a meno di piccole differenze. Ma negli attacchi Bot, PortScan e DoS Slowloris le differenze sono significative.

Anche gli indirizzi IP degli attaccanti e delle vittime presenti nel sito sono stati confrontati con quelli presenti nel dataset. Tutti gli indirizzi IP corrispondono, a meno di alcuni flussi negli attacchi DDoS e Bot.

I flussi DoS slowloris sono stati eseguiti tra le 9:47 e le 10:10, ma sono presenti sei flussi fuori orario, di cui cinque tra le 02:24 e uno alle 09:01 come riportato in tabella 4.3. Non è possibile sapere se questi flussi siano stati realmente degli attacchi DoS slowloris o se siano stati mal etichettati.

Flow ID	Time	Length
172.16.0.1-192.168.10.50-49631-80-6	02:24	14
172.16.0.1-192.168.10.50-49631-80-6	02:24	2
172.16.0.1-192.168.10.50-49632-80-6	02:24	4
172.16.0.1-192.168.10.50-49633-80-6	02:24	15
172.16.0.1-192.168.10.50-49633-80-6	02:25	2
172.16.0.1-192.168.10.50-53058-80-6	09:01	4

Tabella 4.3: Flussi anomali DoS slowloris

I flussi Port Scan presenti nel flows dataset che hanno orari incongruenti rispetto a quanto descritto nel sito sono riportati in tabella 4.4.

Flow ID	Time	Length
172.16.0.1-192.168.10.50-40620-80-6	01:05	11
172.16.0.1-192.168.10.50-40620-80-6	01:05	2
172.16.0.1-192.168.10.50-40622-80-6	01:05	11
172.16.0.1-192.168.10.50-40622-80-6	01:05	2
172.16.0.1-192.168.10.50-40626-80-6	01:05	4
172.16.0.1-192.168.10.50-40628-80-6	01:06	11
172.16.0.1-192.168.10.50-40628-80-6	01:06	2
172.16.0.1-192.168.10.50-40718-80-6	01:52	4

Tabella 4.4: Flussi anomali Port Scan

In questo caso i flussi anomali sono sette, di cui sei tra le 01:05 e le 01:06 e uno alle 01:52. Anche in questo caso non si riesce a stabilire se questi flussi siano stati etichettati male o se siano realmente dei flussi di attacchi Port Scan.

I flussi Botnet anomali sono riportati in tabella 4.5.

Flow ID	Time	Length
192.168.10.12-52.6.13.28-42544-8080-6	09:34	18
192.168.10.12-52.6.13.28-42544-8080-6	09:35	2
192.168.10.17-52.7.235.158-48034-8080-6	11:20	20
192.168.10.17-52.7.235.158-48034-8080-6	11:21	2

Tabella 4.5: Flussi anomali Botnet

I primi due flussi sono in anticipo rispetto a quanto dichiarato nel sito, perché gli attacchi Botnet dovrebbero iniziare alle 10:02. Tutti i quattro flussi hanno indirizzo IP attaccante e indirizzo IP vittima differenti rispetto agli indirizzi IP riportati nel sito. Quindi molto probabilmente c'è stato un errore di etichettatura, dato che solo quattro flussi su quasi 2000 flussi Botnet hanno indirizzi IP differenti. Poi sono presenti anche migliaia di flussi Bot con orario compreso tra le 11:02 e le 12:59, quindi dopo l'orario riportato nel sito, però tutti questi flussi hanno indirizzi IP legittimi e caratteristiche simili.

Per quanto riguarda l'attacco DDoS i flussi anomali trovati nel dataset sono riportati in tabella 4.6.

Flow ID	Ip Src	Ip Dst	Time	Length
172.16.0.1-192.168.10.50-64869-80-6	192.168.10.50	172.16.0.1	04:06	7
172.16.0.1-192.168.10.50-64873-80-6	192.168.10.50	172.16.0.1	04:06	6
172.16.0.1-192.168.10.50-27636-80-6	192.168.10.50	172.16.0.1	04:06	6

Tabella 4.6: Flussi anomali DDoS

Questi flussi hanno degli orari compatibili con quelli nel sito, ma partono dall'indirizzo IP vittima. Controllando le catture dei pacchetti di rete si è scoperto che tutti questi flussi sono stati suddivisi erroneamente dal tool, in quanto in realtà il flusso inizia dalla macchina attaccante e finisce ricevendo due pacchetti FIN. La vecchia versione del tool, che è stata utilizzata per la creazione del dataset, considerava un flusso chiuso alla ricezione di un solo flag FIN. Questi flussi infatti iniziano con il pacchetto contenente il secondo flag FIN, che viene inviato dalla vittima. Ovviamente nel nuovo dataset non sono presenti questi problemi avendo una versione aggiornata e modificata appositamente del tool.

A partire da alcune analisi sul dataset sono emerse delle stranezze riguardanti i flussi benigni. I pattern di questi flussi anomali presentano delle similitudini con i flussi PortScan e controllando meglio sul sito web del dataset si è scoperto che è presente un secondo step dell'attacco Infiltration, ma non è riportato l'orario d'esecuzione. Controllando l'indirizzo IP sorgente di questi flussi anomali si è scoperto che era l'indirizzo IP della vittima nell'attacco Infiltration. Questo perché il secondo step dell'attacco Infiltration consiste nell'effettuare il port scanning tramite la macchina vittima, che diventa quindi l'attaccante.

Tutti questi flussi sono incerti, quindi sono stati etichettati come UNKNOWN.

4.2.2 Script labellatore

É stato creato uno script labellatore in Python, con l'utilizzo della libreria Pandas, che aggiunge la classe ad ogni flusso presente nel flows dataset a partire dalle informazioni sugli orari e sugli indirizzi IP.

In figura 4.9 sono riportate le definizioni dei dizionari utilizzati dallo script. Il dizionario *dict_attacks* contiene come key le tipologie degli attacchi e come values: l'indirizzo IP sorgente, l'indirizzo IP destinazione e gli orari di inizio e di fine per un determinato attacco. Tale dizionario è stato valorizzato a partire dalla tabella 4.2 escludendo i flussi UNKNOWN, prendendo l'intervallo di orari del sito web se uguale all'intervallo del dataset o l'intervallo maggiore tra i due se è presente una leggera differenza. Solo per gli attacchi Botnet è stato prolungato l'orario fino all'orario di fine trovato nel dataset, perché la classe Botnet è poco popolata e sono presenti migliaia di flussi dopo l'orario finale del sito web. Inoltre tutti questi flussi hanno caratteristiche simili, quindi probabilmente è stata una dimenticanza di chi ha riportato gli orari nel sito web. Il dizionario *dict_unknowns* ha la stessa struttura del dizionario *dict_attacks* ma contiene solo le informazioni dei flussi UNKNOWN.

Il dizionario *dict_map* serve per aggiungere ai Case ID un prefisso variabile in base al giorno di cattura dei vari file csv, in modo tale da evitare Case ID duplicati essendo un identificatore numerico. Tutti gli attacchi tranne Bot e Infiltration partono da un singolo indirizzo IP sorgente, perché è presente una NAT. Invece i flussi relativi agli attacchi Bot e Infiltration partendo dai computer delle vittime avranno indirizzi IP sorgenti differenti. Per tale motivo questi due attacchi nel dizionario non contengono il campo IP Source ma solo l'indirizzo IP di destinazione che è unico. Ovviamente è stato controllato che nel dataset non sono presenti altri flussi etichettati diversamente che partono da un qualsiasi indirizzo IP e arrivano a quello specifico indirizzo IP di destinazione in quei determinati intervalli di tempo.

```
# Dizionario degli attachi: IP SRC, IP DST, time inizio e time fine
dict_attacks = {"DoS slowloris": ["172.16.0.1", "192.168.10.50", "09:47", "10:11"],
                "DoS Slowhttptest": ["172.16.0.1", "192.168.10.50", "10:14", "10:37"],
                "DoS Hulk": ["172.16.0.1", "192.168.10.50", "10:43", "11:07"],
                "DoS GoldenEye": ["172.16.0.1", "192.168.10.50", "11:10", "11:23"],
                "Heartbleed": ["172.16.0.1", "192.168.10.51", "03:12", "03:32"],
                "FTP-Patator": ["172.16.0.1", "192.168.10.50", "09:17", "10:30"],
                "SSH-Patator": ["172.16.0.1", "192.168.10.50", "02:00", "03:11"],
                "Web Attack - Brute Force": ["172.16.0.1", "192.168.10.50", "09:15", "10:00"],
                "Web Attack - XSS": ["172.16.0.1", "192.168.10.50", "10:15", "10:35"],
                "Web Attack - Sql Injection": ["172.16.0.1", "192.168.10.50", "10:40", "10:42"],
                "Bot": ["", "205.174.165.73", "10:02", "12:59"], "DDoS": ["172.16.0.1", "192.168.10.50", "03:56", "04:16"],
                "PortScan": ["172.16.0.1", "192.168.10.50", "01:55", "03:29"],
                "Infiltration": ["", "205.174.165.73", "02:19", "03:45"]}

# Dizionario di flussi incerti, non presenti nella tabella ma sul dataset originale si o viceversa
dict_unknowns = {"PortScan": ["172.16.0.1", "192.168.10.50", "01:05", "01:06", "01:52"],
                 "DoS slowloris": ["172.16.0.1", "192.168.10.50", "02:24", "02:25", "09:01"],
                 "Bot": [["192.168.10.12", "52.6.13.28", "09:34"], ["192.168.10.17", "52.7.235.158", "11:20"]],
                 "Infiltration": ["", "192.168.10.8", "03:04", "03:45"]}

dict_map = {"Monday": "a", "Tuesday": "b", "Wednesday": "c", "Thursday": "d", "Friday": "e"}
```

Figura 4.9: Labellatore pt.1

Tramite la libreria Pandas viene creato un dataframe per ogni file csv, vengono tipizzate correttamente alcune colonne del dataframe e viene aggiunto il prefisso al Case ID tramite il dizionario *dict_map*. Successivamente vengono aggiunte due colonne al dataframe:

- Timestamp_new: data calcolata sottraendo 5 ore dalla colonna Timestamp
- Time: calcolata come la colonna Timestamp_new ma contenente solo gli orari

Vengono sottratte 5 ore perché c'è una diversa gestione del fuso orario che provoca una differenza pari a 5 ore tra i flussi generati con il vecchio CICFlowMeter, e quindi anche negli orari riportati nel sito web, e la nuova versione.

Successivamente viene controllato se il csv che si sta leggendo è relativo al giorno Monday e in caso positivo viene aggiunto il label BENIGN a tutti i flussi, come raffigurato in figura 4.11. Perché lunedì sono state effettuate solo catture di flussi benevoli. In caso negativo viene invece creato un altro dataframe a partire dal corrispettivo csv del flows dataset obsoleto e per ogni attacco contenuto vengono selezionati i flussi che corrispondono alle condizioni identificate con il dizionario dict_attacks. Dopo aver selezionato le righe nel nuovo dataframe viene assegnata la classe corrispondente alle condizioni nel dizionario.

```
for elem in os.listdir(path_old):
    day = elem.split("-")[0]  # Prendo il giorno
    print(day)
    df_new = pd.read_csv(path_new + elem, header=0, index_col=0)
    df_new["Src Port"] = df_new["Src Port"].astype(int)
    df_new["Dst Port"] = df_new["Dst Port"].astype(int)
    df_new["Protocol"] = df_new["Protocol"].astype(int)
    df_new.index = dict_map[day] + df_new.index.astype(str)

    df_new["Timestamp1"] = pd.to_datetime(
        pd.to_datetime(df_new["Timestamp"], format="%d/%m/%Y %I:%M:%S %p") - pd.DateOffset(hours=5)).dt.strftime("%d/%m/%Y %I:%M")
    df_new["time"] = pd.to_datetime(
        pd.to_datetime(df_new["Timestamp"], format="%d/%m/%Y %I:%M:%S %p") - pd.DateOffset(hours=5)).dt.strftime("%I:%M")

    if "Monday" not in elem:
        df_old = pd.read_csv(path_old + elem, header=0, index_col=0)

        for attack in df_old["Label"].unique():
            if attack != "BENIGN":
                a = dict_attacks[attack][2]
                b = dict_attacks[attack][3]

                if attack != "Bot" and attack != "Infiltration":
                    df_new.loc[(df_new["Src IP"] == dict_attacks[attack][0]) & (df_new["Dst IP"] == dict_attacks[attack][1]) & (
                            df_new["time"] >= a) & (df_new["time"] <= b), "Label"] = attack
                else:
                    df_new.loc[(df_new["Dst IP"] == dict_attacks[attack][1]) & (df_new["time"] >= a) & (df_new["time"] <= b)
                        , "Label"] = attack
```

Figura 4.10: Labellatore pt.2

```
    # se il giorno è Monday
    else:
        df_new["Label"] = "BENIGN"
        df_new.to_csv("F:/Dataset/CICIDS2017/interleaved/days/" + elem + ".csv", index=True)
        new_df_full = new_df_full.append(df_new)
```

Figura 4.11: Labellatore pt.3

In figura 4.12 è mostrato l'uso del dizionario dict_unknowns, il codice è simile al precedente, infatti vengono etichettati come UNKNOWN tutti i flussi del dataframe che corrispondono alle condizioni contenute nel dict_unknowns.

```
            # Aggiunta degli UNKNOWNS
            if attack in dict_unknowns:
                if attack == "Bot":
                    for row in dict_unknowns[attack]:
                        df_new.loc[((df_new["Src IP"] == row[0]) & (df_new["Dst IP"] == row[1]) | (df_new["Dst IP"] == row[0]) &
                                    (df_new["Src IP"] == row[1])) & (df_new["time"] == row[2]), "Label"] = "UNKNOWN"
                elif attack == "Infiltration":
                    df_new.loc[((df_new["Dst IP"] == dict_unknowns[attack][1]) | (df_new["Src IP"] == dict_unknowns[attack][1])) &
                               (df_new["time"] >= dict_unknowns[attack][2]) &
                               (df_new["time"] <= dict_unknowns[attack][3]), "Label"] = "UNKNOWN"
                else:
                    for x in range(2, len(dict_unknowns[attack])):
                        df_new.loc[((df_new["Src IP"] == dict_unknowns[attack][0]) & (df_new["Dst IP"] == dict_unknowns[attack][1])
                        | (df_new["Dst IP"] == dict_unknowns[attack][0]) & (df_new["Src IP"] == dict_unknowns[attack][1]))
                                   & (df_new["time"] == dict_unknowns[attack][x]), "Label"] = "UNKNOWN"

    df_new.loc[df_new["Label"] == "NeedManualLabel", "Label"] = "BENIGN"
    df_new.to_csv("F:/Dataset/CICIDS2017/interleaved/days/" + elem + ".csv", index=True)
    new_df_full = new_df_full.append(df_new)
```

Figura 4.12: Labellatore pt.4

Dopo aver effettuato questo processo per tutti i file csv, il flows dataset generato

con la nuova versione del tool contiene anche le classi ricavate a partire dal flows dataset obsoleto.

4.2.3 Flows Dataset etichettato

In tabella 4.7 è raffigurata la distribuzione delle classi nel Flows dataset aggiornato, come si può notare rispetto alla distribuzione del flows dataset originale, riportata in tabella 3.3 ci sono 800.000 flussi in meno, ma questi flussi non sono realmente mancanti dato che sono flussi spuri. Infatti, rispetto alla distribuzione del flows dataset originale 3.3 è possibile notare come vari attacchi abbiano meno flussi, ma ovviamente il numero di pacchetti è lo stesso. Quindi la media di pacchetti per flusso del nuovo dataset è aumentata. Ma sono presenti anche attacchi che contengono più flussi rispetto al dataset originale. Questo è dovuto al fatto che il tool usato per generare i flussi nel primo dataset non utilizzava la chiusura dovuta ai flag RST, accorpando quindi flussi diversi in uno unico. I flussi UNKNOWN trovati invece sono più di 73.000.

Label	**Conteggio flussi**
BENIGN	1.519.753
PortScan	159.148
DoS Hulk	154.319
DDoS	95.683
UNKNOWN	73.125
DoS GoldenEye	7.495
DoS Slowhttptest	4.217
FTP-Patator	3.993
DoS slowloris	3.894
SSH-Patator	2.979
Bot	2.206
Web Attack – Brute Force	1.367
Web Attack – XSS	679
Infiltration	45
Web Attack – Sql Injection	12
Heartbleed	1
Totale	**2.028.916**

Tabella 4.7: Distribuzione delle classi nel flows dataset

4.3 Analisi descrittiva

In questa sezione è descritta una breve analisi descrittiva eseguita sul flows dataset.

Come prima analisi si è ottenuta la percentuale di flussi che hanno un determinato protocollo di trasporto, come riportato in figura 4.13.

Come si può notare quasi il 51% dei flussi hanno protocollo 17 ovvero UDP, quasi il 49% ha protocollo 6 ovvero TCP e una percentuale trascurabile ha protocollo

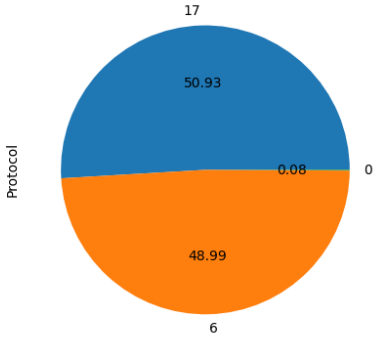


Figura 4.13: Protocolli di rete

0 cioè HOPOPT. L'HOPOPT è un'opzione dell'IPv6 che consente di aggiungere ulteriori opzioni in un pacchetto IPv6. In particolare i flussi sono così suddivisi: 996139 UPD, 958077 TCP, 1575 HOPOPT.

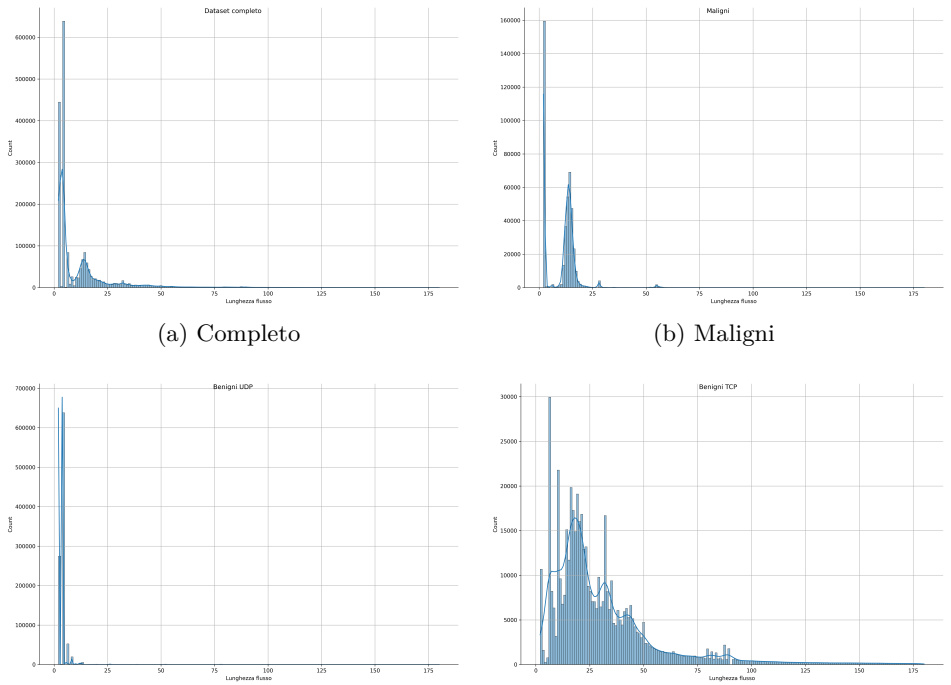
Per quanto riguarda gli attacchi i flussi sono tutti TCP, tranne per l'attacco PorstScan che contiene anche un flusso UDP e tre flussi HOPOPT.

Successivamente si sono analizzate le distribuzioni delle lunghezze dei flussi, al fine di verificare se ogni classe ha caratteristiche differenti rispetto alle altre o no. In figura 4.14 sono riportate le distribuzioni di specifiche sezioni del dataset, gli istogrammi arrivano fino al valore massimo 180 per mostrare le distribuzioni che altrimenti non si vedrebbero. In figura 4.14a è riportata la distribuzione delle lunghezze dei flussi di tutto il dataset. La distribuzione è del tipo power law ma è presente anche una porzione dove la distribuzione è normale e come si può notare dalle figure 4.14b, 4.14d tale porzione è dovuta ai flussi maligni e ai flussi benigni TCP. Per quanto riguarda i flussi benigni UDP invece la distribuzione è solamente power law ed è riportata in figura 4.14c.

Successivamente sono state analizzate le singole distribuzioni di ogni tipologia di attacco. In figura 4.15 sono riportate le distribuzioni relative agli attacchi DDoS, DoS Slowhttptest, SSH-Patator e DoS GoldenEye. Mentre le altre distribuzioni relative agli attacchi non riportati sono disponibili nella sezione dell'appendice A.1.

I flussi relativi all'attacco DDoS hanno una distribuzione normale centrata sul valor medio che è circa 13, anche i flussi dell'attacco DoS Goldeneye hanno una distribuzione simile ma con valor medio e deviazione standard maggiori, inoltre la coda destra è più estesa e i flussi nell'intorno dei valori 10-11 hanno una frequenza maggiore rispetto a quelli successivamente adiacenti. L'attacco SSH-Patator ha una distribuzione molto stretta centrata nel valor medio di circa 55, mentre l'attacco DoS Slowhttptest ha una distribuzione con una varianza più elevata e con una frequenza maggiore nella parte sinistra rispetto alla destra.

Sono state calcolate media, moda, deviazione standard e massimo della lunghezza dei flussi anche in backward e forward per ogni classe, come riportato in tabella



(a) Completo (b) Maligni

(c) Benigni UDP (d) Benigni TCP

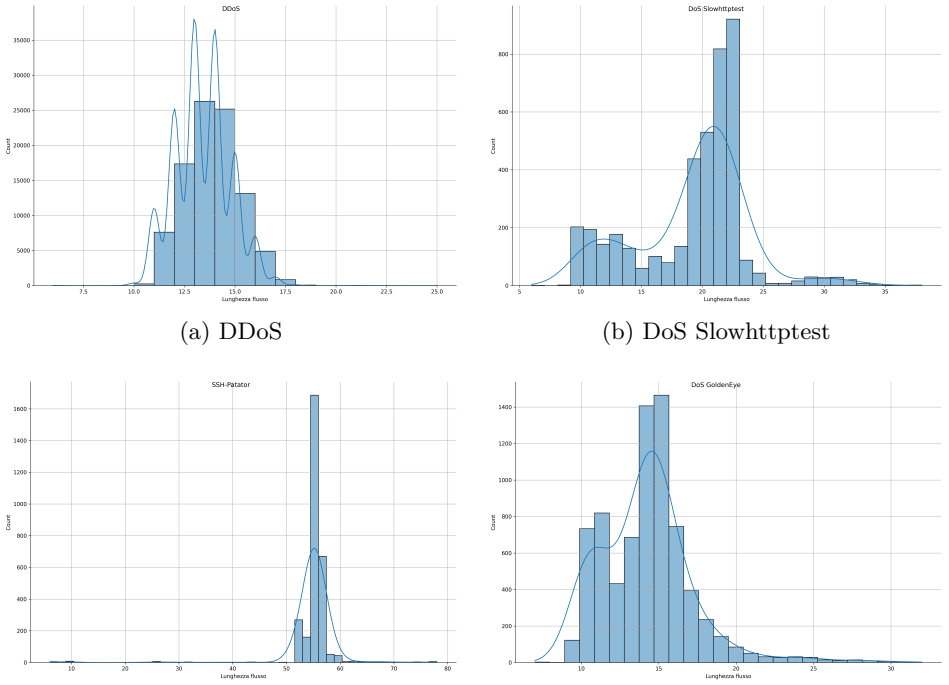
Figura 4.14: Distribuzione lunghezza flussi pt.1

4.8.

L'attacco Heartbleed ha un solo flusso, per questo motivo non ha deviazione standard. Dalla tabella è possibile ricavare alcune informazioni interessanti, in media un flusso benigno ha una lunghezza maggiore rispetto ad un flusso maligno e in media ha più pacchetti in backward al contrario di un flusso maligno che in media contiene più pacchetti in forward. Anche la deviazione standard è molto differente, infatti gli attacchi hanno una deviazione standard molto inferiore rispetto ai flussi benigni. Confrontando invece i flussi benigni TCP e UDP è possibile notare come i flussi UDP abbiano in media una dimensione di gran lunga inferiore rispetto alla dimensione dei flussi TCP e anche i rispettivi flussi con lunghezza massima hanno ordini di grandezza differenti. Anche la deviazione standard quindi è differente risultando molto minore per i flussi UDP, come già visto nelle distribuzioni in figura 4.14. Per quanto riguarda gli attacchi è possibile notare come ogni singolo attacco seguendo un determinato pattern abbia in media caratteristiche diverse dagli altri. Ad esempio i flussi dell'attacco Port Scan hanno quasi tutti una lunghezza pari a due dato che la media è molto vicino a questo valore e la moda è due, mentre l'attacco FTP-Patator ha in media una lunghezza di 28 pacchetti e anche la moda è pari a 28. Mentre ci sono attacchi che hanno una distribuzione più variabile come ad esempio l'attacco Infiltrated che presenta la deviazione standard maggiore tra gli attacchi, una media pari a 1635 ma una moda pari a 2 oppure come l'attacco

Label	avg	fwd avg	bwd avg	mode	fwd mode	bwd mode	std	fwd std	bwd std	max	fwd max	bwd max
Bot	5.81	2.84	2.97	2	1	1	11.54	4.21	7.47	125	40	86
PortScan	2.02	1.02	1	2	1	1	0.56	0.44	0.15	180	150	30
DDoS	13.38	8	5.39	13	8	6	1.36	0.79	0.93	25	14	13
Web Attack - Brute Force	22.14	14.54	7.6	6	4	2	67.18	44.36	22.83	311	205	107
Infiltration	1635.84	869.44	766.4	2	1	1	6611.96	3425.17	3219.59	36571	18284	18287
Web Attack - XSS	14.19	9.37	4.83	6	4	2	48.71	32.13	16.58	321	214	115
Web Attack - Sql Injection	10.5	5.58	4.92	10	5	5	0.8	0.67	0.51	12	7	6
FTP-Patator	27.95	10.99	16.96	28	11	17	0.77	0.22	0.57	30	12	18
SSH-Patator	54.82	22.05	32.77	55	22	33	4.92	2.12	3.02	78	34	44
DoS slowloris	18.37	12.99	5.38	17	14	3	4.6	2.91	3.3	44	27	17
DoS Slowhttptest	19.17	13.12	6.06	21	15	6	4.51	4.14	1.26	38	27	13
DoS Hulk	14.14	7.86	6.28	14	8	6	1.68	1.21	0.96	41	27	15
DoS GoldenEye	14.14	8.9	5.24	15	9	5	3.03	2.2	1.12	32	22	14
Heartbleed	49296	28412	20884	49296	28412	20884				49296	28412	20884
Completo	28.55	13.36	15.19	4	2	2	4474.42	1912.13	2562.67	2466364	1046341	1420035
BENIGN	33.78	15.53	18.25	4	2	2	5075.58	2168.96	2907.04	2466364	1046341	1420035
BENIGN TCP	90.65	40.93	49.72	6	4	7	8659.73	3700.56	4959.88	2466364	1046341	1420035
BENIGN UDP	3.85	2.06	1.79	4	2	2	9.73	4.94	5.87	4231	1431	2800
Attacks	10.31	5.78	4.52	2	1	1	101.62	55.99	46.06	49296	28412	20884

Tabella 4.8: Statistiche lunghezza flussi



(a) DDoS

(b) DoS Slowhttptest

(c) SSH-Patator

(d) DoS GoldenEye

Figura 4.15: Distribuzione lunghezza flussi pt.2

Web Attack - Brute Force che ha una media pari a 26 ma una moda pari a 6. In sostanza quindi è possibile notare come ogni tipologia di attacco abbia caratteristiche differenti dagli altri per quanto riguarda la distribuzione di lunghezza dei flussi.

Per svolgere la successiva analisi si è aggiunta la colonna event nel grouped packets dataset,che è formato dai campi contenuti nella tabella 4.1. La colonna event contenente i nomi dei flag presenti in un determinato pacchetto. Tramite il software Disco si sono ottenuti i direct follower graph dei vari attacchi.

In figura 4.16 è riportato il grafo dell'attacco Bot, la maggior parte dei flussi dopo la richiesta di connessione vengono conclusi tramite il flag RST, per questo la moda di questo attacco è pari a due. Comunque circa 1/3 dei flussi totali riesce a comunicare e dopo vari scambi di pacchetti contenenti flag ACK il flusso viene chiuso tramite lo scambio di flag FIN.

In figura 4.17 è riportato il grafo dell'attacco XSS, come si può notare tutti i flussi riescono a connettersi, a scambiare pacchetti ed infine a concludersi tramite l'invio dei flag FIN. Un solo flusso ha ricevuto un flag RST, ma solo dopo aver ricevuto i due flag FIN.

In figura 4.18 è riportato il grafo dell'attacco DoS Hulk, questo attacco presenta più possibili combinazioni rispetto agli attacchi precedenti e presenta anche l'evento FINPSHACK non contenuto nei precedenti grafi. I Direct Follower Graph di tutti gli altri attacchi sono riportati nella sezione A.2 dell'appendice.

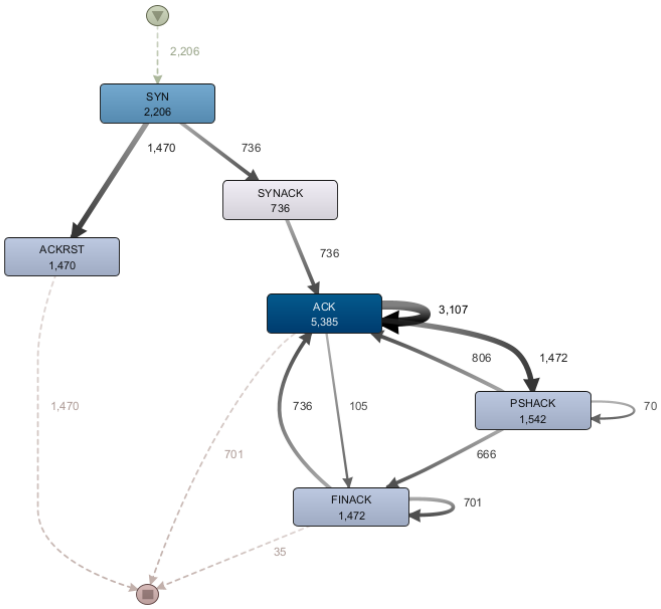


Figura 4.16: Direct Follower Graph Botnet

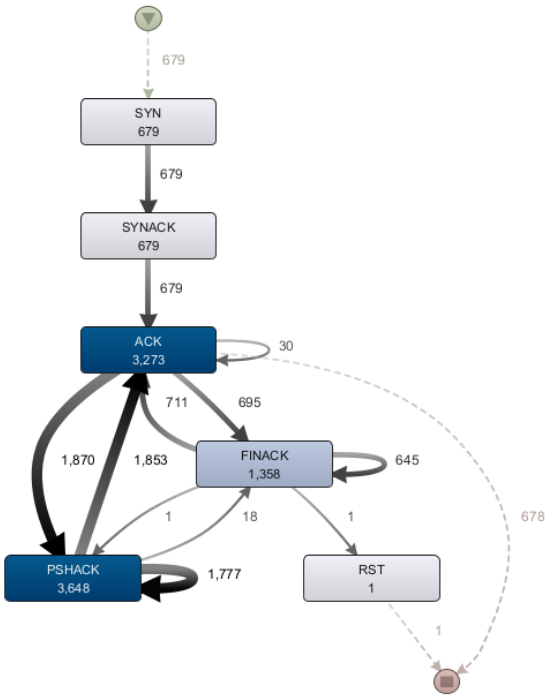


Figura 4.17: Direct Follower Graph XSS

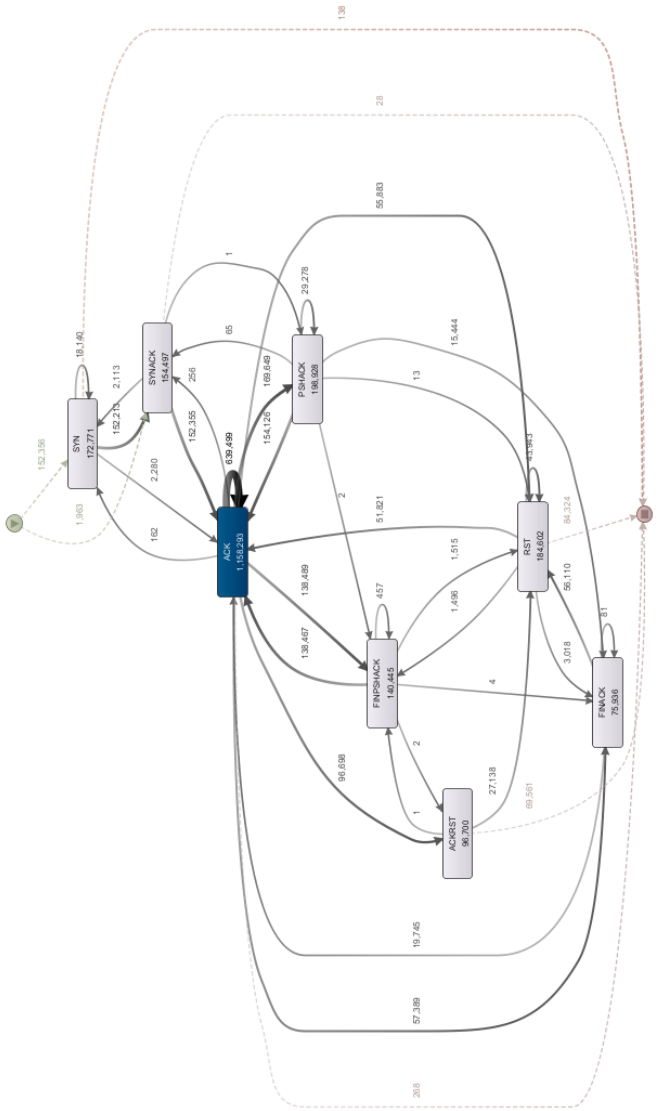


Figura 4.18: Direct Follower Graph DoS Hulk

Successivamente tramite il tool ProM, a partire dai file xes estratti con Disco, sono stati generate le reti di Petri di tutti gli attacchi. Per ottenere le reti è stato utilizzato un inductive miner con noise a zero. Rispetto ai Direct Follower Graph le reti di Petri consentono di visualizzare meglio le sequenzialità e i parallelismi delle tracce. In figura 4.19 è riportata la rete di Petri relativa all'attacco Bot.

In figura 4.20 è riportata la rete di Petri relativa all'attacco XSS.

In figura 4.21 è riportata la rete di Petri relativa all'attacco Dos Hulk.

Come è possibile notare gli attacchi hanno Direct Follower Graph e reti di Petri

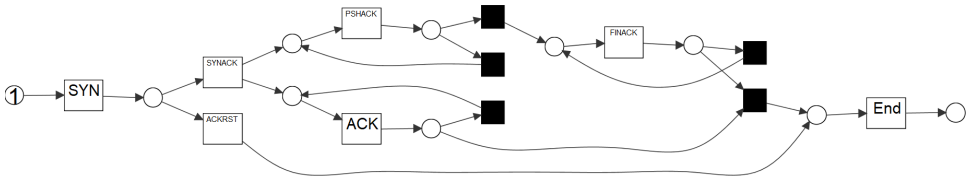


Figura 4.19: Petri Net Botnet

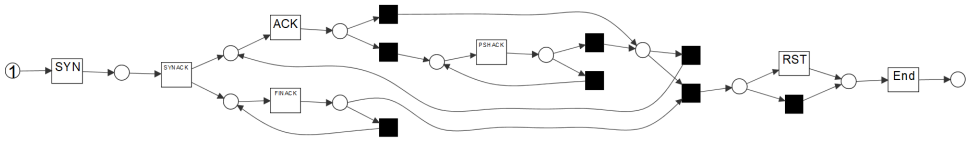


Figura 4.20: Petri Net XSS

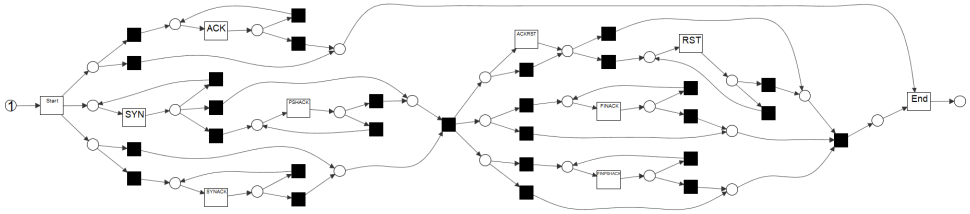


Figura 4.21: Petri Net DoS Hulk

differenti tra loro. Ovviamente ci sono alcune similitudini dato che i flag sono limitati, ma le sequenzialità ed i parallelismi sono differenti. Le reti di Petri mancanti sono riportate nella sezione A.3 dell'appendice.

Una piccola curiosità riguardo i flag: il flag URG non appare mai nel dataset ed i flag ECE e CWR, che sono legati alle congestioni di rete, appaiono solo nei flussi benigni. Ciò probabilmente è legato al fatto che i flussi benigni sono stati generati a partire da algoritmi di machine learning. Quindi il training probabilmente aveva anche dei flussi congestionati, mentre con gli attacchi non ci sono state congestioni.

4.4 Data Selection

In questo paragrafo sono illustrate le condizioni con cui è stata effettuata la selezione dei dati presenti nel flows dataset.

Ovviamente tutti i flussi con label UNKNOWN sono stati scartati. Sono stati esclusi anche tutti i flussi UDP perché sono stream di datagrammi che non conten-

gono flag, e come visto precedentemente tra i flussi maligni solo un flusso PortScan è di tipo UDP. Si è deciso di scartare anche tutti gli attacchi che sono poco rappresentati nel dataset, cioè tutti i flussi con label: Heartbleed, Web Attack – Sql Injection e Infiltration. Infatti come è possibile vedere dalla tabella 4.7 queste classi sono le meno rappresentate nel dataset. Infine, è stata analizzata la lunghezza dei flussi perché sarebbe computazionalmente troppo oneroso fornire sequenze di dati eccessivamente lunghe agli algoritmi di deep learning utilizzati, e sarebbe anche controproducente per le performance delle reti LSTM. Nella tabella 4.9 è riportata la lunghezza massima di un flusso per ogni classe di appartenenza.

Come è possibile notare la maggior parte delle classi non contengono flussi con una lunghezza massima troppo elevata, infatti solo alcuni flussi benigni hanno una lunghezza troppo elevata. Quindi si è deciso di considerare solo i flussi benigni TCP con lunghezza massima pari alla lunghezza massima di un flusso malevolo, cioè 321. Altrimenti il classificatore avrebbe un bias considerando tutti i flussi maggiori di 321 come flussi benigni.

Label	**Lunghezza massimo flusso**
Bot	125
PortScan	180
DDoS	25
Web Attack - Brute Force	311
Web Attack - XSS	321
FTP-Patator	30
SSH-Patator	78
DoS slowloris	44
DoS Slowhttptest	38
DoS Hulk	41
DoS GoldenEye	32
BENIGNI TCP	2466364

Tabella 4.9: Lunghezza massima dei flussi per classe

Infine, sono stati selezionati solo i flussi con lunghezza maggiore di due, perché i flussi con lunghezza due sono solo flussi delle seguenti tipologie:

- Richiesta di connessione rifiutata: ricevuto pacchetto con flag SYN e inviato pacchetto con flag RST
- Richiesta di connessione non risposta: inviati due pacchetti con flag SYN

Addestrare un classificatore a riconoscere un flusso formato da soli due pacchetti di questa tipologia non avrebbe senso.

In figura 4.22 è mostrato lo script che effettua le operazioni descritte.

Nello specifico il codice crea una nuova colonna *sum* calcolata come la somma delle colonne *Tot Fwd Packets* e *Tot Bwd Packets* e poi esegue le selezioni descritte. Inoltre, viene creato anche un csv unico contenente tutti i flussi.

Come si può notare dalla tabella 4.7 sono presenti più flussi benigni che flussi maligni nel dataset, la situazione migliora ma rimane simile anche dopo le operazioni appena descritte. Quindi per bilanciare il numero di flussi benevoli e malevoli

Figura 4.22: Data selection

si è scelto casualmente un numero di flussi benigni pari al numero dei flussi maligni, lasciando inalterata la distribuzione delle lunghezze dei flussi benigni. Il codice relativo è contenuto nel listato 4.1.

```
df_b = df[df.Label == "BENIGN"]
df_m = df[df.Label != "BENIGN"]

df_b['freq'] = df_b.groupby('sum')['sum'].transform('count')
df_b = df_b.sample(n=df_m.Label.count(), weights=df_b.freq)
```

Listato 4.1: Undersampling benigni

In particolare il dataframe costruito a partire dal dataset completo viene diviso in: un dataframe contenente solo i flussi benigni e un dataframe contenente solo i flussi maligni. Successivamente per il dataframe benigno viene calcolata la colonna frequenza a partire dalla lunghezza di ogni flusso e ne viene effettuato l'undersampling, selezionando casualmente un numero di elementi pari al numero di flussi maligni tenendo conto della distribuzione di frequenza calcolata.

Il flows dataset dopo le varie operazioni di data selection descritte ha la distribuzione di classi riportata in tabella 4.10.

Rispetto alla tabella di distribuzione precedente 4.7, solo gli attacchi Bot e PortScan hanno un numero di flussi inferiore, perché come visto nella sezione 4.3 contengono molti flussi con lunghezza pari a due. In particolare l'attacco PortScan passa dall'essere l'attacco più frequente ad essere il terzo attacco meno frequente, mentre l'attacco Bot conterrà solo circa 1/3 dei flussi di partenza.

4.5 Feature Engineering

In questo paragrafo è illustrato il processo di preprocessing eseguito sul grouped packets dataset. Come spiegato nel paragrafo 3.4, i classificatori utilizzano il grouped packets dataset formato dalle colonne riportate in tabella 4.1.

Label	Conteggio flussi
BENIGN	276.536
DoS Hulk	154.319
DDoS	95.683
DoS GoldenEye	7.495
DoS Slowhttptest	4.217
FTP-Patator	3.993
DoS slowloris	3.894
SSH-Patator	2.979
Web Attack – Brute Force	1.367
PortScan	1.174
Bot	736
Web Attack – XSS	679
Totale	**553.072**

Tabella 4.10: Distribuzione delle classi nel flows dataset dopo data selection

La prima operazione da svolgere è l'aggiunta ad ogni pacchetto della classe relativa al rispettivo flusso presente nel flows dataset. La seconda operazione da svolgere è eliminare alcune colonne non utili ai fini della classificazione. Infatti le colonne FrameNumber, acknumber e seqnumber sono state aggiunte per eseguire ricerche mirate tramite WireShark, al fine di verificare la corretta generazione dei flussi dopo le modifiche apportate al tool. E non sono utili ai fine della classificazione in quanto il FrameNumber è la posizione del pacchetto rispetto al file pcap. Invece acknumber e seqnumber sono entrambe funzione di un numero pseudo-casuale più la lunghezza del payload + 1, quindi sono informazioni ridondanti dato che la lunghezza del payload è un campo già presente. Le altre colonne da eliminare sono: flowId, ipSrc, ipDst, srcPort, dstPort, protocol perché la classificazione si deve basare solo sulle informazioni relative ai pacchetti e non ad informazioni aggiuntive relative alle macchine. Inoltre si aggiungerebbe un bias dato che quasi tutti gli attacchi sono stati effettuati dallo stesso indirizzo IP a causa del NAT. Anche la colonna payloadPacket è da eliminare perché fornisce un'informazione ridondante, ovvero se un pacchetto contiene o meno il payload, ma è già presente il campo payloadBytes che indica la dimensione del payload.

In figura 4.23 è riportata la porzione di codice, dello script in Python che svolge il preprocess effettuando le operazioni descritte.

Tutti i campi, tranne il timestamp, possono essere usati come feature senza dover effettuare trasformazioni.

Il timestamp non è utilizzato direttamente perché il dataset è simulato e inoltre si vuole classificare senza prendere in considerazione l'informazione temporale. Dal timestamp si è ricavata la feature *delta* che è stata calcolata come differenza in μs tra il timestamp del pacchetto ed il timestamp del pacchetto precedente.

Le feature estratte sono 13 in totale e sono riportate nella tabella 4.11, ovviamente nel dataset oltre alle feature è presente anche il campo Case ID che associa i pacchetti ai flussi.

Successivamente sono state analizzate le distribuzioni delle 4 feature numeriche: *delta*, *TCPWindow*, *headerBytes* e *payloadBytes* riportate in figura 4.24.

```
for file in os.listdir(path):
    df = pd.read_csv(path + file, header=0, index_col=0)
    day = file.split("-")[0]  # Prendo il giorno
    print(day)
    elem = ''.join([x for x in grouped if day in x])  # Trovo il file di quel giorno nel dataset grouped
    df1 = pd.read_csv(path_grouped + elem, header=0)

    df1["caseId"] = dict_map[day] + df1["caseId"].astype(str)

    for label in df["Label"].unique():
        print(label)
        df_labels = df[df["Label"] == label].index.tolist()
        df_grouped = df1.loc[df1["caseId"].isin(df_labels), df1.columns.drop(
            ["flowId", "ipSrc", "ipDst", "srcPort", "dstPort", "protocol", "payloadPacket", "FrameNumber", "seq number",
             "ack number"])]

        df_grouped["Label"] = label  # Aggiungo la Label nei grouped
        df_grouped_full = df_grouped_full.append(df_grouped)
```

Figura 4.23: Data Preprocess

Feature	Descrizione
headerBytes	lunghezza dell'header di livello 4 in byte
payloadBytes	lunghezza del payload di livello 4 in byte
TCPWindow	dimensione della finestra TCP
flagFIN	indica se il pacchetto contiene il flag FIN
flagPSH	indica se il pacchetto contiene il flag PSH
flagURG	indica se il pacchetto contiene il flag URG
flagECE	indica se il pacchetto contiene il flag ECE
flagSYN	indica se il pacchetto contiene il flag SYN
flagACK	indica se il pacchetto contiene il flag ACK
flagCWR	indica se il pacchetto contiene il flag CWR
flagRST	indica se il pacchetto contiene il flag RST
isForward	indica se il segmento è forward rispetto al flusso
delta	μs passati tra l'arrivo del pacchetto precedente e l'attuale

Tabella 4.11: feature estratte dal grouped packets dataset

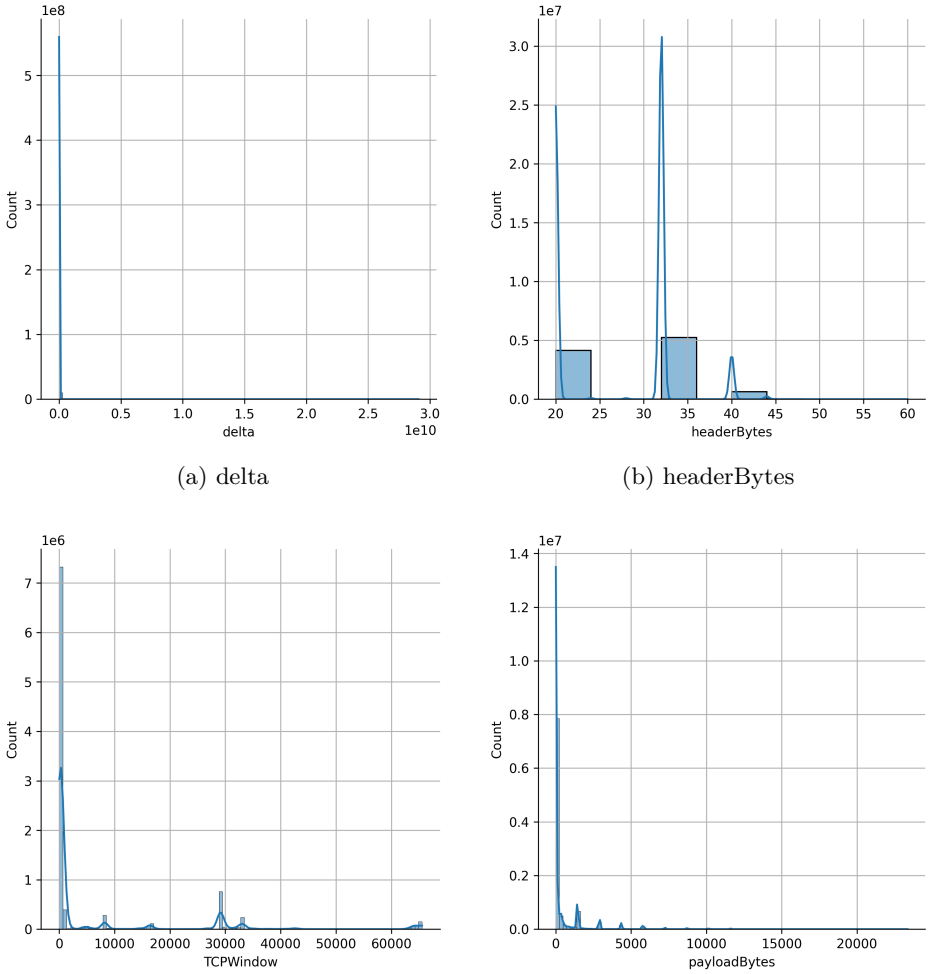
Come si può notare le distribuzioni non sono uniformi, tranne per la feature *headerBytes*, ma sono del tipo Power Law. Infatti le distribuzioni presentano un picco e tutti gli altri valori assunti sono molto sparsi nel grafico e poco frequenti.

In tabella 4.12 sono riportati media, moda, deviazione standard e massimo per ogni feature.

Feature	Avg	Mode	Std	Max
headerBytes	27.62	32	6.67	60.0
payloadBytes	448.03	0	1256.59	23360
TCPWindow	6966.19	229	14795.51	65535
delta	1864852.85	0	48176629.62	29070732447

Tabella 4.12: Statistiche feature

Come si può vedere la feature *delta* ha la distribuzione più sparsa rispetto alle



(a) delta

(b) headerBytes

(c) TCPWindow

(d) payloadBytes

Figura 4.24: Distribuzione feature

altre, infatti ha una deviazione standard elevatissima, una differenza tra il valore medio e la moda pari a quasi 2 milioni e la moda si trova nel valore zero, mentre il valore massimo è pari quasi a 30 miliardi. Ma anche le feature *TCPWindow* e *payloadBytes* hanno una distribuzione a power law molto sparsa, mentre *headerBytes* ha la distribuzione che più si avvicina alla distribuzione normale.

Per rendere le distribuzioni più uniformi, al fine di ottenere migliori performance nella classificazione, si è applicata una trasformazione tramite il logaritmo naturale. Dato che le feature *TCPWindow*, *payloadBytes* e *delta* assumono anche il valore zero, si è usata la funzione *log1p* della libreria numpy che prima di calcolare il logaritmo somma il valore uno. La feature *headerBytes* non è stata trasformata dato che non

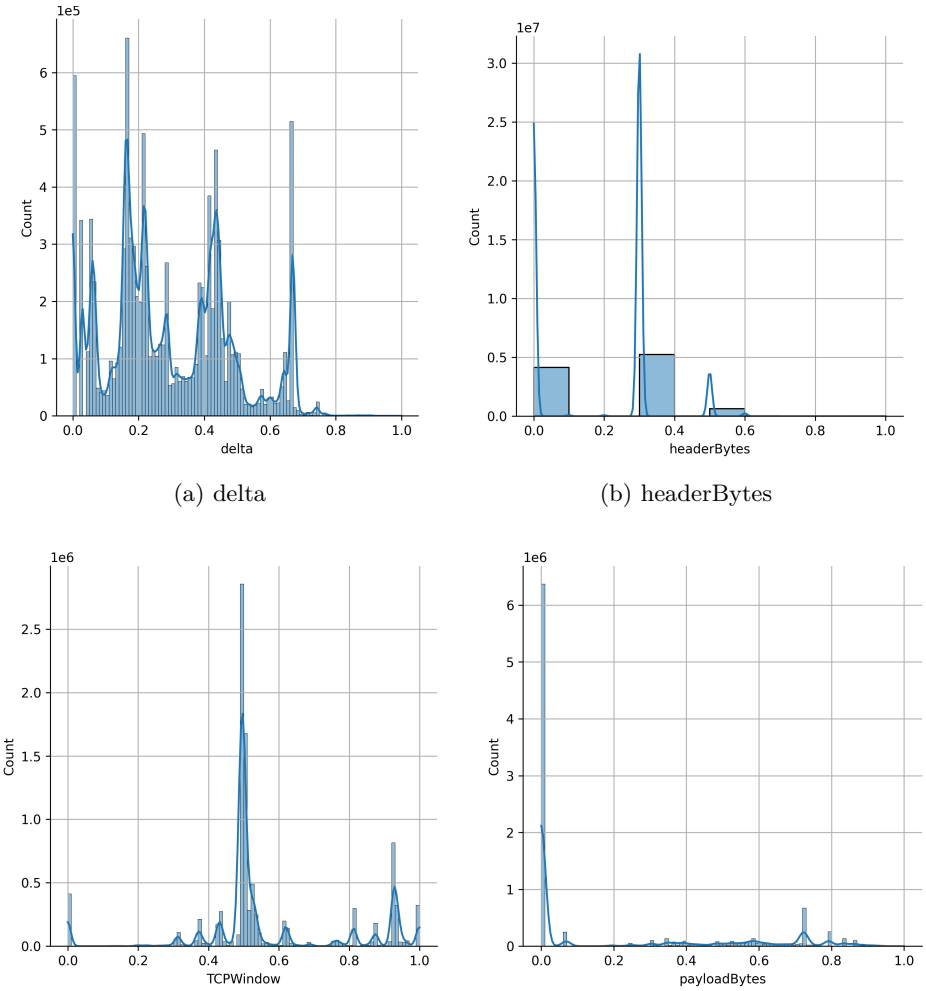
presenta una distribuzione power law. Infine, le colonne sono state normalizzate tra 0 e 1 tramite la classe MinMaxScaler di scikit-learn. In figura 4.25 è riportato il codice che effettua le operazioni descritte.

```
df_grouped_full = df_grouped_full.loc[:, df_grouped_full.columns.drop(
          ["flowId", "ipSrc", "ipDst", "srcPort", "dstPort", "protocol", "payloadPacket", "event"])]
# Calcolo la differenza in us del timestamp tra una riga e la successiva, quindi tra un pacchetto ed il successivo
timestamp = pd.to_datetime(df_grouped_full["timestamp"], format="%H:%M:%S.%f", errors="coerce")
timestamp_s = pd.to_datetime(df_grouped_full["timestamp"], format="%H:%M:%S", errors="coerce")
df_grouped_full['timestamp'] = timestamp.combine_first(timestamp_s)

df_grouped_full["delta"] = df_grouped_full["timestamp"].diff().astype('timedelta64[us]')
# Imposto a zero il delta del primo pacchetto di ogni flusso
df_grouped_full.loc[df_grouped_full.drop_duplicates("caseId").index.tolist(), "delta"] = 0
#df_grouped_full["delta"] = df_grouped_full["delta"].astype(np.uint)

df_grouped_full.drop(columns="timestamp", inplace=True)

min_max_scaler = MinMaxScaler()

df_grouped_full["delta"] = np.log1p(df_grouped_full["delta"])
df_grouped_full["payloadBytes"] = np.log1p(df_grouped_full["payloadBytes"])
df_grouped_full["TCPWindow"] = np.log1p(df_grouped_full["TCPWindow"])

df_grouped_full[["delta", "headerBytes", "payloadBytes", "TCPWindow"]] = min_max_scaler.fit_transform(
    df_grouped_full[["delta", "headerBytes", "payloadBytes", "TCPWindow"]])
```

Figura 4.25: Feature Selection

Le distribuzioni ottenute sono riportate in figura 4.26.

Sono state provate altre trasformazioni più tradizionali ma non hanno portato dei miglioramenti significativi. Solo la famiglia di trasformazioni boxcox ha prodotto delle distribuzioni simili a quelle ottenute con la trasformazione logaritmica, ma sono state ritenute migliori queste ultime. Come si può notare dalla figura 4.26 la distribuzione relativa alla feature *delta* ora è molto più vicina ad una distribuzione gaussiana rispetto a prima, anche le feature *TCPWindow* e *payloadBytes* hanno delle distribuzioni più vicine a quella normale. In tabella 4.13 è riportato il numero di pacchetti presenti nel grouped packets dataset per classe. Anche se il dataset è bilanciato, avendo lo stesso numero di flussi benigni e maligni, il numero di pacchetti è leggermente sbilanciato verso i benigni. Infatti i pacchetti benigni sono circa il 60 % del dataset contro il circa 40 % dei maligni. Perché come visto nella sezione 4.3 i flussi benigni hanno in media una lunghezza maggiore rispetto ai flussi maligni.



(a) delta

(b) headerBytes

(c) TCPWindow

(d) payloadBytes

Figura 4.26: Distribuzione feature

Label	**Packets count**
BENIGN	6032535
DoS Hulk	2182174
DDoS	1280602
SSH-Patator	163320
FTP-Patator	111618
DoS GoldenEye	105997
DoS Slowhttptest	80846
DoS slowloris	71531
Web Attack - Brute Force	30267
Bot	9871
Web Attack - XSS	9638
PortScan	5584
Total	**10083983**

Tabella 4.13: Numero di pacchetti per label

5

Esperimenti

In questo capitolo prima viene descritto il modo in cui è stato suddiviso in fold il dataset. Successivamente sono descritte le architetture di deep learning utilizzate ovvero: le LSTM e i Transformer, con una particolare attenzione alla spiegazione del preprocess da effettuare sul dataset prima di poter essere utilizzato come input per le reti. Infine è presente la spiegazione di come è stato implementato il calcolo della metrica earliness.

5.1 Suddivisione del dataset

Gli esperimenti inizialmente sono stati eseguiti utilizzando Google Colab e Kaggle, successivamente sono stati eseguiti su una macchina del dipartimento con una gpu Nvidia V100. Per quanto riguarda l'ambiente di lavoro si è utilizzato Jupyter notebook con Python 3.8 ed in particolare sono state utilizzate le librerie Tensorflow 2.3, Pandas e Scikit-learn. La sezione di codice relativa al preprocess del dataset è comune in entrambe le reti. É stato necessario innanzitutto trasformare le etichette da stringhe a categorie numeriche, come mostrato nella funzione nel listato 5.1.

```
1 def preprocess(df_grouped, df_flow):
2     le = LabelEncoder()
3     df_flow["categorical_label"] = le.fit_transform(df_flow.Label)
4     df_grouped["categorical_label"] = le.transform(df_grouped.Label)
5     df_flow.drop("Label", axis=1, inplace=True)
6     df_grouped.drop("Label", axis=1, inplace=True)
7     return df_flow, df_grouped
```

Listato 5.1: Preprocess

Successivamente il dataset è stato suddiviso in 5 fold utilizzando la funzione split riportata nel listato 5.2, che sfrutta alcune funzioni della libreria Scikit-learn. La suddivisione del dataset in fold è stratificata, ovvero ogni fold ha la stessa distribuzione delle classi. In particolare la stratificazione è stata eseguita sulle classi dei flussi, in questo modo tutti i pacchetti relativi ad un determinato flusso sono contenuti solamente in una singola fold. Il set di validazione invece è stato ottenuto prendendo casualmente il 15% di flussi a partire dal set di addestramento, replicandone la distribuzione delle classi. Infine, i set di train, validation e test vengono

salvati in file csv dove i file che iniziano con x contengono le feature dei pacchetti, mentre i file che hanno y come iniziale contengono le relative classi.

```
def split(df_flow, df_grouped, nfolds):
    # Divide il  dataset in training (+ holdout) e test con un k fold stratificato
    sss = StratifiedKFold(n_splits=nfolds, shuffle=True, random_state=42)
    df_flow = df_flow.reset_index()
    x = df_flow.iloc[:, :-1].to_numpy()  # Get features
    y = df_flow.iloc[:, -1].to_numpy()  # Get categorical_label

    fold = 0
    for train, test in sss.split(x, y):
        print("Writing fold " + str(fold))
        x_train, x_test, y_train, y_test = x[train], x[test], y[train], y[test]
        # Suddivido il train set in train set e validation set stratificando in base alle classi
        x_train, x_validation = train_test_split(x_train, stratify=y_train, test_size=0.15)

        df_train = pd.DataFrame(x_train, columns=df_flow.columns[:-1])
        df_validation = pd.DataFrame(x_validation, columns=df_flow.columns[:-1])
        df_test = pd.DataFrame(x_test, columns=df_flow.columns[:-1])

        df_grouped.loc[df_grouped["caseId"].isin(df_train["caseId"]), df_grouped.columns != "categorical_label"].to_csv(
            "/content/drive/MyDrive/Tesi/Dataset/CrossValidation/x_train" + str(fold) + ".csv", index=False)
        df_grouped.loc[df_grouped["caseId"].isin(df_train["caseId"]), ["caseId", "categorical_label"]].to_csv(
            "/content/drive/MyDrive/Tesi/Dataset/CrossValidation/y_train" + str(fold) + ".csv", index=False)
        df_grouped.loc[df_grouped["caseId"].isin(df_validation["caseId"]), df_grouped.columns != "categorical_label"].to_csv(
            "/content/drive/MyDrive/Tesi/Dataset/CrossValidation/x_validation" + str(fold) + ".csv", index=False)
        df_grouped.loc[df_grouped["caseId"].isin(df_validation["caseId"]), ["caseId", "categorical_label"]].to_csv(
            "/content/drive/MyDrive/Tesi/Dataset/CrossValidation/y_validation" + str(fold) + ".csv", index=False)
        df_grouped.loc[df_grouped["caseId"].isin(df_test["caseId"]), df_grouped.columns != "categorical_label"].to_csv(
            "/content/drive/MyDrive/Tesi/Dataset/CrossValidation/x_test" + str(fold) + ".csv", index=False)
        df_grouped.loc[df_grouped["caseId"].isin(df_test["caseId"]), ["caseId", "categorical_label"]].to_csv(
            "/content/drive/MyDrive/Tesi/Dataset/CrossValidation/y_test" + str(fold) + ".csv", index=False)
        fold += 1
    print("Files created")
```

Listato 5.2: Stratified K-Fold Split

5.2 Esperimenti con la rete LSTM

Le reti LSTM(Long Short-Term Memory) e più in generale le reti ricorrenti possono gestire input sequenziale di lunghezza fissa, classificando in base all'intera sequenza di elementi e non rispetto ad un singolo elemento come nelle altre tipologie di classificatori. Ogni sample è un array bidimensionale (M x N) con N colonne pari al numero di feature e M righe pari alla lunghezza della sequenza che si sceglie di

utilizzare, chiamata *window size* (WS). Il dataset è stato adattato a questa tipologia di funzionamento tramite la funzione create_windows riportata nel listato 5.3. Concettualmente, per ogni gruppo di pacchetti relativo ad un determinato flusso, si fa scorrere una finestra di lunghezza fissa ottenendo così l'input correttamente dimensionato per la rete. Il codice raggruppa tutti i pacchetti in base al relativo Case ID, creando un array dove in ogni posizione è contenuto un array che include le feature di tutti i pacchetti di un determinato flusso. Per ogni raggruppamento viene quindi controllato se il numero di pacchetti è almeno pari alla dimensione della window size, altrimenti il raggruppamento viene escluso dalla classificazione perché non è abbastanza grande. Per ogni raggruppamento che invece soddisfi la condizione, vengono selezionati sequenzialmente WS pacchetti facendo scorrere la finestra dal primo pacchetto all'ultimo. In questo modo si genera un tensore dove in ogni riga è contenuto un sample, cioè un array bidimensionale (M x N). Il vettore delle etichette viene invece costruito prendendo la classe relativa al flusso per ogni sample generato.

```
def create_windows(x_dataset, y_dataset, window_size, num_feature):
    dataset = pd.concat([x_dataset, y_dataset["categorical_label"]], axis=1)
    num_row = dataset.shape[0]
    dataset = [group[1].to_numpy() for group in dataset.groupby("caseId")]
    x_data, y_data = np.zeros((num_row, window_size, num_feature), dtype=np.
     float16), np.zeros((num_row), dtype=np.int8)
    j = 0

    for case, group in enumerate(dataset):
        num_line = group.shape[0]

        if num_line >= window_size:
            for i in range((num_line - window_size) + 1):
                x_i = group[i: i + window_size, 1:-1]
                x_data[j, :, :] = x_i
                y_data[j] = group[0, -1]
                j += 1

    x_data = x_data[:j, :, :]
    y_data = y_data[:j]

    return x_data, y_data
```

Listato 5.3: Create windows for LSTM

La funzione create_windows viene richiamata su: train set, validation set e test set di ogni fold e ogni file generato viene salvato con estensione .npy per permetterne il caricamento tramite numpy. Per la classificazione multi-label il procedimento è analogo, ma prima di eseguire la funzione create_windows vengono selezionati dal dataset solo i pacchetti relativi ai flussi malevoli. Successivamente il train set risultante viene mescolato prima della fase di addestramento. Perché dopo la suddivisione in finestre c'è il rischio di avere batch altamente sbilanciate e al limite contenenti solo feature di una stessa classe.

Come visto nella sezione 4.3 le distribuzioni delle lunghezze dei flussi variano in base alla classe di appartenenza. In particolare i flussi maligni in media hanno una lunghezza inferiore rispetto ai benigni. Quindi all'aumentare della window size aumenta lo sbilanciamento tra la classe maligna e la classe benigna. Per ovviare a questa problematica si sono selezionati casualmente un numero di sample benigni

pari al 110% del numero di sample maligni, perché il dataset è composto dal 60% di pacchetti benigni e dal 40% di pacchetti maligni.

La funzione build_LSTM, riportata nel listato 5.4, genera il modello della rete neurale sia per la classificazione binaria che per la classificazione multi-label.

```
def build_LSTM(ws, units, layers_number, dropout, features, num_labels=1):
    # Initialising the RNN
    model = Sequential(name="model")
    for layer in range(layers_number):
        if layer == 0 and layers_number == 1:
            model.add(LSTM(units=units, input_shape=(ws, features),
     return_sequences=False, dropout=dropout))
        elif layers_number - layer > 1:
            model.add(LSTM(units=units, input_shape=(ws, features),
     return_sequences=True, dropout=dropout))
        else:
            model.add(LSTM(units, dropout=dropout))

    # Output dense layer
    if num_labels == 1:
        model.add(Dense(units=num_labels, activation="sigmoid"))
        model.compile(loss='binary_crossentropy', optimizer='nadam', metrics=['
     accuracy', Precision(), Recall()])
    else:
        model.add(Dense(units=num_labels, activation="softmax"))
        model.compile(loss='categorical_crossentropy', optimizer='nadam', metrics
     =['accuracy', Precision(), Recall()])

    return model
```

Listato 5.4: Build LSTM Network

Nello specifico la funzione genera un modello contenente un numero di layer LSTM pari al parametro *layers_number*, ognuno con un numero di neuroni pari al parametro *units* e con *dropout* pari al relativo parametro. Il parametro *return_sequences* va impostato a True nei layer LSTM a monte di un altro layer LSTM, in modo tale che il layer a valle riceva tutti gli hidden state ad ogni time-step dal layer a valle. I parametri *ws* e *feature* servono invece per specificare la dimensione dell'input. Il numero di unità dello strato finale fully connected dipende dal parametro *num_labels*, per la classificazione binaria è pari ad uno mentre per la classificazione multi-label è pari al numero di classi. Anche la funzione di attivazione del layer fully connected varia in base al tipo di classificazione: è una sigmoide per la classificazione binaria oppure è una softmax per per la classificazione multi-label.

Viene prima eseguito l'addestramento, poi selezionando il modello migliore tra le varie epoche, viene eseguito il testing con il classificatore binario. Successivamente viene eseguito il training sul classificatore multi-classe e selezionando il modello migliore tra le varie epoche viene effettuato il testing, passando le predizioni che il classificatore binario a monte ha classificato come malevole. Al fine di trovare il modello che produca i risultati migliori si è eseguita una grid search facendo variare gli iperparametri come riportato in tabella 5.1.

In totale le combinazioni sono pari a 160, ma dato che ogni configurazione viene ripetuta per 5 fold sono stati eseguiti 800 addestramenti e test in totale. Come ottimizzatore si è usato Nadam e si è usata una dimensione di batch pari a 512. Il numero di epoche è stato scelto pari a 30, ma si è utilizzato l'early-stopping per terminare in anticipo un addestramento se la loss di validazione non decresce entro tre epoche. Per quanto riguarda la window size pari a 20 non è stato possibile eseguire

Iperparametro	Valori
window size	3, 5, 10, 20
unità LSTM	20, 50, 100, 200, 500
layer	1, 2
dropout	0, 0.2, 0.4, 0.6

Tabella 5.1: Configurazioni LSTM

la classificazione multi-label su tutte la classi dato che la classe PortScan contiene pochissimi flussi che hanno una dimensione maggiore di 20. Quindi gli esperimenti effettuati con tale window size non contengono anche i flussi dei PortScan per questo motivo, ovviamente il numero di unità del layer di output è stato modificato di conseguenza.

5.3 Esperimenti con Transfomer

Come visto nella sezione 2.2, anche i Transformer possono essere utilizzati per classificare sequenze di dati. Ma a differenza delle reti LSTM possono accettare come input anche lunghe sequenze, contenenti molti zeri. Questo permette di poter utilizzare una finestra incrementale, ovvero una finestra che cresce con l'arrivo di nuovi elementi che appartengono alla sequenza. Quindi a differenza delle reti LSTM non è presente la necessità di scartare i flussi che non raggiungono la dimensione WS, tutti i flussi saranno classificati a prescindere dalla dimensione. In un contesto di streaming questo si traduce nella capacità del Transformer di classificare un nuovo flusso appena arriva un pacchetto, a differenza della LSTM con la quale si deve attendere che il flusso raggiunga la dimensione WS minima.

La funzione create_windows, riportata nel listato 5.5, è stata creata a partire dalla versione LSTM adattandola a questo scenario. La parte iniziale delle funzione è analoga al caso precedente, infatti vengono raggruppati i pacchetti in base al Case ID corrispondente e per ogni flusso viene controllato se il numero di pacchetti associato non sia maggiore rispetto alla window size. In caso affermativo vengono create N finestre, con N pari al numero di pacchetti di un flusso, ognuna con dimensione incrementale rispetto alla precedente. Quindi la prima finestra contiene solo le feature relative al primo pacchetto, mentre l'ultima contiene le feature di tutti i pacchetti di un flusso. Ognuna delle N - 2 finestre intermedie contiene le feature della finestra precedente e in aggiunta le feature del pacchetto successivo. Se invece il numero di pacchetti dovesse essere maggiore rispetto alla window size il procedimento è analogo, ma si considerano solo gli ultimi WS pacchetti del flusso. In tal caso le finestre non partono dal primo pacchetto bensì dal pacchetto che si trova in posizione N - WS, con N pari al numero di pacchetti del flusso.

La finestre hanno una dimensione variabile, per portare tutte le finestre alla dimensione WS viene effettuato un pre-zero padding aggiungendo sequenze di zeri prima delle feature. Nel codice questo passaggio è implicito perché viene istanziato un tensore di zeri con le dimensioni opportune che viene popolato progressivamente.

L'ultima differenza rispetto alla versione per la rete LSTM è la presenza di una matrice aggiuntiva: *z_data*. Tale matrice viene utilizzata per il calcolo della metrica

earliness, ha la stesso numero di righe degli altri tensori e ogni riga contiene il numero del flusso a cui è relativo il pacchetto, la dimensione effettiva della finestra e la dimensione totale del flusso.

```
def create_windows(x_dataset, y_dataset, window_size, num_feature):
    dataset = pd.concat([x_dataset, y_dataset["categorical_label"]], axis=1)
    num_row = dataset.shape[0]
    dataset = [group[1].to_numpy() for group in dataset.groupby("caseId")]

    x_data, y_data = np.zeros((num_row, window_size, num_feature), dtype=np.
     float16), np.zeros((num_row), dtype=np.int8)
    z_data = np.zeros((num_row, 3), dtype=np.uint)
    j = 0

    for case, group in enumerate(dataset):
        num_line = group.shape[0]

        if num_line <= window_size:
            for i in range(num_line):
                x_i = group[0: i + 1, 1:-1]
                x_data[j, window_size - x_i.shape[0]:  window_size, :] = x_i
                y_data[j] = group[0, -1]
                z_data[j] = np.array([case, x_i.shape[0], num_line])
                j += 1
        else:
            offset = num_line - window_size
            for i in range(window_size):
                x_i = group[offset: offset + i + 1, 1:-1]
                x_data[j, window_size - x_i.shape[0]:  window_size, :] = x_i
                y_data[j] = group[0, -1]
                z_data[j] = np.array([case, x_i.shape[0], num_line])
                j += 1

    x_data = x_data[:j, :, :]
    y_data = y_data[:j]
    z_data = z_data[:j]

    return x_data, y_data, z_data
```

Listato 5.5: Create windows for Transformer

5.3.1 Positional Encoding

Come spiegato nella sezione 2.2 i positional encoding servono per aggiungere delle informazioni rispetto alla posizione relativa di ogni feature nel sample di dimensione (M x N). Infatti i Transformer non avendo una struttura ricorrente non hanno informazioni sulle posizioni relative nelle sequenze, per questo vengono aggiunti i positional encoding. Le formule utilizzate in questo modello sono quelle standard 2.1.

Nel listato 5.6 è riportato il codice che calcola i positional encoding, il codice è stato adattato a partire dal codice di Tensorflow [30].

```
def get_angles(pos, i, d_model):
    angle_rates = 1 / np.power(10000, (2 * (i // 2)) / np.float16(d_model))
    return pos * angle_rates


def positional_encoding(position, d_model):
    angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                            np.arange(d_model)[np.newaxis, :],
                            d_model)
```

```
10
11      # apply sin to even indices in the array; 2i
12      angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
13
14      # apply cos to odd indices in the array; 2i+1
15      angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
16
17      pos_encoding = angle_rads[np.newaxis, ...]
18
19      return tf.cast(pos_encoding, dtype=tf.float16)
20
21
22  class Positional_enc(layers.Layer):
23      def __init__(self, maximum_position_encoding, d_model):
24          super(Positional_enc, self).__init__()
25          self.maximum_position_encoding = maximum_position_encoding
26          self.d_model = d_model
27          self.pos_encoding = positional_encoding(self.maximum_position_encoding,
          self.d_model)
28
29      def call(self, inputs):
30          inputs += self.pos_encoding[:, :, :]
31          return inputs
32
33      def get_config(self):
34          config = super().get_config().copy()
35          config.update({
36              'maximum_position_encoding': self.maximum_position_encoding,
37              'd_model': self.d_model,
38          })
39          return config
```

Listato 5.6: Positional Encoding

La funzione get_angles calcola l'argomento delle funzioni seno e coseno. Mentre la funzione positional_encoding calcola gli encoding applicando la funzione seno sugli indici pari e la funzione coseno sugli indici dispari dell'output della funzione get_angles. Utilizzando come valore per la variabile *pos* la window size e come valore per la variabile *d_model* il numero di feature si ottiene un encoding della stessa dimensione di un sample. Quindi la classe Positional_enc richiama la funzione positional_encoding e ne somma il risultato all'input della rete neurale.

5.3.2 Transformer Model

Nel listato 5.7 è riportato il codice relativo alla Multi-Head Self Attention. Il codice per implementare l'architettura Transformer è stato adattato a partire dal repository [31] che crea un transformer per risolvere problemi di computer vision.

```
1   class MultiHeadSelfAttention(layers.Layer):
2       def __init__(self, embed_dim, num_heads=8):
3           super(MultiHeadSelfAttention, self).__init__()
4           self.embed_dim = embed_dim
5           self.num_heads = num_heads
6           if embed_dim % num_heads != 0:
7               raise ValueError(
8                   f"embedding dimension = {embed_dim} should be divisible by number
        of heads = {num_heads}"
9               )
10          self.projection_dim = embed_dim // num_heads
11          self.query_dense = layers.Dense(embed_dim)
12          self.key_dense = layers.Dense(embed_dim)
13          self.value_dense = layers.Dense(embed_dim)
```

```
        self.combine_heads = layers.Dense(embed_dim)

    def attention(self, query, key, value):
        score = tf.matmul(query, key, transpose_b=True)
        dim_key = tf.cast(tf.shape(key)[-1], tf.float16)  # tf.float16 era
        scaled_score = score / tf.cast(tf.math.sqrt(dim_key), tf.float16)  # ***
        weights = tf.nn.softmax(scaled_score, axis=-1)
        output = tf.matmul(weights, value)
        return output, weights

    def separate_heads(self, x, batch_size):
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.projection_dim))
        return tf.transpose(x, perm=[0, 2, 1, 3])

    def call(self, inputs):
        # x.shape = [batch_size, seq_len, embedding_dim]
        batch_size = tf.shape(inputs)[0]
        query = self.query_dense(inputs)  # (batch_size, seq_len, embed_dim)
        key = self.key_dense(inputs)  # (batch_size, seq_len, embed_dim)
        value = self.value_dense(inputs)  # (batch_size, seq_len, embed_dim)
        query = self.separate_heads(
            query, batch_size
        )  # (batch_size, num_heads, seq_len, projection_dim)
        key = self.separate_heads(
            key, batch_size
        )  # (batch_size, num_heads, seq_len, projection_dim)
        value = self.separate_heads(
            value, batch_size
        )  # (batch_size, num_heads, seq_len, projection_dim)
        attention, weights = self.attention(query, key, value)
        attention = tf.transpose(
            attention, perm=[0, 2, 1, 3]
        )  # (batch_size, seq_len, num_heads, projection_dim)
        concat_attention = tf.reshape(
            attention, (batch_size, -1, self.embed_dim)
        )  # (batch_size, seq_len, embed_dim)
        output = self.combine_heads(
            concat_attention
        )  # (batch_size, seq_len, embed_dim)
        return output
```

Listato 5.7: Multi-Head Self Attention

Come si può notare per creare un oggetto MultiHeadSelfAttention bisogna passare come parametri la dimensione dell'embedding e il numero di heads che si vuol usare. Inoltre le matrici query, key e value sono uguali, ovvero l'input della rete, per questo motivo l'attenzione è detta self.

Nel listato 5.6 è riportato il codice relativo all'Encoder del Transformer.

```
class TransformerBlock(layers.Layer):
    def __init__(self, embed_dim, num_heads, ff_dim, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.att = MultiHeadSelfAttention(embed_dim, num_heads)
        self.ffn = tf.keras.Sequential(
            [layers.Dense(ff_dim, activation="relu"), layers.Dense(embed_dim), ]
        )
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(rate)
        self.dropout2 = layers.Dropout(rate)

    def call(self, inputs, training):
        attn_output = self.att(inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
```

```
19          return self.layernorm2(out1 + ffn_output)
20
21      def get_config(self):
22          config = super().get_config().copy()
23          config.update({
24              'embed_dim': embed_dim,
25              'num_heads': num_heads,
26              'ff_dim': ff_dim,
27          })
28          return config
```

Listato 5.8: Transformer Encoder

Come visto nella sezione 2.2 l'encoder è formato da un multi head self attention seguito da un layer di normalizzazione seguito da uno strato denso ed infine è presente un ulteriore layer di normalizzazione. Nel codice sono stati aggiunti anche due layer di dropout prima dei layer di normalizzazione.

Nel listato 5.9 è riportato il codice che genera il modello Transformer binario e multi-classe.

```
1  def build_transformer(pos_enc, transformer_block, ws, num_features, num_labels=1)
       :
2      inputs = layers.Input(shape=([ws, num_features]))
3      x = pos_enc(inputs)
4      x = tf.keras.layers.Flatten()(inputs)
5      x = transformer_block(x)
6      x = layers.GlobalAveragePooling1D()(x)
7      x = layers.Dropout(0.1)(x)
8      x = layers.Dense(20, activation="relu")(x)
9      x = layers.Dropout(0.1)(x)
10
11     if num_labels == 1:  # binary
12         outputs = layers.Dense(num_labels, activation="sigmoid")(x)
13         model = Model(inputs=inputs, outputs=outputs)
14         model.compile(loss='binary_crossentropy', metrics=['accuracy', AUC(),
        Precision(), Recall()], optimizer='nadam')
15     else:  # categorical
16         outputs = layers.Dense(num_labels, activation="softmax")(x)
17         model = Model(inputs=inputs, outputs=outputs)  # Provare con sparse
18         model.compile(loss='categorical_crossentropy', metrics=['accuracy', AUC()
         , Precision(), Recall()], optimizer='nadam')
19
20     return model
```

Listato 5.9: Build Transformer

Come si può notare il primo layer della rete è un layer di input che accetta sample con dimensione pari a (M x N), successivamente l'input viene passato al layer che calcola il positional encoding e restituisce come output l'input sommato ai positional encoding. Dato che i Transformer lavorano con embedding il risultato del layer positional_encoding viene ridimensionato come array unidimensionale e utilizzato come input dall'encoder del Transformer. In questo modo l'input della rete non subisce modifiche dovute all'embedding ma cambia solo dimensione per essere utilizzato dall'encoder. Il risultato dell'encoder viene poi inviato a un layer che effettua una global average pooling, infine sono presenti: un layer di dropout, un layer denso con relu come funzione di attivazione e un ultimo layer di dropout. Dato che si vuole solo classificare e non ottenere delle squenze di pacchetti come output non c'è bisogno del decoder. Quindi come layer finale viene inserito un layer denso con un neurone e funzione di attivazione sigmoide nel caso di classificazione

binaria, oppure un layer denso con un numero di neuroni pari al numero di classi e funzione di attivazione softmax in caso di classificazione multi-label.

5.3.3 Earliness

La earliness [29] è una metrica che indica in media "quanti elementi di una sequenza bastano al classificatore per dare una classificazione corretta". Dato che ogni sample contiene gli eventi, ovvero le feature dei pacchetti, del sample precedente e anche l'evento successivo questa metrica permette di sapere in quale sample il risultato del classificatore coincide con l'etichetta corretta. Viene calcolata come:

$$\frac{len(s)}{len(t)} \tag{5.1}$$

dove len(s) indica la lunghezza minore del sample che è stato predetto correttamente rispetto ad una determinata traccia e len(t) indica la lunghezza totale della traccia. In questo specifico caso quindi la lunghezza del sample è il numero di pacchetti in una certa finestra, mentre la lunghezza della traccia è la dimensione del flusso, cioè il numero totale di pacchetti in un flusso. Ovviamente minore è il valore di questa metrica e più il classificatore riesce a predire correttamente l'esito di una traccia utilizzando meno eventi della traccia stessa.

Come detto precedentemente per il calcolo di questa metrica viene utilizzato il vettore z, generato durante la creazione delle finestre. Infatti la funzione compute_earliness calcola la earliness a partire dal vettore z, dal vettore contenente i label predetti e dal vettore contenente i label corretti.

Ogni i-esimo elemento del vettore z corrisponde alle feature del i-esimo sample nel vettore x e alla classe i-esima del vettore y. Come visto in precedenza, il vettore z contiene per ogni elemento un Case ID numerico, la lunghezza del sample e la lunghezza totale della traccia relativa. Le righe del vettore z vengono raggruppate in base al Case ID e ogni gruppo viene ordinato in ordine crescente rispetto alla colonna sample_len, ovvero il numero di pacchetti che contiene una certa finestra. Successivamente viene istanziato un vettore chiamato earliness con un numero di righe pari al numero di gruppi nel dataset. Infine, per ogni sample di ogni gruppo viene confrontato se la classe predetta è uguale alla classe reale e in caso positivo viene aggiunto nell'array earliness il valore della lunghezza del sample diviso il numero totale di pacchetti del flusso. Una volta generato il vettore earliness viene calcolata la media e la deviazione standard. Tutti gli eventuali sample che non dovessero mai venir classificati correttamente non vengono considerati nel calcolo della earliness, dato che è una metrica relativa solo ai sample che vengono classificati correttamente.

```
1 def compute_earliness(z, y_pred, y_true):
2     df_z = pd.DataFrame(z, columns=["caseid", "sample_len", "flow_len"])
3     df_z = df_z.reset_index()
4     group = [group[1].sort_values(["sample_len"]) for group in df_z.groupby("
        caseid")]
5     earliness = np.zeros(len(group), dtype=np.float16)
6     i = 0
7
8     for sample in group:
9         for x in sample.itertuples():
```

```
            if y_pred[x.index] == y_true[x.index]:
                earliness[i] = x.sample_len/x.flow_len
                i += 1
                break

    return earliness
```

Listato 5.10: Compute Earliness

Per quanto riguarda le configurazioni testate, si è utilizzata la configurazione di iperparametri riportata in tabella 5.2. Non sono state scelte tante configurazioni come per le reti LSTM per motivi di tempo, dato che il numero di parametri da allenare con i Transformer è molto maggiore utilizzando una finestra più grande rispetto alle reti LSTM.

Iperparametro	**Valore**
Window size	100
Batch size	64
Unità FF	128
Heads	26

Tabella 5.2: Configurazione Transformer

In particolare si è utilizzato un solo layer di encoder con un numero di teste della multi-head attention pari a 2 volte il numero di feature e 128 unità dense nel sub-layer di feed forward. Come dimensione di batch è stato scelto il valore 64 perché è il valore, che sulla macchina dove si sono svolti i test, ha permesso di utilizzare tutta la memoria RAM in modo ottimale. Infatti con le LSTM dove la WS massima è pari a 20 si è potuta scegliere una batch pari a 512, ma non con i Transformer dove la window size scelta è pari a 100. Poteva essere scelta una window size pari a 321, ovvero la lunghezza massima di un flusso nel dataset, ma per motivi di tempistiche è stato scelto il valore 100 che comunque include pienamente la maggior parte dei flussi. Come ottimizzatore si è usato Nadam, mentre il numero massimo di epoche è stato posto pari a 25 sempre usando l'early-stopping.

La parte finale in cui vengono calcolate le metriche accuracy, precision, recall e la matrice di confusione sia nella classificazione binaria che nella classificazione multi-classe è comune in entrambe le architetture. Mentre il calcolo della metrica earliness avviene solo per l'architettura Transformer.

6

Risultati

In questa sezione vengono riportati i risultati ottenuti con le reti deep learning utilizzate, sia per la classificazione binaria che per la classificazione multi-label.

6.1 Risultati LSTM

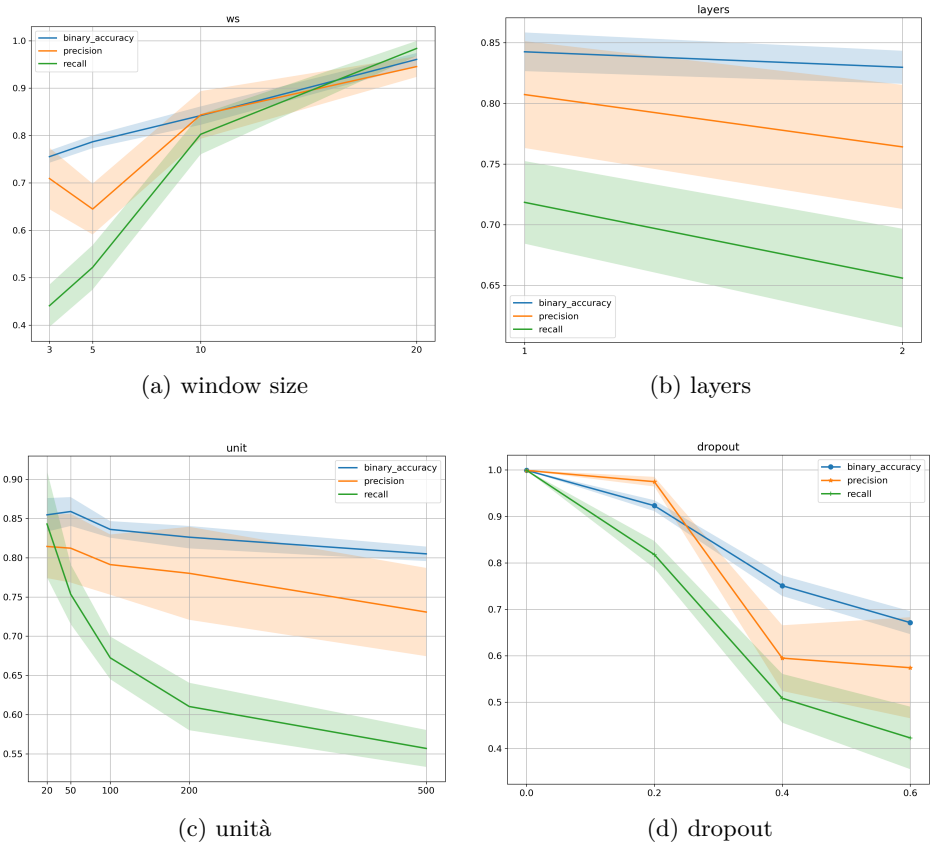
6.1.1 Classificazione binaria

I risultati ottenuti dalle varie configurazioni delle reti LSTM binarie provate sono riportati di seguito. Per ogni configurazione eseguita i risultati sono stati mediati tra le varie fold calcolando anche la deviazione standard. Essendo 160 le configurazioni totali i risultati sono stati nuovamente aggregati, rispetto ad ogni iperparametro riportato in tabella 5.1, calcolandone la media e la media delle deviazioni standard. In figura 6.1 sono riportate le metriche: accuracy, precision e recall al variare di uno specifico iperparametro. L'intervallo di confidenza nelle figure è stato calcolato come la media delle deviazioni standard di tutte le fold rispetto ad un certo iperparametro.

Come si può notare dalla figura 6.1a in media la classificazione sembra migliorare all'aumentare della window size, mentre peggiora significativamente all'aumentare del dropout. Inoltre i risultati sono leggermente migliori con un solo layer e con meno unità come mostrato nelle figure 6.1c e 6.1b. Durante l'addestramento si è notato che aggiungendo il dropout nei modelli la loss di validazione cresce con l'aumentare delle epoche portando all'overfitting. Questo porta a fare terminare l'addestramento molto in anticipo a causa dell'early stopping, tranne nei modelli con molti layer ed unità dove l'addestramento è durato di più ma sempre meno rispetto alle 30 epoche. Come si può notare dalla figura 6.2 anche nel test set la loss aumenta all'aumentare del valore di dropout.

Quindi sono stati creati i grafici considerando soltanto i modelli senza dropout, come mostrato in figura 6.3.

Come è possibile notare i risultati delle configurazioni senza dropout sono nettamente migliori rispetto ai precedenti. Inoltre non sono presenti differenze rilevanti tra le varie configurazioni di iperparametri. L'unica cosa a cui prestare attenzione è il fatto che le prestazioni risultano più variabili con window size pari a 20.



(a) window size

(b) layers

(c) unità

(d) dropout

Figura 6.1: Risultati LSTM con dropout

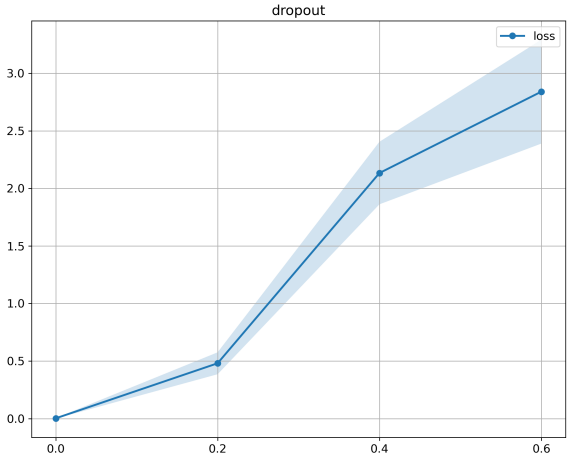
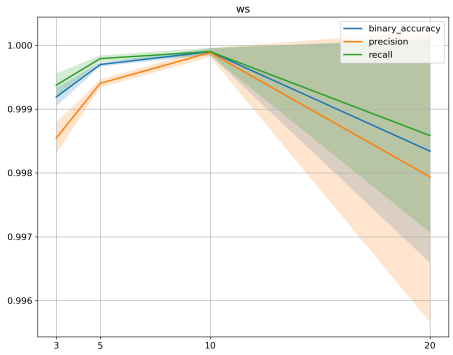
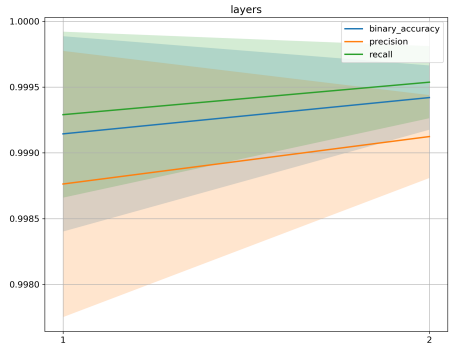


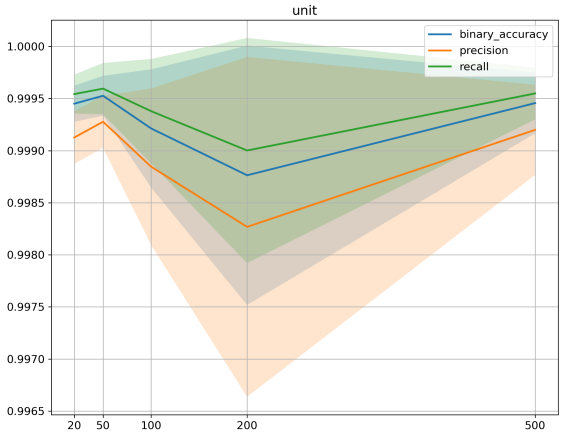
Figura 6.2: Loss al variare del dropout



(a) window size



(b) layers



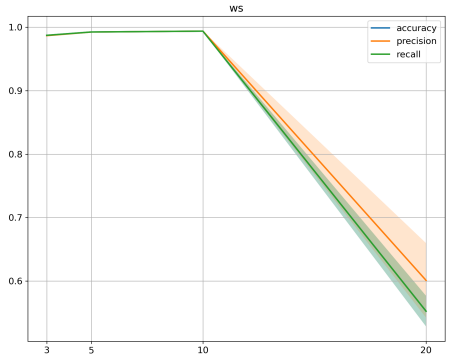
(c) unità

Figura 6.3: Risultati LSTM senza dropout

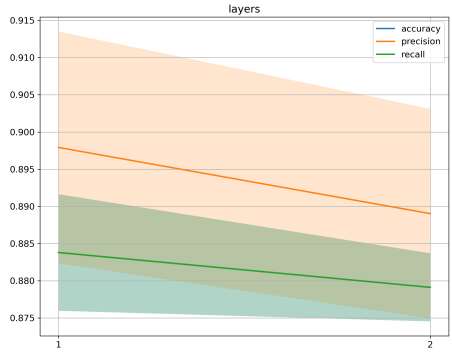
6.1.2 Classificazione ensemble

Dopo aver classificato il test set tramite una rete LSTM binaria, i sample classificati come maligni sono stati aggiunti come input alla rete LSTM multi-classe con la stessa configurazione di iperparametri della rete a valle. I risultati ottenuti sono riportati di seguito. Le metriche sono state mediate in base al numero di sample per classe, quindi sono ponderate. Come si può notare dalla figura 6.4 i risultati sono ottimi anche con la classificazione ensemble. Ma a differenza della classificazione binaria ci sono delle differenze. I particolare l'iperparametro window size fa variare molto le performance della rete, come si vede dalla figura 6.4a. Anche il numero di unità ed il numero di layer fanno variare molto i risultati rispetto ai precedenti risultati. Ponendo la window size pari a 20 tutte le tre metriche degradano, ciò è spiegabile dal fatto che con WS pari a 20 il numero di sample è molto minore dato che relativamente pochi flussi nel dataset raggiungono una lunghezza pari o

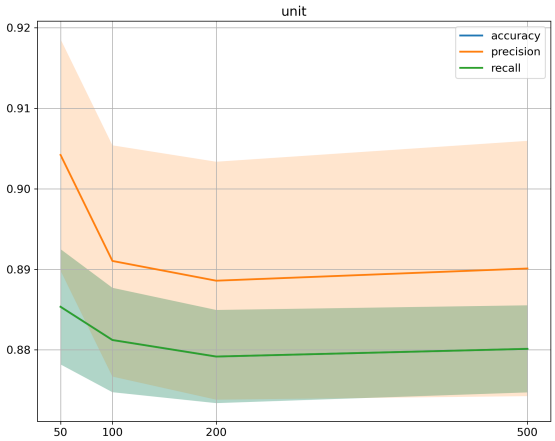
maggiore di 20. Come visto nella sezione 5.2 con WS pari a 10 e 20 si è dovuto fare un undersampling dei benigni perché troppo maggiori rispetto ai maligni. Le performance migliori sia di accuracy che di precision e recall medie sono state ottenute con window size pari a 10, perché essendo ogni sequenza composta da più elementi la classificazione risulta leggermente migliore. Infatti i valori di tali metriche arrivano anche al 99.3% in media sulle 5 fold in questa configurazione, a prescindere dal numero di unità e di layer. Per quanto riguarda il numero di layer e di unità invece, la variazione è dovuta solamente ai risultati ottenuti con WS pari a 20 che ottenendo risultati di gran lunga inferiori fa spostare la media ed aumentare la deviazione standard. Ciò è possibile vederlo nelle figure in appendice B.



(a) window size



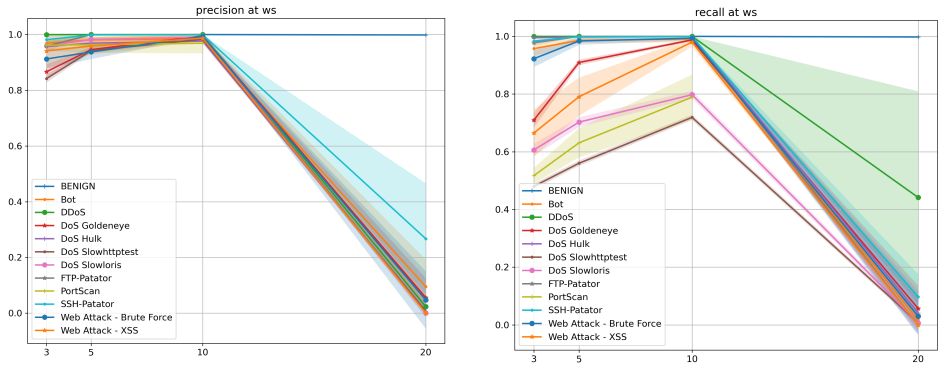
(b) layers



(c) unità

Figura 6.4: Risultati LSTM ensemble

Sono state calcolate anche le metriche precision e recall per ogni classe al variare degli iperparametri. In figura 6.5 sono riportate le due metriche al variare della WS, le variazioni rispetto agli altri iperparametri sono riportati nell'appendice B.



(a) precision per classe

(b) recall per classe

Figura 6.5: Risultati per classe

Come si può notare, tralasciando la WS 20, la precision e la recall di ogni classe sono ottime, in particolare nella WS 10 dove le differenze tra classi sono minime. Mentre quando la WS è pari a 20 essendoci pochi sample di classi maligne le metriche peggiorano, tranne per la classe BENIGN che è sufficientemente popolata dove le performance rimangono ottime.

6.2 Risultati Transformer

I risultati di classificazione binaria mediati tra le 5 fold con i Transformer sono riportati in tabella 6.1

earliness media	**earliness std**	**precision**	**accuracy**	**recall**
0.0873	0.05316	0.963002	0.974961	0.975361

Tabella 6.1: Risultati classificazione binaria Transformer

Come si può notare i risultati sono inferiori di circa il 2% rispetto ai risultati ottenuti con le reti LSTM. Bisogna considerare che si è provata una sola configurazione con i Transformer per motivi di tempo e che è stata utilizzata una WS pari a 100, mentra con le reti LSTM è stata utilizzata al massimo una WS pari a 20. Inoltre l'addestramento dei modelli è terminato molto in anticipo rispetto alle 25 epoche massime a causa dell' early-stopping. Interessante è il valore della metrica earliness: in media dopo l'8% di una traccia la rete riesce a classificare correttamente un flusso, con una deviazione standard di circa il 5%.

I risultati della classificazione ensemble sono inferiori di circa l'1% e sono riportati nella tabella 6.2.

Per quanto riguarda la metrica earliness invece i valori sono similari a quelli ottenuti con la classificazione binaria. Le metriche precision e recall calcolate per

earliness media	**earliness std**	**precision**	**accuracy**	**recall**
0.081732	0.053372	0.966585	0.965164	0.965164

Tabella 6.2: Risultati classificazione binaria Transformer

ogni classe ottenute con l'architettura Transformer sono riportate nell'appendice B. La maggior parte delle classi ha precision e recall inferiori rispetto ai valori ottenuti con la LSTM. Solo alcune classi come BENIGN e DDoS ottengono risultati migliori rispetto a molte configurazioni LSTM.

7

Conclusioni e sviluppi futuri

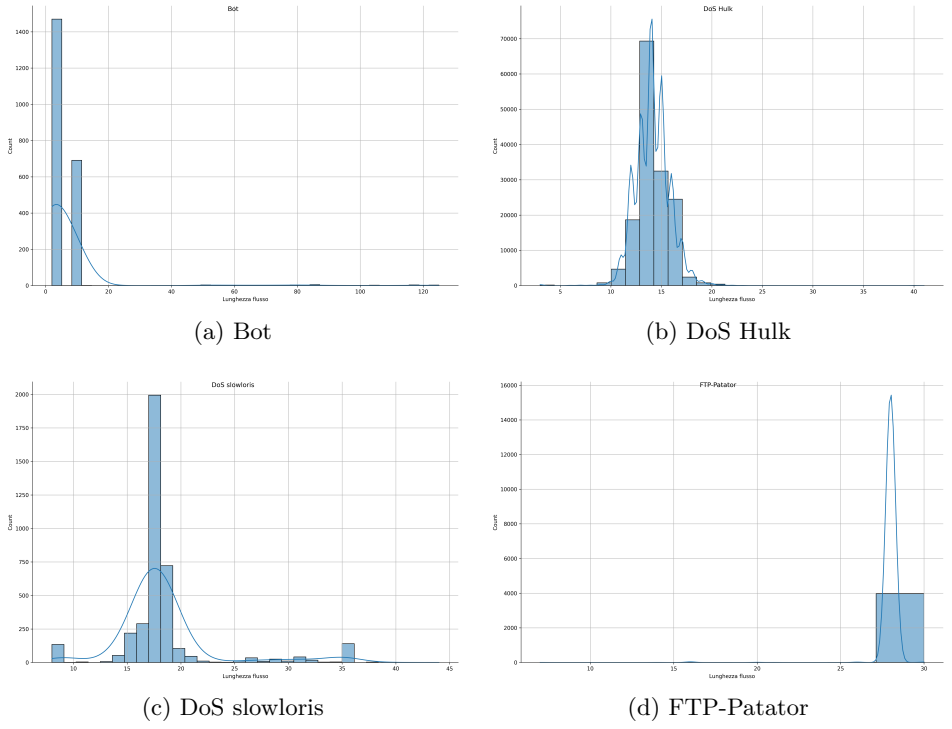
Dai risultati ottenuti è possibile ricavare diverse considerazioni.

Si è riusciti a dimostrare che è possibile classificare il traffico di rete in tempo reale in base solo alle informazioni contenute nei pacchetti di rete con dei risultati sorprendenti, raggiungendo e addirittura superando lo stato dell'arte. In particolare con sole 13 feature si è riusciti a discriminare prima tramite reti LSTM e successivamente tramite i Transformer che è possibile classificare se un flusso di rete è benevolo o malevolo. Per quanto riguarda i risultati ottenuti dalle varie reti, questi non possono essere facilmente confrontabili in quanto sono due tipologie di funzionamenti differenti. La rete LSTM funziona con una finestra scorrevole e come visto nei risultati la dimensione di tale finestra è un parametro critico per ottenere delle buone performance di classificazione. Mentre i Transformer funzionando con finestre incrementali non hanno il problema di dover selezionare un ulteriore iperparametro e fornisco comunque degli ottimi risultati. Quindi con il Transformer è possibile creare un solo modello, con la LSTM invece devono essere addestrati e testati vari modelli per trovare la miglior dimensione per la finestra. Controllando con il passare del tempo che il modello creato sia ancora buono oppure cambiando nuovamente gli iperparametri al fine di ottenere un modello migliore. Essendo il primo lavoro su questo tema i possibili sviluppi futuri sono diversi e riguardano varie tematiche. Prima di tutto bisognerebbe testare la metodologia proposta anche su altri dataset e vedere se i risultati rispecchiano quanto ottenuto con questo lavoro. Possibilmente tale dataset dovrebbe contenere flussi benigni reali e un quantità maggiore di attacchi recenti. Un'ulteriore estensione di questo lavoro potrebbe essere fatta analizzando i pacchetti non i base ad ogni specifico flusso, ma in intervalli di tempo regolare. Questo perché attacchi distribuiti come DDoS e Botnet provengono da diverse macchine in un certo intervallo di tempo. Quindi per scovare queste tipologie di attacchi sarebbe più utile controllare simultaneamente i pacchetti provenienti da varie macchine. Sei risultati dovessero consolidarsi anche su altri dataset, si potrebbe in seguito sviluppare un IDS basato su questa tipologia di algoritmi.

A

Appendice

A.1 Distribuzioni delle lunghezze dei flussi maligni



(a) Bot

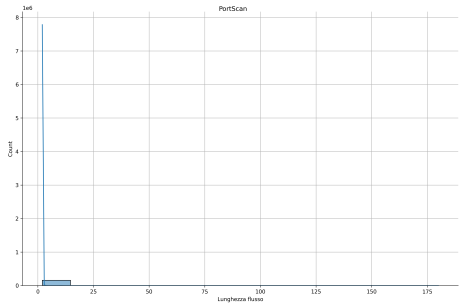
(b) DoS Hulk

(c) DoS slowloris

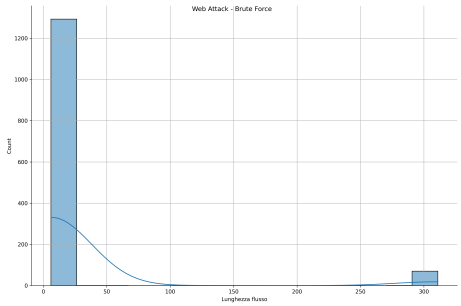
(d) FTP-Patator

Figura A.1: Distribuzione lunghezza flussi pt.3

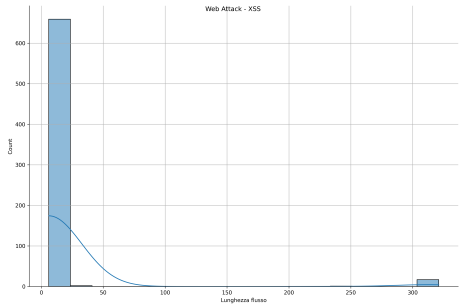
L'attacco Heartbleed essendo formato da un solo flusso non ha una distribuzione.



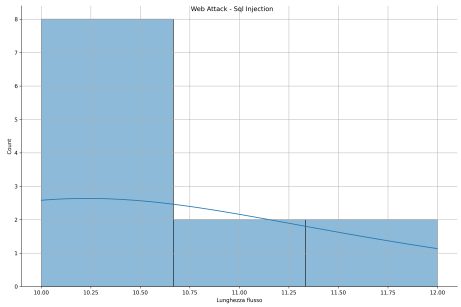
(a) PortScan



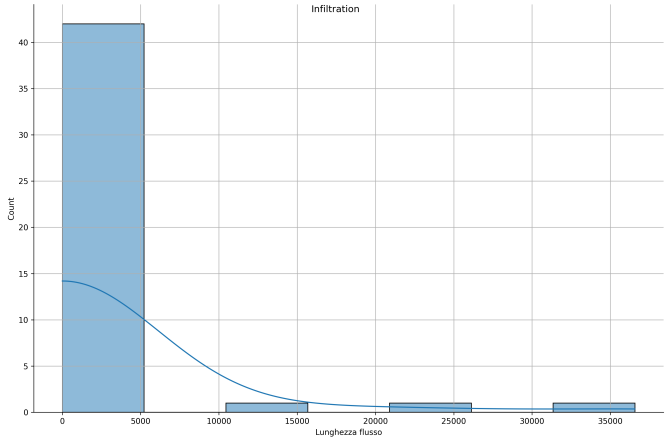
(b) Web Attack - Brute Force



(c) Web Attack - XSS



(d) Web Attack - Sql Injection



(e) Infiltration

Figura A.2: Distribuzione lunghezza flussi pt.4

A.2 Direct Follower Graph

In questa sezione sono riportate tutti i Direct Follower Graph ottenuti per ogni classe, tranne per la classe BENIGN perché troppo numerosa.

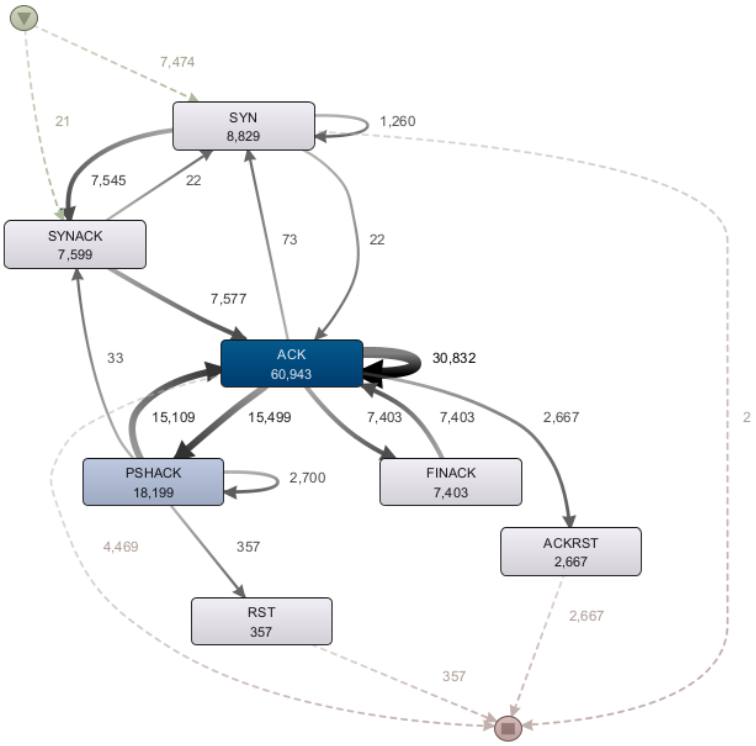


Figura A.3: Direct Follower Graph DoS Goldeneye

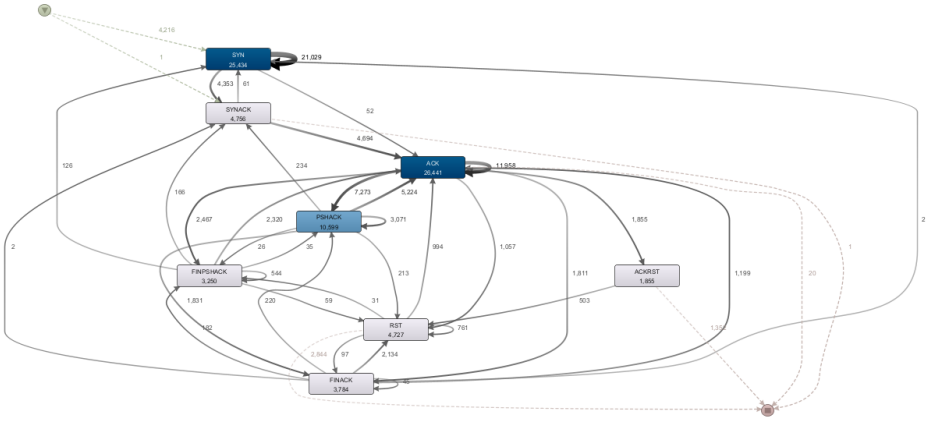


Figura A.4: Direct Follower Graph DoS Slowhttptest

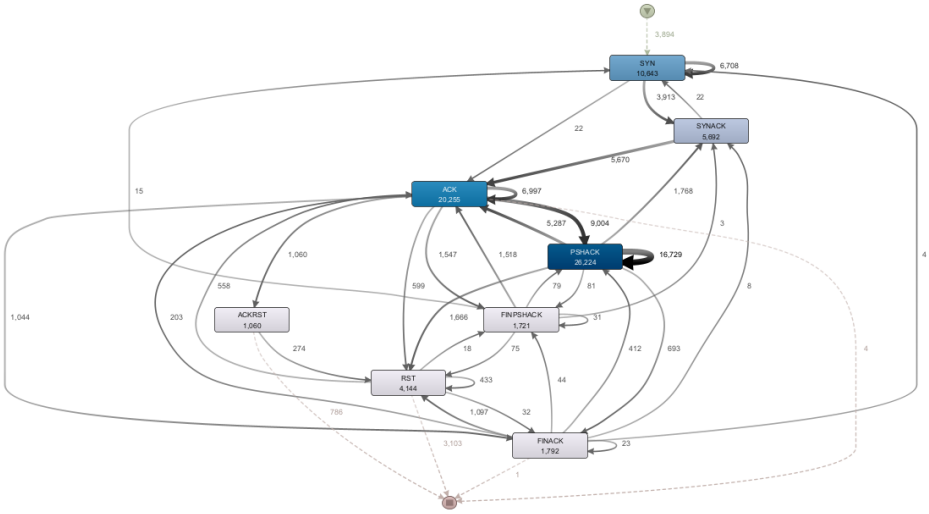


Figura A.5: Direct Follower Graph DoS Slowloris

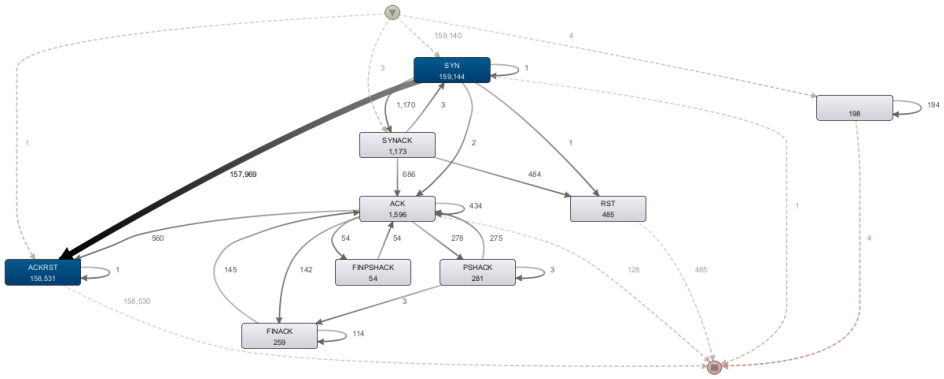


Figura A.10: Direct Follower Graph Port Scan

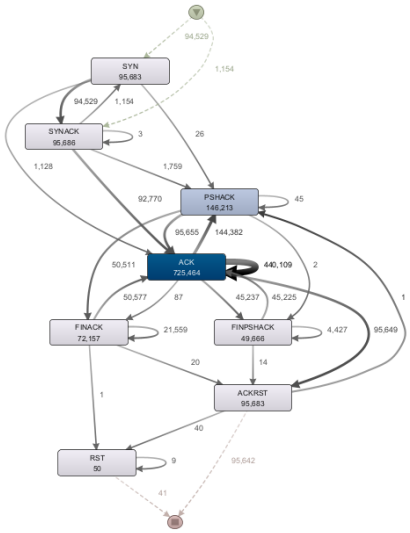


Figura A.6: Direct Follower Graph DDoS

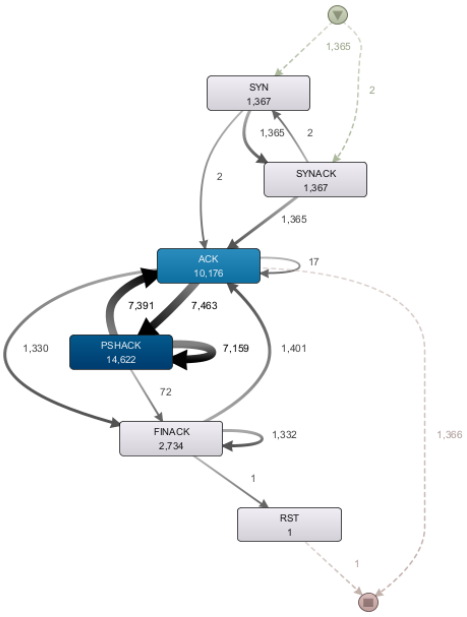


Figura A.7: Direct Follower Graph Brute Force

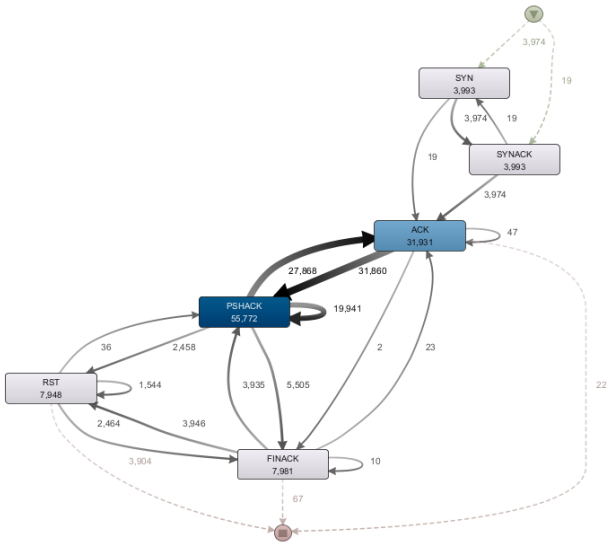


Figura A.8: Direct Follower Graph FTP-Patator

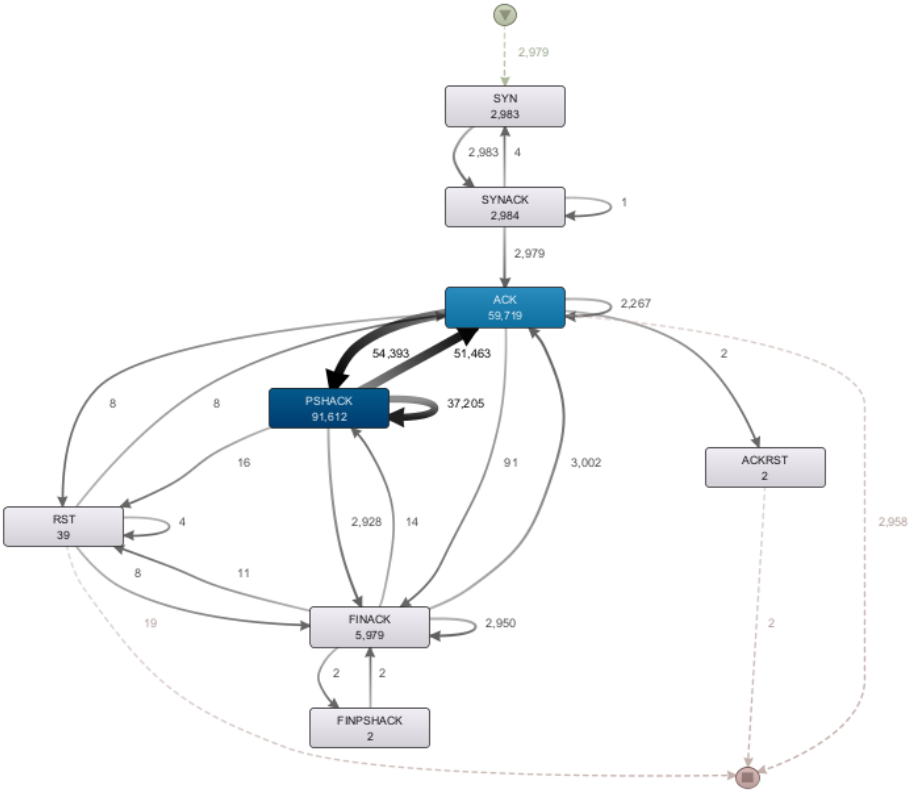


Figura A.9: Direct Follower Graph SSH-Patator

A.3 Petri Net

In questa sezione sono riportate tutte le reti di petri ottenute per ogni classe, tranne per la classe BENIGN perché troppo numerosa.

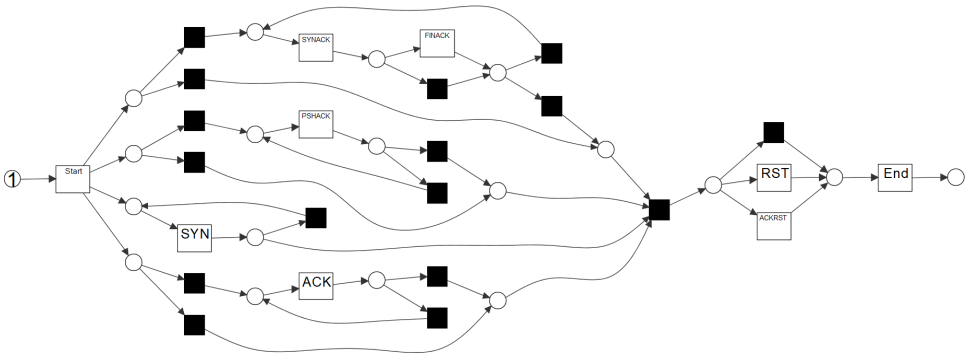


Figura A.11: Petri Net DoS Goldeneye

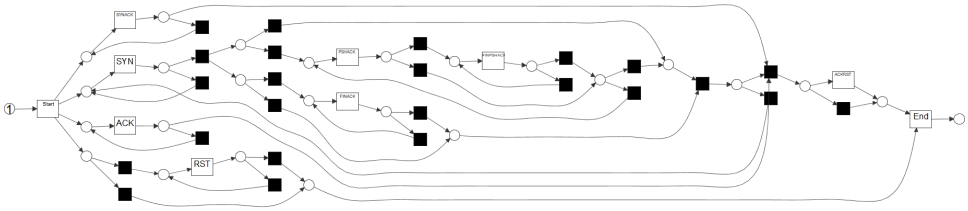


Figura A.12: Petri Net DoS Slowhttptest

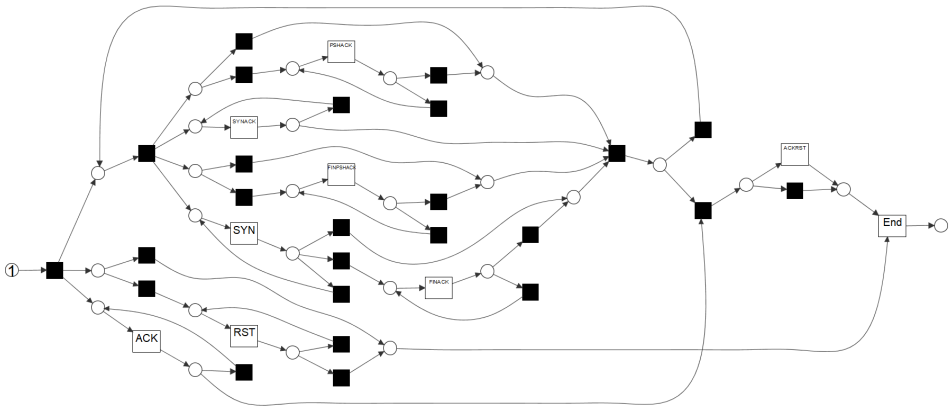


Figura A.13: Petri Net DoS Slowloris

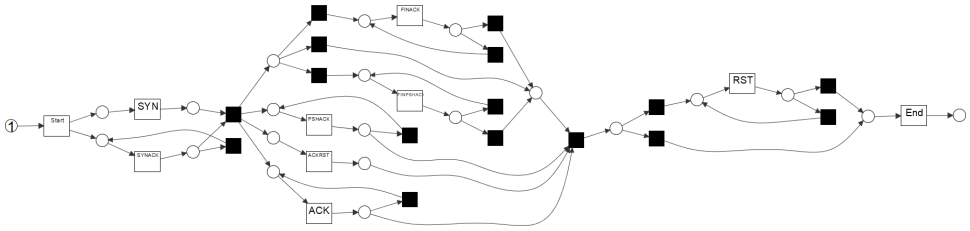


Figura A.14: Petri Net DDoS

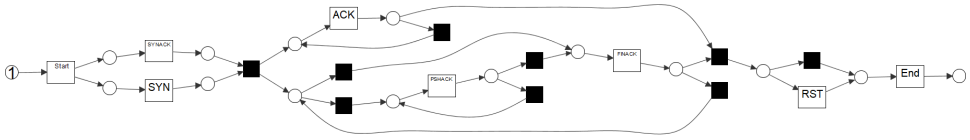


Figura A.15: Petri Net Brute Force

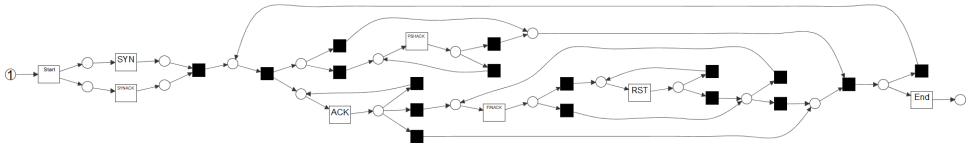


Figura A.16: Petri Net FTP Patator

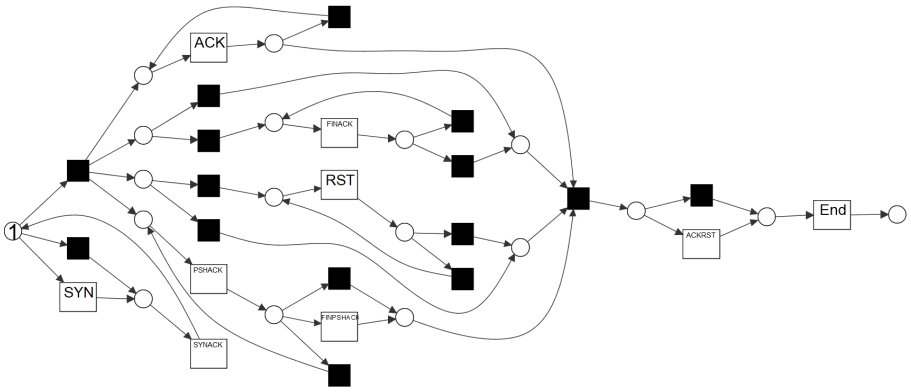


Figura A.17: Petri Net SSH Patator

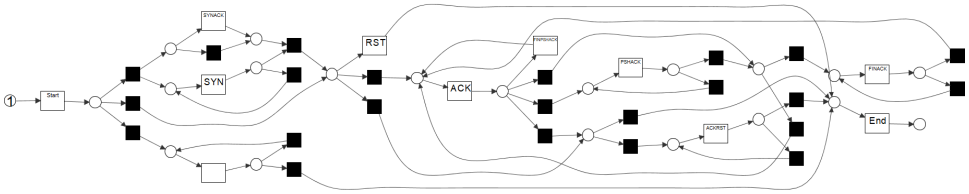


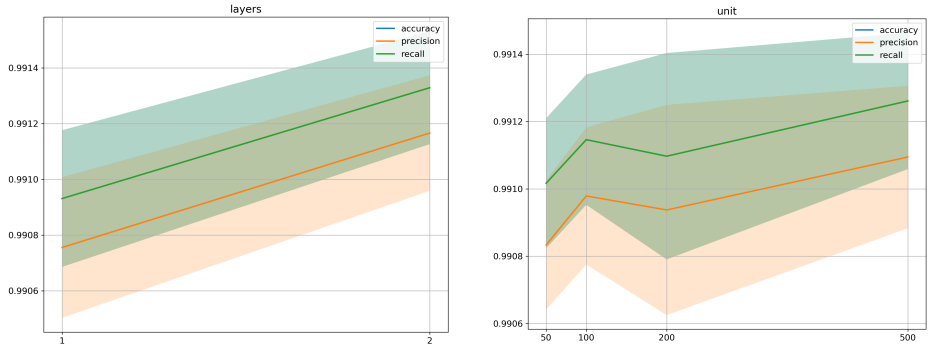
Figura A.18: Petri Net Port Scan

B

Appendice

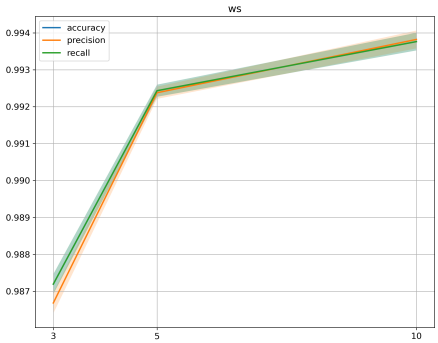
B.1 Risultati ottenuti con l'architettura LSTM

In figura B.1 sono riportati i valori di accuracy, precision e racall medi ottenuti con le reti LSTM, fino alla window size 10. Come si può notare le variazioni sono molto più contenute perché le variazioni maggiori sono generate appunto con la window size 20.



(a) Layer

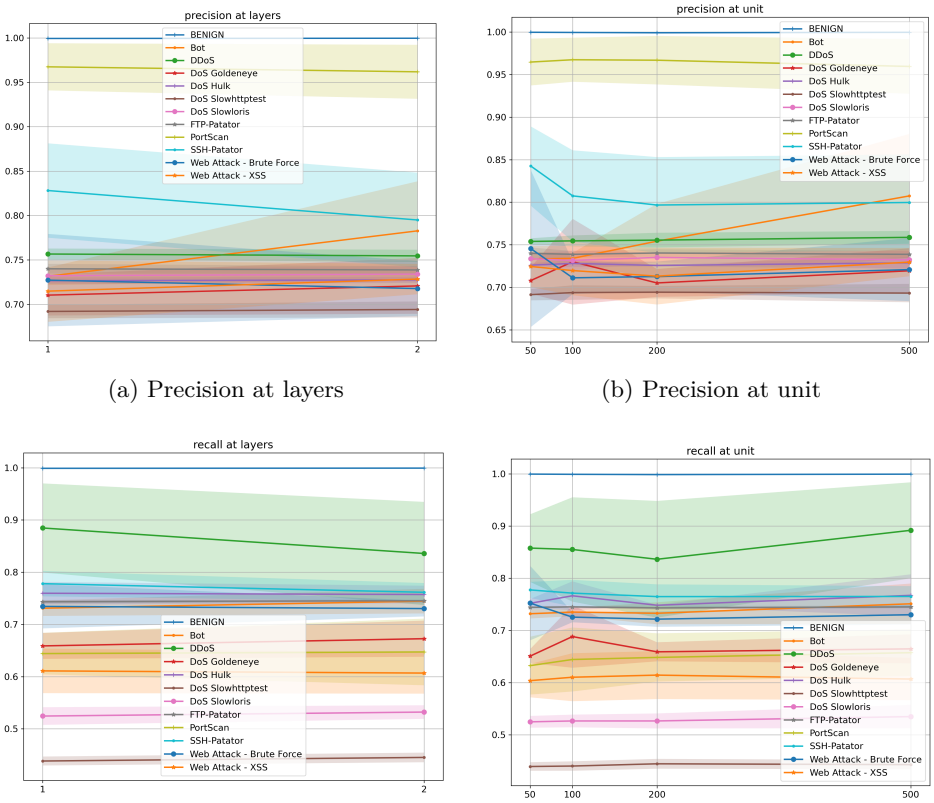
(b) Unità



(c) Window size

Figura B.1: Variazione delle metriche con WS fino a 10

In figura B.3 sono riportati i valori di precision per ogni classe nel dataset ottenuti con le reti LSTM, fino alla window size 20.



(a) Precision at layers

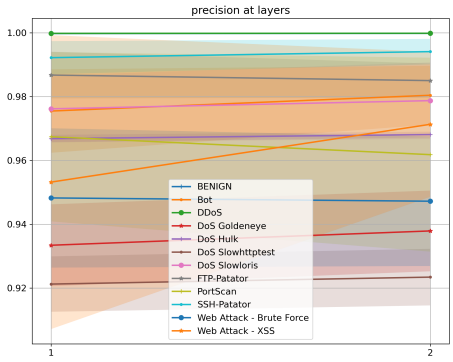
(b) Precision at unit

(c) Recall at layers

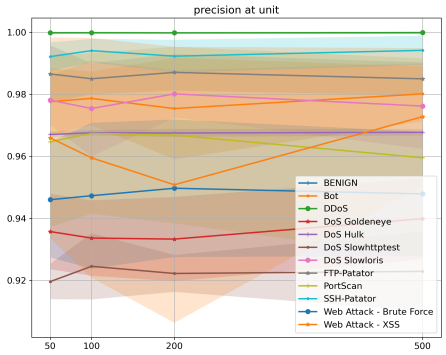
(d) Recall at unit

Figura B.2: Precision e recall per classe al variare degli iperparametri

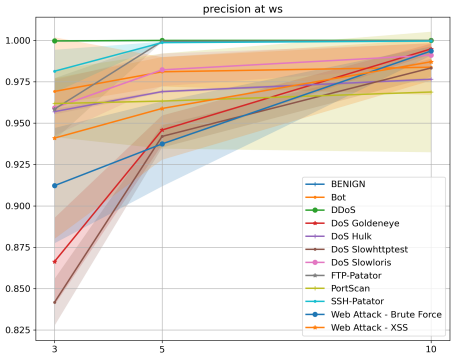
In figura B.3 sono riportati i valori di precision per ogni classe nel dataset ottenuti con le reti LSTM, fino alla window size 10.



(a) Layer



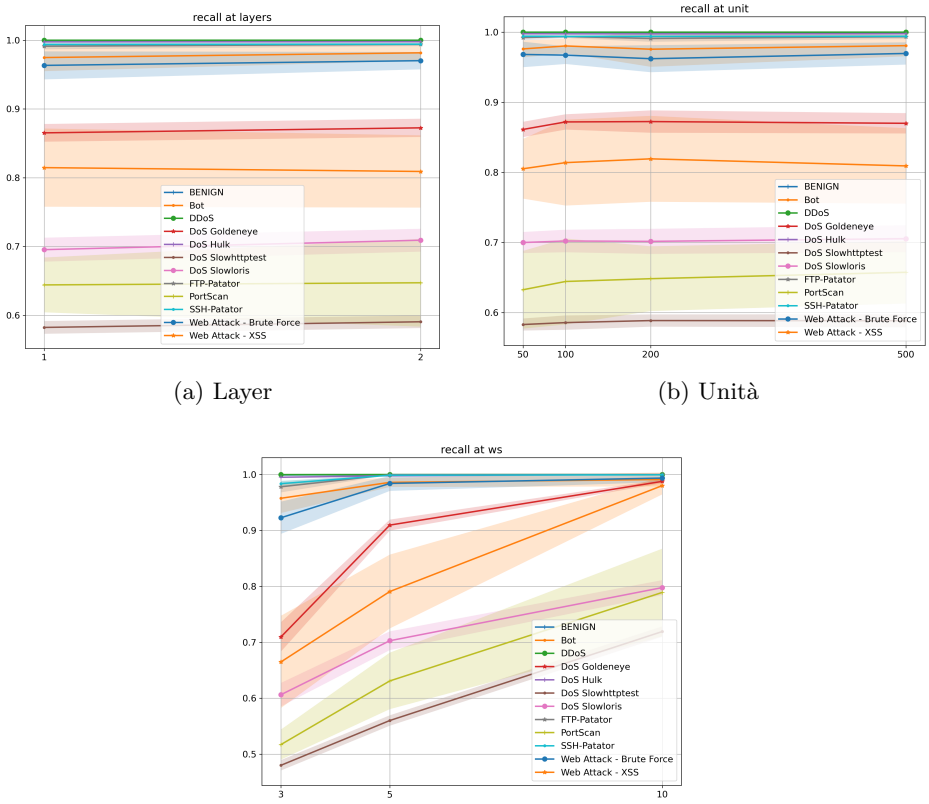
(b) Unità



(c) Window size

Figura B.3: Precision per classe

In figura B.4 sono riportati i valori di recall per ogni classe nel dataset ottenuti con le reti LSTM, fino alla window size 10.



(a) Layer

(b) Unità

(c) Window size

Figura B.4: Recall per classe

B.2 Risultati ottenuti con l'architettura Transformer

In tabella B.1 sono riportate i valori della metrica precision per ogni classe mediate per i 5 fold con l'architettura Transformer.

In tabella B.2 sono riportate i valori della metrica recall per ogni classe mediate per i 5 fold con l'architettura Transformer.

BENIGN	Bot	DDoS	DoS Goldeneye	DoS Hulk	DoS Slowhttptest	DoS Slowloris	FTP-Patator	PortScan	SSH-Patator	Web Attack - Brute Force	Web Attack - XSS
0.983337712	0.87214	0.99896	0.87933041	0.911870996	0.90162	0.940505125	0.968884101	0.885007718	0.957412167	0.727828863	0.925397161

Tabella B.1: Precision multi-classe Transformer

BENIGN	Bot	DDoS	DoS Goldeneye	DoS Hulk	DoS Slowhttptest	DoS Slowloris	FTP-Patator	PortScan	SSH-Patator	Web Attack - Brute Force	Web Attack - XSS
0.974683879	0.774049	0.925962	0.716338164	0.996306	0.825276847	0.781295915	0.901485953	0.418338166	0.942081579	0.713455461	0.15603496

Tabella B.2: Recall multi-classe Transformer

Riferimenti bibliografici

1. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.
2. Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSP*, 2018.
3. Mohamed Amine Ferrag, Leandros Maglaras, Sotiris Moschoyiannis, and Helge Janicke. Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study. *Journal of Information Security and Applications*, 50:102419, 2020.
4. Nour Moustafa. The bot-iot dataset, 2019.
5. Mossa Ghurab, Ghaleb Gaphari, Faisal Alshami, Reem Alshamy, and Suad Othman. A detailed analysis of benchmark datasets for network intrusion detection system. 04 2021.
6. Joffrey Leevy and Taghi Khoshgoftaar. A survey and analysis of intrusion detection models based on cse-cic-ids2018 big data. *Journal of Big Data*, 7, 11 2020.
7. Mingyi Zhu, Kejiang Ye, Yang Wang, and Cheng-Zhong Xu. A deep learning approach for network anomaly detection based on amf-lstm. In Feng Zhang, Jidong Zhai, Marc Snir, Hai Jin, Hironori Kasahara, and Mateo Valero, editors, *Network and Parallel Computing*, pages 137–141, Cham, 2018. Springer International Publishing.
8. Peng Lin, Kejiang Ye, and Cheng-Zhong Xu. *Dynamic Network Anomaly Detection System by Using Deep Learning Techniques*, pages 161–176. 06 2019.
9. N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, Jun 2002.
10. Yuyang Zhou, Guang Cheng, Shanqing Jiang, and Mian Dai. Building an efficient intrusion detection system based on feature selection and ensemble classifier. *Computer Networks*, 174:107247, Jun 2020.
11. Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Stefanos Gritzalis. Intrusion detection in 802.11 networks: empirical evaluation of threats and a public dataset. *IEEE Communications Surveys & Tutorials*, 18(1):184–208, 2016.
12. Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A. Ghorbani. A detailed analysis of the kdd cup 99 data set. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pages 1–6, 2009.
13. Jiyeon Kim, Jiwon Kim, Hyunjung Kim, Minsun Shim, and Eunjung Choi. Cnn-based network intrusion detection against denial-of-service attacks. *Electronics*, 9(6), 2020.
14. Hao Zhang, Shumin Dai, Yongdan Li, and Wenjun Zhang. Real-time distributed-random-forest-based network intrusion detection system using apache spark. In *2018*

IEEE 37th International Performance Computing and Communications Conference (IPCCC), pages 1–7, 2018.
15. Dongzi Jin, Yiqin Lu, Jiancheng Qin, Zhe Cheng, and Zhongshu Mao. Swiftids: Real-time intrusion detection system based on lightgbm and parallel intrusion detection mechanism. *Computers & Security*, 97:101984, 07 2020.
16. Aechan Kim, Mohyun Park, and Dong Hoon Lee. Ai-ids: Application of deep learning to real-time web intrusion detection. *IEEE Access*, 8:70245–70261, 2020.
17. Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. A survey of network-based intrusion detection data sets. *Computers & Security*, 86:147–167, Sep 2019.
18. Sebastián García, Martin Grill, Jan Stiborek, and Alejandro Zunino. An empirical comparison of botnet detection methods. *Computers & Security*, 45:100–123, 09 2014.
19. Arash Habibi Lashkari. Cicflowmeter-v4.0 (formerly known as iscxflowmeter) is a network traffic bi-flow generator and analyser for anomaly detection. https://github.com/iscx/cicflowmeter, 08 2018.
20. sweetsoftware. Ares. `https://github.com/sweetsoftware/Ares`, 2017.
21. lanjelot. Patator. `https://github.com/lanjelot/patator`, 2011.
22. digininja. Dvwa. `https://github.com/digininja/DVWA`, 2015.
23. jseidl. Goldeneye. `https://github.com/jseidl/GoldenEye`, 2012.
24. shekyan. Dos slowhttptest. `https://github.com/shekyan/slowhttptest/wiki`, 2015.
25. grafov. Dos hulk. `https://github.com/grafov/hulk`, 2012.
26. Ddos loic. `https://www.imperva.com/learn/ddos/low-orbit-ion-cannon/`, 2008.
27. robertdavidgraham. Heartleech. `https://github.com/robertdavidgraham/heartleech`, 2014.
28. Iman Sharafaldin, Amirhossein Gharib, Arash Habibi Lashkari, and Ali Ghorbani. Towards a reliable intrusion detection benchmark dataset. *Software Networking*, 2017:177–200, 01 2017.
29. Chiara Di Francescomarino, Marlon Dumas, Fabrizio Maria Maggi, and Irene Teinemaa. Clustering-based predictive process monitoring, 2015.
30. Tensorflow. Positional encoding tensorflow. `https://www.tensorflow.org/text/tutorials/transformer#positional_encoding`, 2021.
31. emla2805. Vision transformer. `https://github.com/emla2805/vision-transformer`, 2020.
32. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.