Indice:

1. Introduzione
   1. Definizione del problema
   2. Struttura della tesi
2. Traffico di rete
   1. Pacchetti di dati
   2. Zeek
3. Process discovery
   1. Trace di eventi
   2. Conversione di pacchetti in Trace
   3. Reti di Petri
   4. Fitness di un trace
4. Validazione empirica
   1. CICD 2017
   2. Risultati dell’estrazione dei trace
   3. Etichettatura dei trace
   4. Metriche di valutazione e setting sperimentale
   5. Risultati
5. Conclusioni

Appendice A - Diagramma delle classi

1. Zeek
2. Creazione dei trace dai pacchetti
3. Anomaly detection

Appendice B -Manuale Utente

1. Download e installazione dei software necessari
2. Download del Dataset
3. Generazione di un file xes
4. Avvio sperimentazione

Bibliografia

1. Introduzione
   1. Definizione del problema

Questo lavoro di tesi ha l’obiettivo di sintetizzare un approccio basato su process discovery e conformance checking[1] per **rilevare connessioni anomale**, cioè connessioni con un attacco a livello di sicurezza, in un traffico di rete.

Inizialmente è stato necessario ottenere i pacchetti e memorizzarli all’interno di *log*, le cui righe rappresentano delle connessioni (sequenza di pacchetti con stessi indirizzo IP e porta dell’*host* mittente, come anche indirizzo IP e porta dell’*host* destinatario). Le connessioni possono essere catturate tramite un software *sniffer* di pacchetti come *Wireshark*[2], che memorizza i pacchetti osservati in un file con estensione *.pcap*. I pacchetti sono stati analizzati con *Zeek[3]*, un software a linea di comando che dato in input un file *.pcap*, genera una collezione di file *log*, ognuno dei quali contiene informazioni specifiche per i diversi protocolli adoperati da ogni connessione, e un file *conn.log* che contiene tutte le connessioni catturate nel file *.pcap* e le informazioni comuni a tutte le connessioni.

A questo punto è stato creato un applicativo che converte i file *log* di *Zeek*, in file *xes*, che è una specifica di file *xml*, contenente una serie di *trace*. Ogni trace è una sequenza di *event*, uno per ogni pacchetto raggruppato nella stessa connessione. I file xes sono dei contenitori chiave per il *process mining*, le esecuzioni dei processi da analizzare sono acquisite come dei trace e le attività per ogni trace sono memorizzati negli eventi che compongono il trace.

I *trace,* in questo caso, sono identificati dalla quadrupla indirizzo IP sorgente, porta sorgente, indirizzo IP destinazione, porta di destinazione, mentre, gli eventi, sono tutti i pacchetti che fanno parte di una stessa connessione.

Una volta trasformato il traffico di rete in trace, si eseguiranno le seguenti fasi: process discovery con la generazione delle *PetriNet* e calcolo dei livelli conformità dei trace alla PetriNet tramite l’operazione di *conformance*. La **prima fase** ha come obiettivo quello di generare le *PetriNet* per ciascuno degli attributi catturati nei *trace* normali. Gli attributi estratti con Zeek sono a turno considerati come l’attività dell’evento. Successivamente si calcolano i livelli di *conformance* di ulteriori connessioni normali. La **seconda fase** ha l’obiettivo di calcolare i livelli di *conformance* di nuovi trace rispetto alle *PetriNet* generate durante la prima fase.

Infine, ciascun trace è considerato anomalo se il valore di conformità calcolato per un numero di attributi è inferiore ad con una soglia scelta dall’utente.

* 1. Struttura della tesi

La tesi è stata suddivisa in tre sezioni principali:

* Analisi e studio del traffico di rete per generare dei *dataset*, analisi del software (*Zeek*) che è stato adoperato, e che tipo di file vengono generati;
* Studio dei file generati Progettazione e implementazione dell’applicativo per generare il *dataset* formato dai file *log* generati da *Zeek* e convertirli in *xes*;
* Utilizzo di *PetriNet* e *Anomaly Detector* per predire se delle connessioni siano anomale, ovvero, con il quale si sta effettuando un attacco a livello di sicurezza di rete, oppure se fanno parte di un traffico di rete normale.

Nel capitolo 2 si effettua un’introduzione spiegando i protocolli di connessioni di rete della specifica TCP/IP, approfondendo con il sotto-capitolo 2.1 i pacchetti a livello di rete e suddividendoli in pacchetti TCP e datagrammi UDP, come vengono catturati e in che formato vengono memorizzate le catture; mentre, nel sotto-capitolo 2.2 viene spiegato l’utilizzo di *Zeek* per la conversione da file *pcap* in file *log*.

Il capitolo 3 si concentra inizialmente sulla definizione e struttura dei *trace* e degli *event* (sotto-capitolo 3.1), nel sotto-capitolo 3.2 sulla progettazione e l’implementazione del software utilizzato per la generazione dei file *xes.* Il sotto-capitolo 3.3 fornisce la definizione e la struttura delle reti di Petri e come vengono utilizzate nel progetto, il sotto-capitolo 3.4, infine, descrive come viene effettuato il *fitness* di un *trace*, quale algoritmo di *fitness* viene utilizzato e il *dataset* risultante.

Il capitolo 4 racchiude la fase di sperimentazione. Inizialmente viene descritto il *dataset* di pacchetti utilizzato in questa tesi (sotto-capitolo 4.1), inseguito, nel sotto-capitolo 4.2, vengono descritti i risultati dell’estrazione dei *trace*, per ogni giornata del *dataset*. Il sotto-capitolo 4.3 pone l’attenzione sulla tecnica di etichettatura dei *trace*, mentre il successivo (sotto-capitolo 4.4) spiega come è stata impostata la sperimentazione e quali metriche di valutazione sono state utilizzate per misurare la qualità della validazione della sperimentazione.

Nel capitolo 5 si raccolgono le conclusioni della progettazione, sullo studio effettuato, e sui risultati della sperimentazione.

Nell’appendice A sono presenti i diagrammi delle classi, nello specifico:

* Appendice A.1: il diagramma derivato dallo studio degli output di *Zeek*;
* Appendice A.2: il diagramma progettato della sezione del software per la conversione dei file *log* in *xes*;
* Appendice A.3: il diagramma progettato della sezione del software che si occupa della sperimentazione.

Nell’appendice B contiene il manuale per guidare l’utente nelle seguenti fasi:

* Appendice B.1: *download* dei *software* necessari (Appendice B.1);
* Appendice B.2: download dei file del Dataset;
* Appendice B.3: impostazione del file *config.ini* e avvio dell’applicativo per la generazione dei file *xes*;
* Appendice B.4: impostazione del file *config.ini* e avvio della sperimentazione.

1. Traffico di rete

I calcolatori per poter comunicare tra di loro, devono essere collegati tra di loro tramite un mezzo fisico, di solito un cavo o l’etere. Dopo essere stati collegati tra loro, direttamente o tramite degli intermediari, i calcolatori possono iniziare a comunicare, e questo è possibile perché sono programmati tramite uno stesso protocollo di rete, chiamato *Internet Protocol[4]* (o IP), che permette di inviare e ricevere messaggi in due protocolli di rete principali: UDP[5] e TCP[6] (rispettivamente *“User Datagram Protocol”* e “*Transmission Control Protocol*”).

Il protocollo **UDP** non è orientato alla connessione, permette, cioè, di inviare e ricevere pacchetti di dati in maniera molto veloce ed efficiente, ma senza sicurezza che questi arrivino effettivamente a destinazione. Questo protocollo viene adoperato, solitamente, quando non è necessario che tutti i pacchetti arrivino a destinazione e che vengano inviati e ricevuti più velocemente possibile, quindi in applicazioni di *live streaming* o di chiamate di tipo VOIP, o in protocolli come il DNS, dove anche se un pacchetto dovesse essere perso, potrebbe sempre essere inviato nuovamente.

Il protocollo **TCP** permette di inviare e ricevere pacchetti con sicurezza che arrivino a destinazione o che, in caso di perdita di pacchetto per un qualsivoglia motivo indipendente dagli interlocutori (interferenza, traffico elevato, …), il pacchetto verrà ritrasmesso. Questo è possibile perché TCP è un protocollo orientato alla connessione, che consiste in tre fasi principali (come rappresentato schematicamente nella Figura 1):

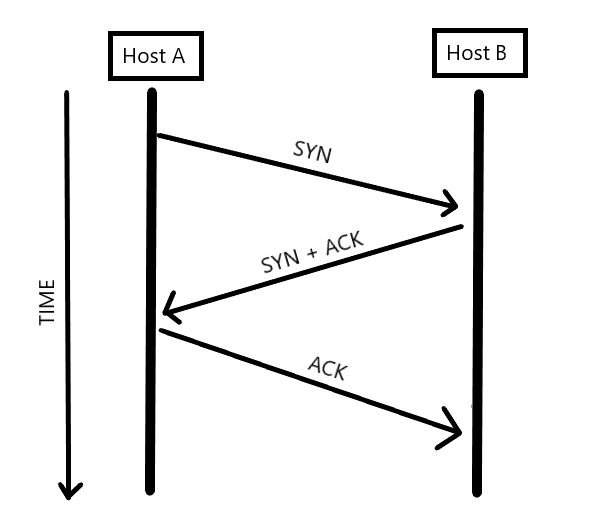


Figura 1. Esempio di avvio di una connessione con protocollo TCP.

1. Avvio della connessione: il *client* richiede al *server* di avviare una connessione tra i due, quindi di voler comunicare. Questo avviene tramite la “stretta di mano a tre vie” (*three-way handshake*), che consiste di tre pacchetti scambiati con dei particolari bit impostati a *true* nella testa del pacchetto:
   1. Il *client* invia un pacchetto col bit *SYN* a *true*;
   2. Il *server* risponde inviando un pacchetto con i bin *SYN+ACK* a *true;*
   3. Il *client* invia un pacchetto col bit *ACK* a *true*;

A questo punto, se tutto è andato a buon fine, la connessione viene avviata con successo e due *host* iniziano a scambiarsi messaggi.

1. Scambio di messaggi: a questo punto ogni messaggio che viene scambiato tra i due *host* deve seguire il seguente schema (rappresentato schematicamente nella seguente Figura 2):

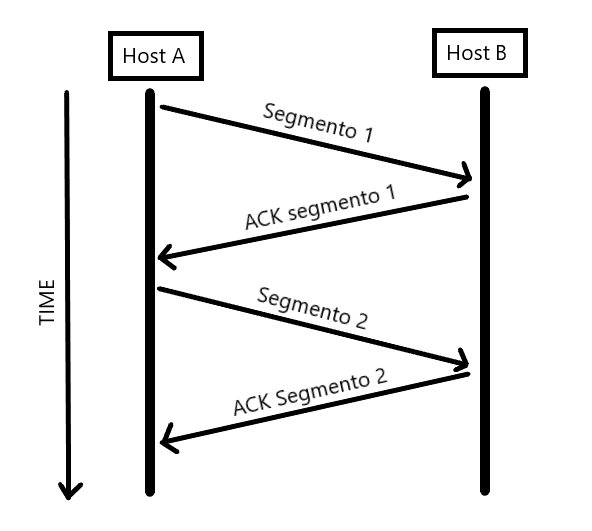


Figura 2. Esempio di scambio di messaggi con protocollo TCP.

* 1. Un *host* invia un messaggio all’altro *host*, il pacchetto ha con sé il numero della sequenza, che corrisponde al numero di bytes inviati dall’inizio della connessione (sommato ad un numero generato casualmente all’apertura della connessione, questo per evitare che dei pacchetti che sono stati persi precedentemente con stessa quadrupla identificativa e numero di sequenza simile, possano contrastare con i pacchetti inviati in questa connessione)
  2. L’*host* ricevente, se il messaggio arriva a destinazione, analizza il pacchetto e invia un pacchetto con il bit *ACK* impostato a *true* e il numero di sequenza pari al numero di sequenza del pacchetto ricevuto sommato alla dimensione in bytes del pacchetto ricevuto. In questo modo è possibile per il mittente sapere se almeno il numero di bytes ricevuti dal destinatario corrispondono a quelli inviati dal mittente. Nel caso in cui il messaggio non arrivi a destinazione, il mittente attenderà un certo periodo di tempo, se non arriva alcuna risposta dal destinatario, questo invierà nuovamente il messaggio.

1. Terminazione della connessione: in questa fase, uno dei due *host* richiede di terminare la connessione, avvengono queste tre fasi (rappresentato schematicamente nella seguente Figura 3):

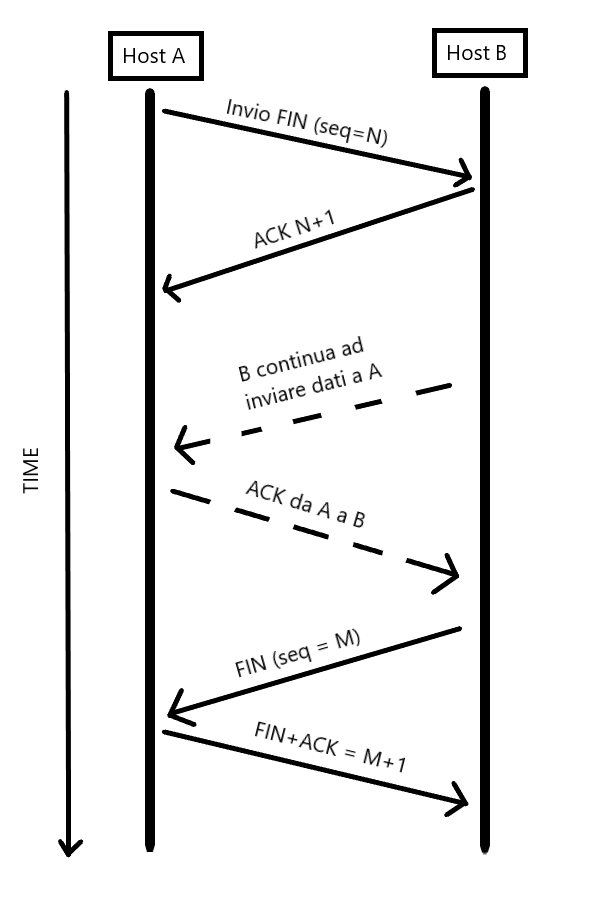


Figura 3. Esempio di chiusura di connessione con protocollo TCP

1. Uno dei due *host* invia un pacchetto col bit *FIN* impostato a *true*, l’altro *host* riceve il pacchetto e risponde con un pacchetto con pacchetto con bit *ACK* a *true*.
2. Se l’*host* ricevente deve ancora inviare dati, continua ad inviarli come avviene nella fase di scambio di messaggi.
3. Quando l’*host* ha concluso la trasmissione di dati, invia un pacchetto con bit *FIN* a *true*, il ricevente risponde con bit *ACK* a *true* e la connessione viene terminata.
   1. Pacchetti di dati

I pacchetti che vengono scambiati tra i vari *host* hanno due strutture principali:

Pacchetto UDP: il pacchetto contiene solo le informazioni essenziali (Figura 4), indirizzo IP e porta degli *host* mittente e destinatario (la porta del mittente è opzionale), la lunghezza del pacchetto, un *checksum* opzionale (per controllare che il pacchetto sia arrivato integro a destinazione), e il *payload* di dati tra trasmettere[7]. Non avendo una connessione vera e propria, avviene solitamente solo l’invio di un messaggio al destinatario e l’eventuale risposta da parte del destinatario verso il mittente.

Immagine che contiene tavolo

Descrizione generata automaticamente

Figura 4. Struttura di un pacchetto UDP

Pacchetto TCP: Il pacchetto TCP contiene diversi campi per gestire il flusso di dati[8] (Figura 5), tra questi i più importanti sono:

* porta sorgente e di destinazione (entrambe obbligatorie);
* numero di sequenza;
* numero di *ACK* (se il flag *ACK* è impostato a *true*);
* Flag per il controllo del protocollo (i più importanti):
  + *ACK*: indica che il pacchetto ha lo scopo di notificare la corretta ricezione del pacchetto appena inviato;
  + *RST*: indica che la connessione non è valida, di solito in caso di grave errore, a volte utilizzato insieme al flag ACK per la chiusura di una connessione;
  + *SYN*: usato per l’apertura della connessione, ha lo scopo di sincronizzare i numeri di sequenza dei due host;
  + *FIN*: usato per la chiusura della connessione;
* *Payload*: dati effettivi del pacchetto da inviare

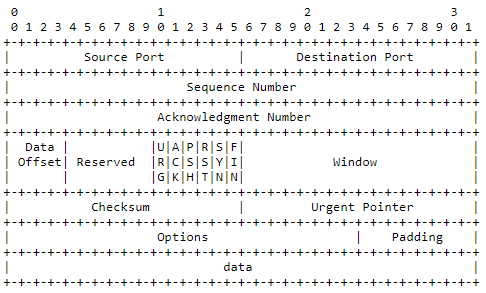


Figura 5. Struttura di un segmento TCP

Il contenuto di dati dei vari pacchetti può essere letto, ma risulta essere comprensibile solo ai programmi degli *host* che inviano e ricevono i messaggi. Nel caso di connessioni *http* (*HyperText Transfer Protocol*[9]) per esempio, che è il protocollo usato per la trasmissione e richiesta di pagine Web, il contenuto è puramente testuale e leggibile da tutti, poiché si può usare l’euristica che la porta utilizzata sia 80, però, dal 2018, il protocollo *http* è stato deprecato poiché non sicuro, ed è stato reso obbligatorio l’uso di *https*[10] il quale lavora sulla porta 443, che obbliga il sito a trasmettere il contenuto dei pacchetti in maniera sicura effettuando la crittazione dello stesso. In questo modo nessuno al difuori del server e del client sono grado di capirne il contenuto. Inoltre, ci sono molte applicazioni di rete che trasmettono dati in formato binario e in un formato proprietario scelto dai progettisti di tali applicazioni, quindi rendendo quasi impossibile l’utilizzo di un’euristica.

Quindi, l’unico modo per interpretare dei pacchetti come **normali** o **anomali**, è controllando che la loro trasmissione avvenga tramite il protocollo TCP sfruttando lo storico dello stato dei *flag* dei pacchetti scambiati.

I pacchetti che vengono inviati e ricevuti possono essere catturati tramite un software di *sniffing* di rete, che consiste nel *bypassare* il protocollo di rete standard che indica al computer di ignorare tutti i pacchetti che non sono diretti a lui, e di reindirizzarli al programma che effettua lo *sniffing*. Il programma adoperato è stato *Wireshark*, un software gratis e open-source per l’analisi di pacchetti, che permette di avviare la cattura dei pacchetti, analizzarli man mano che vengono catturati, visualizzandone il contenuto utilizzando un’euristica basata sui protocolli, se utilizzano delle porte conosciute (Well-known ports[11]), altrimenti mostrano il contenuto del pacchetto in esadecimale. Dopo aver concluso la cattura dei pacchetti, è possibile esportarli in dei file binari con estensione “.pcap”[12] che è uno standard per definire i file contenenti i pacchetti di rete, in questo modo i pacchetti possono essere analizzati successivamente utilizzando altri software, o creando software ad-hoc per studiare i pacchetti.

* 1. *Zeek*

I pacchetti catturati da *Wireshark* sono stati analizzati tramite un programma open-source per l’analisi di sicurezza chiamato **Zeek**. Questo software, con cui si può interagire tramite la linea di comando permette di analizzare un file “pcap” e generare dei file *log* contenenti le connessioni del file in input. Le connessioni vengono ricostruite in base allo standard derivabile dal protocollo TCP/IP, per esempio tutti i pacchetti che condividono la stessa quadrupla indirizzo IP e porta di mittente e destinatario e sfruttando i flag dei vari pacchetti scambiati. Ogni *log* generato equivale ad un file “*csv*”, quindi a tabelle, le cui colonne sono solitamente separate tramite delle virgole[13], in questo caso però il separatore è il carattere di tabulazione. Le righe dei *log* sono le connessioni riconosciute nel file pcap, e le colonne le informazioni relative alle connessioni. Il *log* principale estratto con *Zeek* dai file *pcap* è memorizzato nel file “*conn.log*”[14]. Tale *log* contiene le informazioni relative a tutti i tipi di connessioni e protocolli. Nel particolare contiene le seguenti informazioni:

* i partecipanti della connessione (*Zeek* interpreta l’*host* che ha iniziato la connessione come “*originator*” e quello a cui è stato richiesto di avviare la connessione come “*responder*” analizzando i pacchetti invertendo dove necessario il mittente ed il destinatario per poter raccogliere tutti i pacchetti in un’unica connessione);
* il numero di bytes inviati e ricevuti;
* l’orario di inizio della connessione;
* il protocollo usato.

Gli altri *log* estratti da *Zeek* contengono le informazioni relative a protocolli (per esempio “*dns.log*” contiene le informazioni del protocollo *DNS,* Domain Name System,[15] come: il tipo di query, la classe di query, …), o eventuali file scambiati durante lo scambio di messaggi delle connessioni. Alcuni dei file generati riportano un campo chiamato “*uid*” (*unique identifier*) che identifica la connessione e permette di ritrovarla, se presente, negli altri file generati, come se fosse la chiave esterna di un *database*. I *log* che tracciano i file scambiati condividono anche il campo “*fuid*” (*file unique identifier*) che identifica, quindi, i file scambiati tra le varie tabelle riguardanti solo le informazioni di determinati file. Osservando gli identificatori e i collegamenti logici che implicano, è stato generato uno pseudo-diagramma Entità Relazione interpretando i file come delle Entità (Figura 6 e Figura 7) per capire quali file possono essere utili a questa sperimentazione.

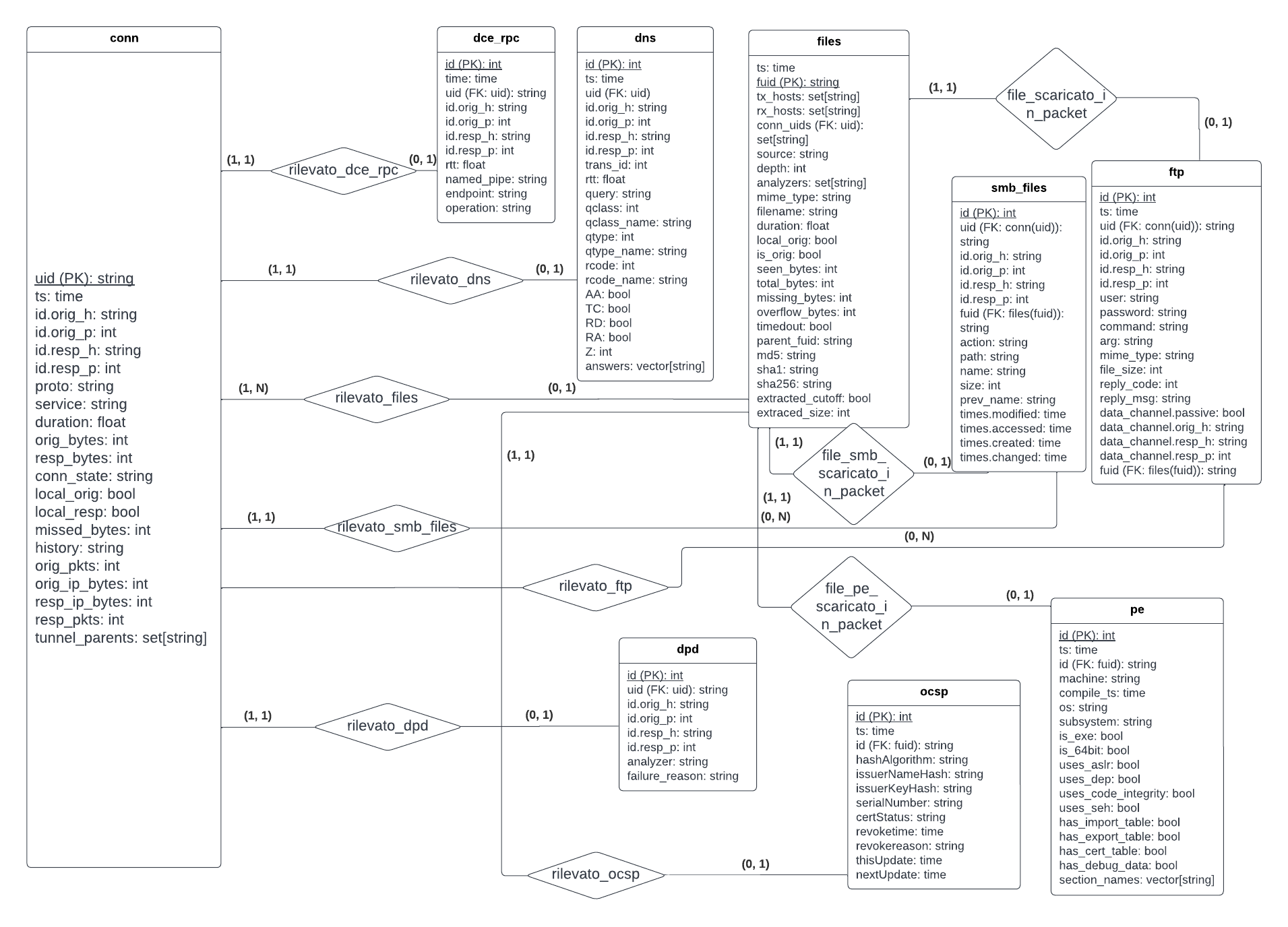
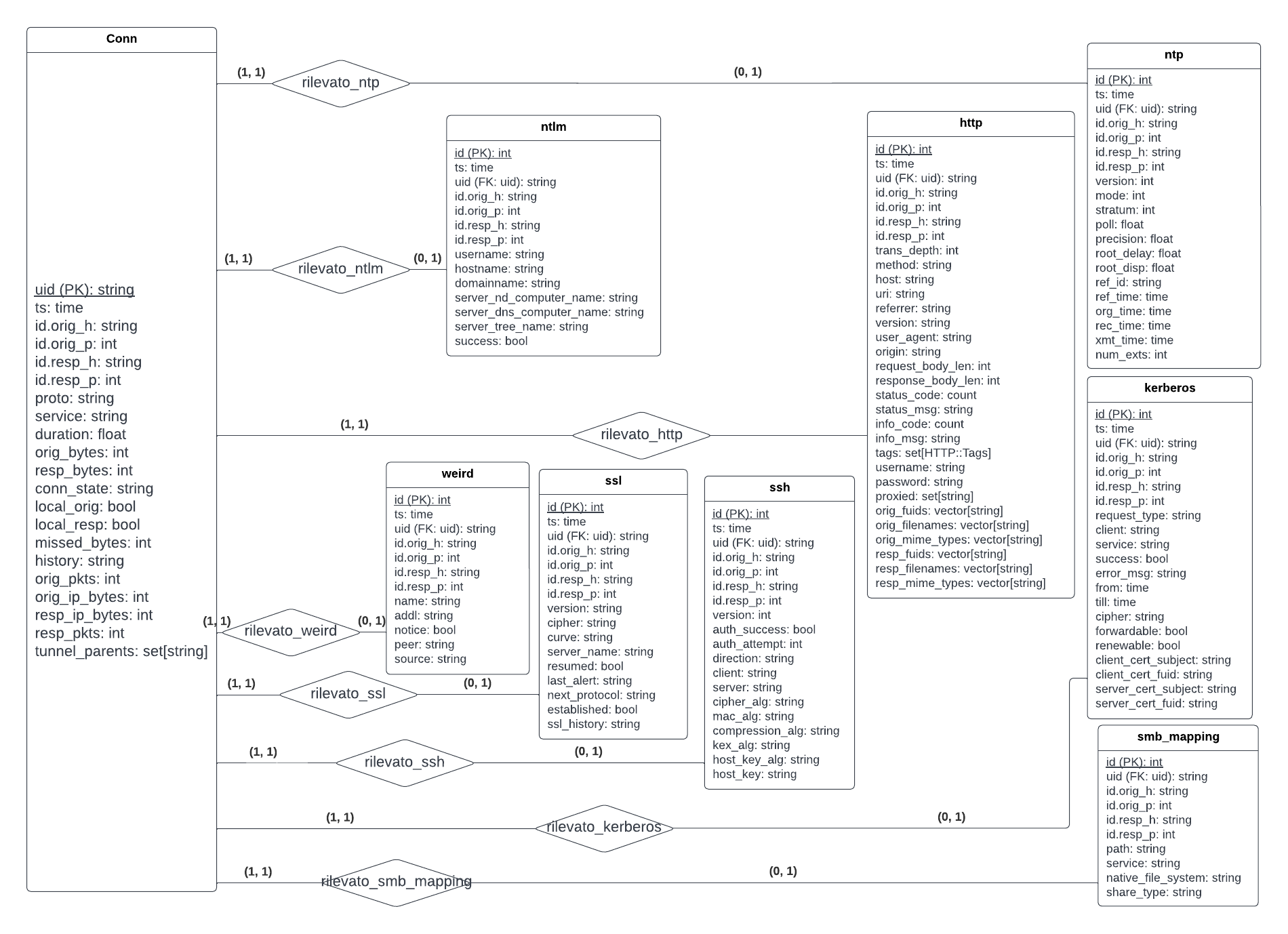


Figura 6. Prima parte dello schema E/R ricavato dai log

Figura 7. Seconda parte dello schema E/R ricavato dai log

Dopo un attento studio sui dati e sugli attributi si è concluso che il file di interesse per questo lavoro di tesi è “conn.log” contenente gli attributi[16]:

* *ts*: timestamp di questa connessione;
* *uid*: id unico di questa connessione;
* *id*: quadrupla (ip orig, port orig, ip resp, port resp);
* *proto*: protocollo del livello di trasporto di questa connessione;
* *service*: identificativo del protocollo applicazione inviato durante la connessione;
* *duration*: durata della connessione;
* *orig\_bytes*: numero di bytes nel payload che l’originatore ha inviato, per TCP potrebbe essere estratto dal numero di sequenza;
* *resp\_bytes*: numero di bytes nel payload del rispondente;
* *conn\_state*: rappresenta lo stato della connessione e può avere valori come: tentativo di connessione ma nessuna risposta, connessione stabilita ma non terminata, connessione rifiutata, “responder” ha inviato un pacchetto con RST attivo, ecc…;
* *local\_orig*: se la connessione è originata localmente, questo valore sarà T, altrimenti F;
* *local\_resp*: se la connessione è stata risposta localmente, questo valore sarà T, altrimenti F;
* *missed\_bytes*: indica il numero di bytes persi nelle lacune di contenuti, che è rappresentativo dei pacchetti persi.;
* *History* memorizza lo stato delle connessioni come stringhe di lettere. Il significato di queste lettere è:
  + s: un SYN senza il bit di ACK impostato a 1;
  + h: un SYN+ACK (“handshake”);
  + a: un puro ACK;
  + d: pacchetto con carico (payload) (“dati”);
  + f: pacchetto con il bit FIN impostato a 1;
  + r: pacchetto con il bit RST impostato a 1;
  + c: pacchetto con un checksum non corretto (si applica anche a pacchetti UDP);
  + g: una lacuna di contenuto (content gap);
  + t: pacchetto con payload ritrasmesso;
  + w: pacchetto con “zero window advertisement”;
  + i: pacchetto inconsistente (es: bit FIN+RST impostati a 1);
  + q: pacchetto con più flag (bit SYN+FIN o SYN+RST impostati a 1);
  + ^: la direzione della connessione è stata invertita dall’euristica di zeek;
* *orig\_pkts*: numero dei pacchetti che l’originatore ha inviato;
* *orig\_ip\_bytes*: numero di bytes al livello IP che l’originatore ha inviato (visti sul cavo, dal campo total\_length nel campo dell’header);
* *resp\_pkts*: numero dei pacchetti che il responder ha inviato;
* *resp\_ip\_bytes:* numero di bytes al livello IP che il responde ha inviato (visti sul cavo, dal campo total\_length nel campo dell’header);
* *tunnel\_parents*: se questa connessione avviene tramite un tunnel, questo campo indica gli uid per qualsiasi connessione genitore incapsulante usata lungo la vita di questa connessione interna.

Si noti che l’attributo “*history*”è una stringa di caratteri alfabetici che viene composta in base ai tipi di pacchetti inviati e ricevuti: se il pacchetto viene inviato dall’*host* che ha avviato la connessione la lettera sarà maiuscola, altrimenti sarà minuscola. In base al contenuto di questo attributo è possibile sapere, per esempio, se una connessione è stata chiusa correttamente o è stato effettuato un *Reset* (pacchetto con *RST* impostato a *true*) che corrisponde ad uno dei possibili attacchi a livello di rete.

1. Process Discovery
   1. Trace di Eventi

Il *process discovery* è l’operazione di generazione di modelli di processi e necessita di *Trace* di *Event*. “Si assuma che sia possibile registrare sequenzialmente degli eventi, tali che un evento si riferisca ad un’attività, ed è relativo ad un particolare caso. […] I casi sono richieste individuali e per caso si può memorizzare un *trace* di eventi. […]. La maggior parte di *event logs* memorizzano informazioni addizionali sugli eventi. Infatti, quando possibile, le tecniche di *process mining* usano informazioni in più come le risorse che eseguono o iniziano l’attività, il *timestamp* degli eventi, o elementi dei dati memorizzati con l’evento.[…]Il primo tipo di process mining è *discovery*. Una tecnica di scoperta prende in input un *log* di *event* e produce un modello senza usare alcuna informazione a-priori. […] Se l’*event log* contiene informazioni su risorse, si possono anche scoprire modelli relativi alle risorse.”[18]

In questo progetto, un trace è identificato dalla tupla:

1. Indirizzo IP di origine;
2. Porta di origine;
3. Indirizzo IP del rispondente;
4. Porta del rispondente;
5. Protocollo utilizzato;
6. Timestamp della prima connessione registrata con i primi cinque valori di questa tupla.

Un trace è quindi una sequenza di eventi dove ogni evento è la sequenza di righe memorizzate in *conn.log* che condividono: IP di origine, porta di origine, IP del rispondente, porta del rispondente e protocollo.

Un evento contiene tutte le informazioni contenute in una riga *conn.log* generato da *Zeek*: ts, service, duration, orig\_bytes, resp\_bytes, conn\_state, missed\_bytes, history, orig\_pkts, orig\_ip\_bytes, resp\_pkts, resp\_ip\_bytes.

3.2 Conversione di pacchetti in trace del formato xes

Nella tesi è stato progettato e realizzato un software per convertire i dati dal *conn.log* nel formato *xes,* che è un formato basato sul formato *XML*, e il suo nome è l’acronimo di *eXtensible Event Stream.* Un file *xes* mantiene la struttura generale di un *event log*: un *Log* contiene un insieme di *traces,* che a loro volta contengono una sequenza di *event*. Ognuno di questi può contenere un insieme di *attributes,* che conterranno i dati. Gli attributi sono fortemente tipizzati e possono essere dei seguenti tipi: stringhe, interi, numeri a virgola mobile o valori *boolean*[19].

Per il progetto di tale software sono stati identificati i seguenti *package* (Figura 8):

* *ConnectionsModule;*
* *EntryPoint;*
* *DiscretizerModule.*

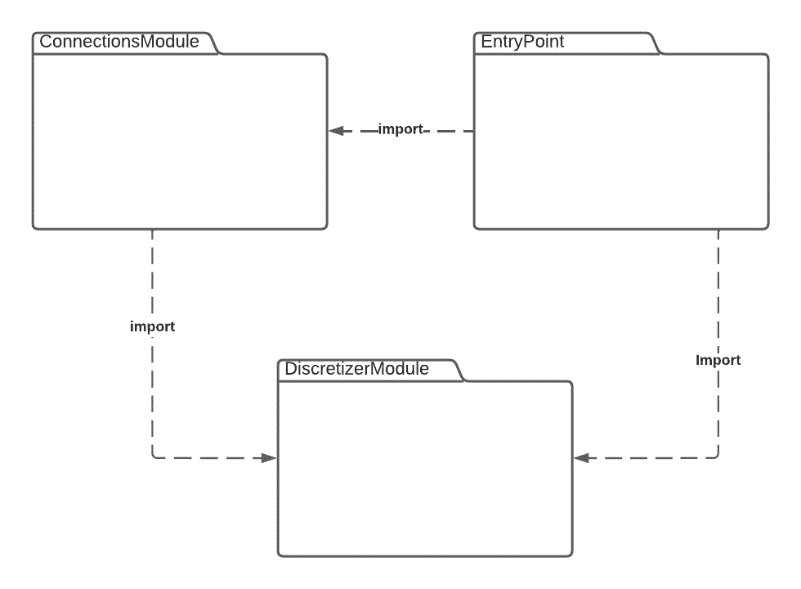
**

Figura 8. Diagramma dei package del software

Il “*package*” ***ConnectionsModule*** (Figura 9) contiene sette cassi:

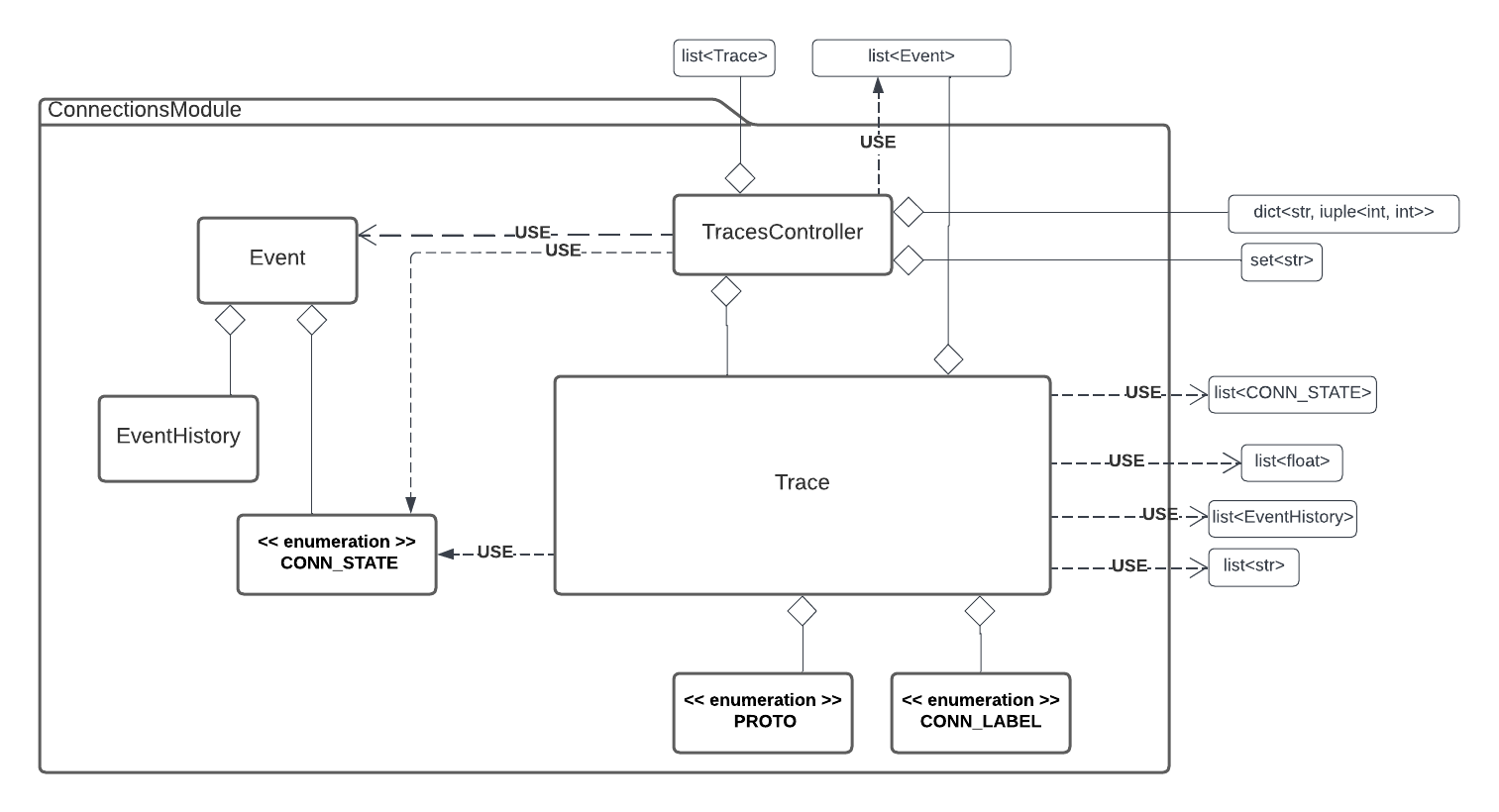


Figura 9. Diagramma delle classi del *package ConnectionsModule*

* *TraceController* (Figura 10) contiene i metodi per leggere il file *conn.log* ed interpretarlo, effettuare la conversione delle righe in *trace* ed *event* e la scrittura del file *xes.* I suoi attributi sono:

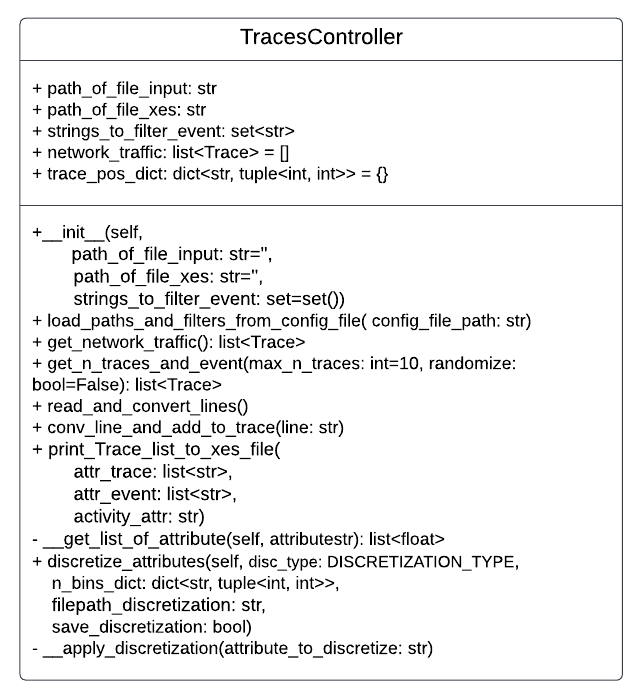


Figura 10. Diagramma della classe *TracesController*

* + *path\_of\_file\_input e path\_of\_file\_xes*:i percorsi dei file di *input* e di *output*;
  + *strings\_to\_filter\_event*: un insieme delle stringhe da rimuovere;
  + *network\_traffic*:la lista dei *trace* derivati dal *log;*
  + *trace\_pos\_dict*: dizionario per rendere più efficiente l’aggiunta degli *event* nei *trace*.
* *Trace* (Figura 11) contiene gli attributi:
  + *Orig\_ip, orig\_port, resp\_ip, resp\_port*: indirizzi e porte degli *host*;
  + *proto*: protocollo utilizzato;
  + *ts\_on\_open*: il *timestamp* del primo *event*;
  + *label*: l’etichetta (di tipo *CONN\_LABEL*) per indicare se si tratta di una connessione normale o anomala;
  + *events*: la lista degli *event*.

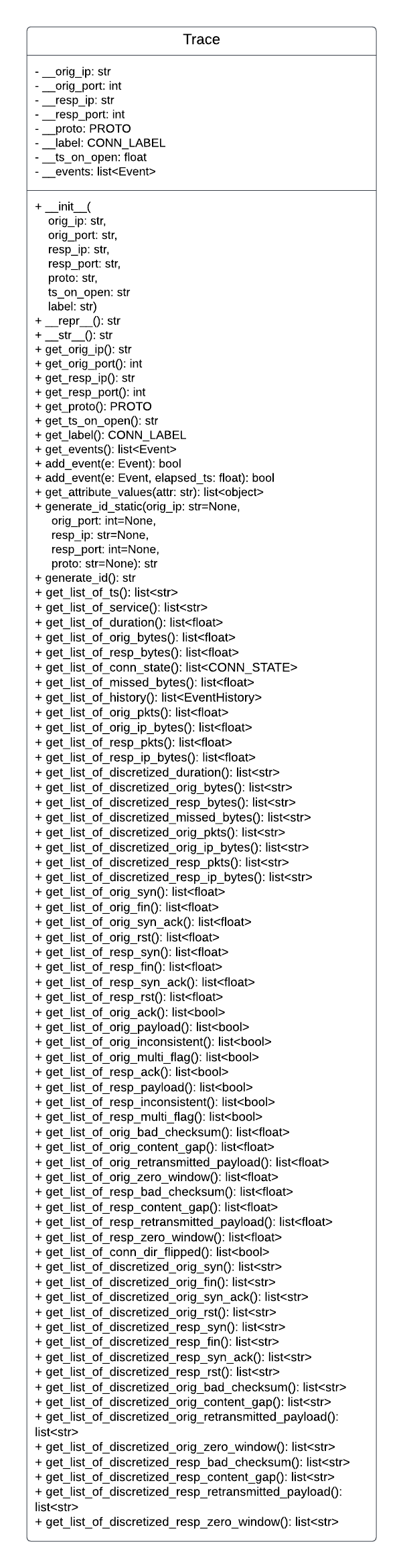


Figura 11. Diagramma della classe *Trace*

La classe include i metodi per realizzare le seguenti operazioni:

* + la lettura delle informazioni del *Trace*;
  + la lettura delle informazioni degli *Event* del *Trace* per ogni attributo degli *Event* e *EventHistory*, tramite generazione di liste dei suddetti valori;
  + discretizzazione degli attributi continui di *Event* e *EventHistory*;
  + la generazione di un “*id*” per il *Trace* (basata sulla concatenazione delle informazioni del *Trace*) usato durante l’aggiunta degli *Event* al *Trace* come input nel dizionario di *TraceController*;
  + l’aggiunta di un *Event* al *Trace*, basandosi anche sul tempo intercorso dal prima *Event* dello stesso *Trace*.
* *Event* (Figura 12) contiene gli attributi:
  + *ts*: *timestamp* di inizio della connessione;
  + *service*: servizio della connessione;
  + *duration*: durata in secondi della connessione;
  + *orig\_bytes*: numero di bytes inviati dall’*host* di origine durante la connessione;
  + *resp\_bytes*: numero di bytes inviati dall’*host* rispondente durante la connessione;
  + *conn\_state*: stato della connessione di tipo *CONN\_STATE*;
  + *missed\_bytes*: numero di bytes persi durante la connessione;
  + *history*: di tipo *EventHistory* gestisce lo storico della connessione;
  + *orig\_pkts*: numero di pacchetti inviati dall’*host* di origine;
  + *orig\_ip\_bytes*: numero di bytes inviati dall’*host* di origine, dal punto di vista del cavo;
  + *resp\_pkts*: numero di pacchetti inviati dall’*host* rispondente;



Figura 12. Diagramma della classe *Event*

* + *resp\_ip\_bytes*: numero di bytes inviati dall’*host* rispondente, dal punto di vista del cavo;
  + *i discretizzatori degli attribute,* ovvero attributi statici che serviranno per effettuare la discretizzazione dei valori degli attributi continui : *duration, orig\_bytes, resp\_bytes, missed\_bytes, orig\_pkts, orig\_ip\_bytes, resp\_pkts, resp\_ip\_bytes.*

La classe contiene i metodi per la lettura dei valori degli attributi di *Event* e *EventHistory*.

* *EventHistory* (Figura 13) contiene gli attributi numerici, in versione continua e discretizzata, sia dal punto di vista dell’*host* di origine che del rispondente: *syn, fin*, *syn\_ack, rst, bad\_checksum, content\_gap, retransmitted\_payload, zero\_window, conn\_dir\_flipped.*

E i seguenti attributi discreti *boolean*: *ack, payload, inconsistent, multi\_flag.* La classe contiene i metodi per la lettura dei valori degli attributi e la conversione della stringa *history* del file *conn.log* nell’oggetto *EventHistory*.



Figura 13. Diagramma della classe *EventHistory*

Inoltre, sono stati definiti enumeratori che sono classi usate per raggruppare delle costanti che sono accomunate semanticamente, e vengono usate, per esempio, per definire i valori discreti di un determinato attributo. In questo caso sono stati creati i seguenti enumeratori (Figura 14):

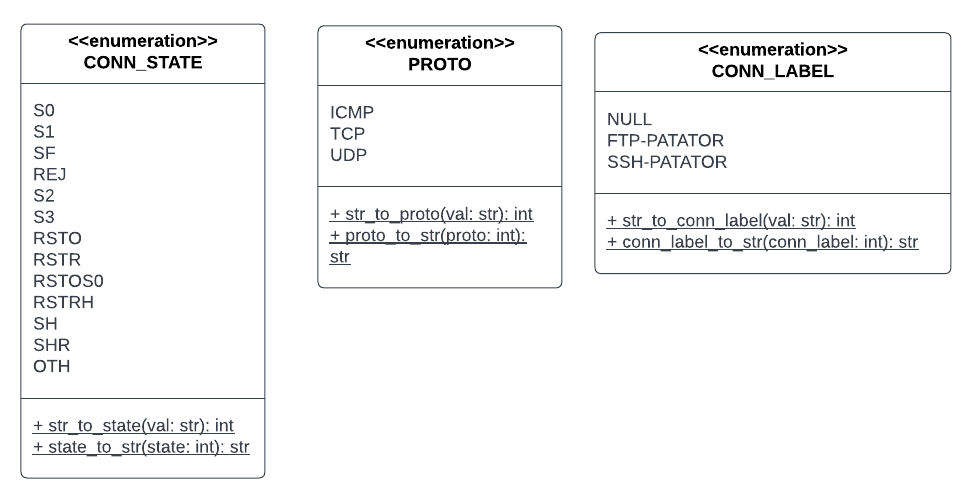


Figura 14. Diagramma degli Enumeratori *CONN\_STATE, PROTO* e *CONN\_LABEL*

* *CONN\_STATE* che assume i valori: *S0, S1, SF, REJ, S2, S3, RSTO, RSTR, RSTOS0, RSTRH, SH, SHR* e *OTH*;
* *PROTO* che assume i valori: *ICMP*, *TCP* e *UDP*;
* *CONN\_LABEL* che assume i valori *NORMAL* e *ANOMALY*.

L’operazione di *process discovery* accetta esclusivamente valori discreti, quindi è stato creato il package *DiscretizerModule* (Figura 15) che contiene tre classi, di cui la superclasse astratta *Discretizer* e le sottoclassi *EqualWidthDiscretizer* e *EqualFrequencyDiscretizer*, e l’enumeratore *DISCRETIZATION\_TYPE*. La discretizzazione consiste nel suddividere il dominio di valori per ogni attributo in intervalli distinti, e assegnare ad ogni intervallo una etichetta, che corrisponde al valore discreto dell’attributo. Per discretizzare un valore di un attributo, una volta che la discretizzazione è stata appresa, si deve trovare l’intervallo in cui ricade il valore e assegnare l’etichetta corrispondente all’intervallo trovato precedentemente.

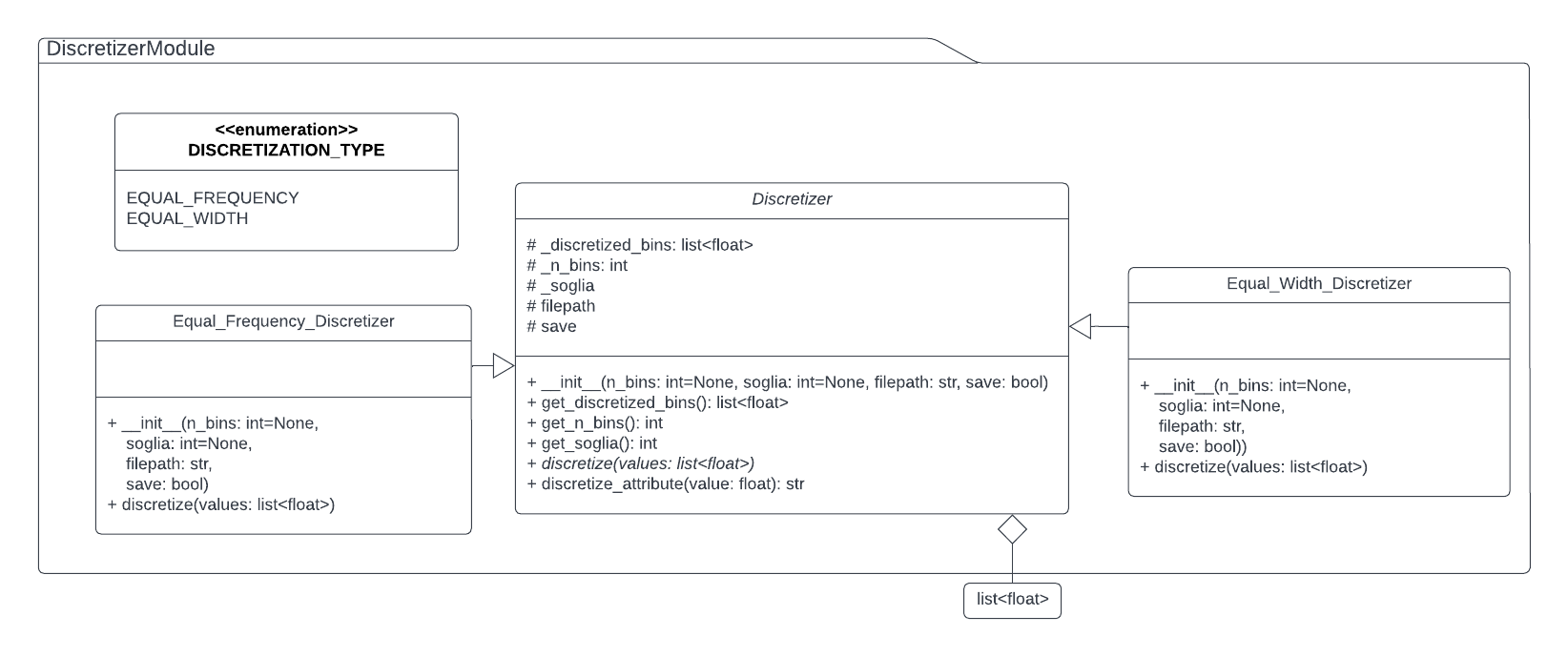


Figura 15. Diagramma delle classi del *package DiscretizerModule*.

La classe astratta *Discretizer* contiene i seguenti attributi:

* *discretized\_bins*: lista di cui ogni valore corrisponde ad un estremo di un intervallo, contenente valori, dove è il numero di intervalli da discretizzare;
* *n\_bins*: numero di intervalli della discretizzazione;
* *soglia*: valore per decidere se utilizzare l’algoritmo di discretizzazione normale o se applicare un altro algoritmo. Per applicare l’altro algoritmo si effettua il test:

se il confronto risulta vero, vengono generati degli intervalli utilizzando come estremi degli intervalli i valori medi tra l’-esimo valore dell’attributo e il successivo;

* *filepath*: percorso del file dove memorizzare o leggere gli intervalli di discretizzazione per essere utilizzati per più *conn.log*;
* *save*: valore *boolean* che determina se salvare o caricare gli intervalli di discretizzazione nel file identificato dall’attributo *filepath*.

Inoltre, implementa i metodi per poter leggere gli attributi *discretized\_bins, n\_bins* e *soglia*, e discretizzare un valore in input in base agli intervalli di discretizzazione. Inoltre, possiede la specifica del metodo astratto *discretize* che prende in input una lista di valori per cui generare gli intervalli di discretizzazione.

Le classi *EqualWidthDiscretizer* e *EqualFrequencyDiscretizer* implementano il metodo *discretize* in base ai rispettivi algoritmi:

* *EqualWidthDiscretizer* genera gli intervalli secondo il seguente algoritmo:
  + Si genera l’intervallo totale di valori ;
  + Si suddivide l’intervallo in parti uguali in base al numero di intervalli (*n\_bins*), dove gli estremi dell’-esimo intervallo sono definiti nel seguente modo:

.

* *EqualFrequencyDiscretizer* genera gli intervalli secondo la seguente implementazione:
  + Si ordinano i valori della lista di valori da discretizzare;
  + Per ogni estremo viene selezionato l’-esimo valore della lista ordinata.

L’enumeratore *DISCRETIZATION\_TYPE* ha due valori: EQUAL\_FREQUENCY e EQUAL\_WIDTH, ed è usato per indicare che tipo di discretizzazione effettuare in base al file *config.ini* in input.

Il *package* *EntryPoint* (Figura 16) contiene due classi *MainTraceController* e *MainMachineLearning*. La classe *MainTraceController* ha i metodi per realizzare le seguenti funzionalità:

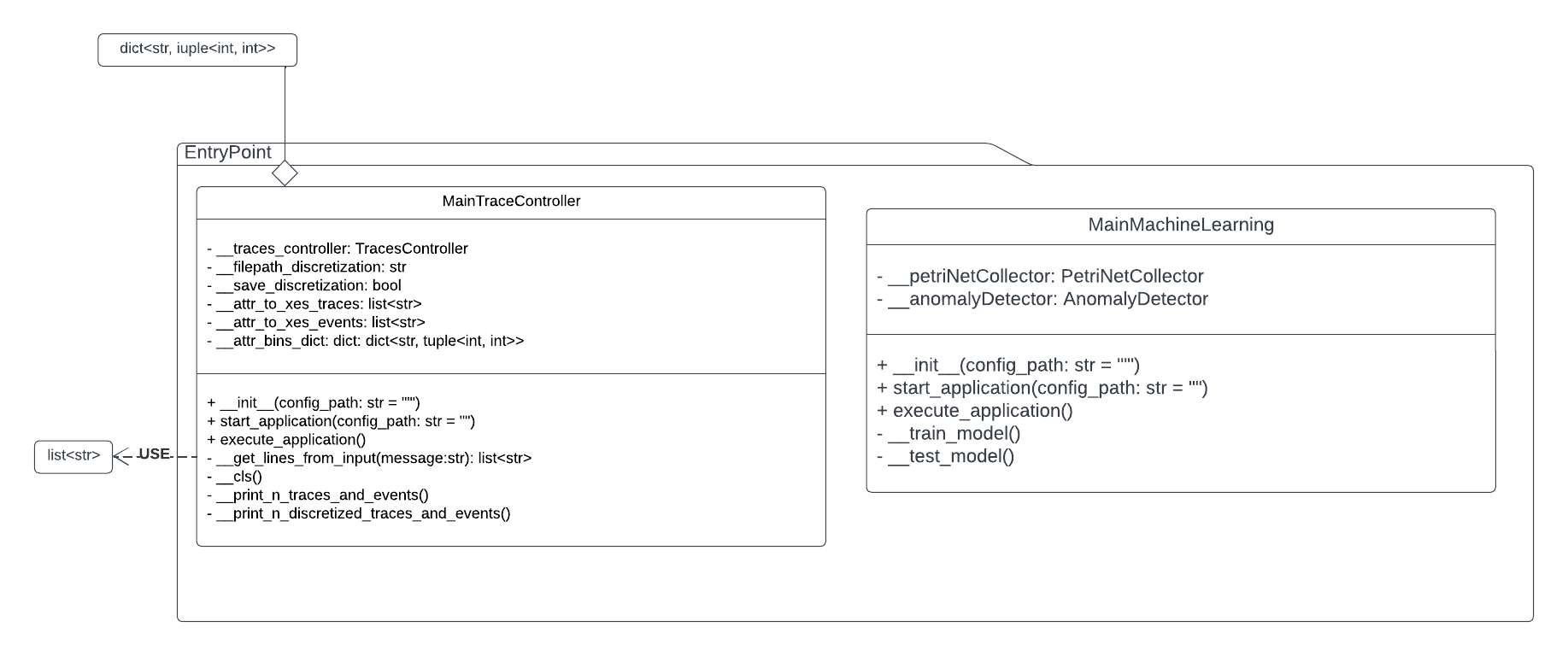


Figura 16. Diagramma delle classi del *package* *EntryPoint*

1. leggere i comandi di configurazione dal file *config.ini;*
2. istanziare un oggetto di classe *TraceController;*
3. eseguire le operazioni di lettura del file *conn.log;*
4. effettuare la conversione da *conn.log* in *traces*;
5. generare gli intervalli di discretizzazione;
6. applicare la discretizzazione agli attributi;
7. generare il file *xes*.

Inoltre, contiene gli attributi:

* *traces\_controller*: possiede l’oggetto *TraceController* per effettuare la generazione del file *xes*;
* *filepath\_discretization*: contiene il percorso del file dove salvare la discretizzazione appresa;
* *save\_discretization*: *flag* che indica se salvare (*true*) o caricare dalla memoria (*false*) le discretizzazioni apprese;
* *attr\_to\_xes\_trace*: lista di attributi da inserire nei *trace* dello xes
* *attr\_to\_xes\_events*: lista di attributi da inserire negli *events* dello xes
* *attr\_bins\_dict*: dizionario che ha per chiave l’attributo e per valore una coppia il cui primo valore è il numero di intervalli di discretizzazione, e il secondo è la soglia.

*MainMachineLearning*, invece, contiene gli attributi:

* *petriNetCollector*: contiene un oggetto di classe *PetriNetCollector* per la gestione delle varie *PetriNets*;
* *anomalyDetector*: contiene un oggetto di classe *AnomalyDetector* per effettuare le predizioni, la validazione e la generazione del report dei risultati del *test*.

i metodi per realizzare le seguenti operazioni:

1. leggere i comandi di configurazione dal file *config.ini;*
2. eseguire il *training* delle *PetriNets;*
3. effettuare il *test*;
4. generare un *report* dei risultati di sperimentazione.

L’applicazione necessita di un file *config.ini* (Figura 17) (se non presente o non contiene tutti i dati, verranno utilizzati dei dati di default), con delle impostazioni personalizzabili per il corretto funzionamento dell’applicazione. Il file deve contenere i seguenti campi:

Immagine che contiene testo

Descrizione generata automaticamente

Figura 17. Esempio di file *config.ini* per l’applicazione

* *path\_of\_file\_input*: percorso del file *log* da cui generare gli *xes*;
* *path\_of\_file\_xes*: percorso dove memorizzare il file *xes* da generare;
* *path\_of\_file\_xes\_train*: percorso da cui caricare lo *xes* per l’apprendimento delle *PetriNet*;
* *path\_of\_file\_xes\_test*: percorso da cui caricare lo *xes* per effettuare il *testing* con l’*AnomalyDetector*;
* *path\_pf\_petriNet\_models\_train*: percorso nel quale salvare i modelli delle *PetriNets* addestrate;
* *path\_of\_petriNet\_models\_dataset\_test*: percorso nel quale salvare i *dataset* generati con i livelli di *fitness* delle petriNet;
* *path\_of\_report\_file*: percorso nel quale salvare il *report* generato;
* *strings\_to\_filter\_event*: elenco delle stringhe da ignorare dai *log* prima di generare lo *xes*;
* *discretization\_type*: tipo di discretizzazione da effettuare (*equal\_width* o *equal\_frequency*);
* *n\_bins*: numero di intervalli di discretizzazione;
* *soglia*: valore di soglia per la discretizzazione;
* *filepath*: percorso dove memorizzare gli intervalli di discretizzazione addestrati;
* *save*: *flag* per indicare se salvare (1) o caricare (0) gli intervalli di discretizzazione appresi;
* *attributes\_to\_xes\_traces*: elenco degli attributi da inserire nei *trace* dello *xes* da generare;
* *attributes\_to\_xes\_traces*: elenco degli attributi da inserire negli *event* dello *xes* da generare;
* *show\_examples*: *flag* per determinare se mostrare (1) o no (0) degli esempi di *trace* ed *eventi* e esempi di discretizzazione (usato per *debugging*);
* *n\_trace\_to\_print*: numero di *trace* da mostrare se *show\_examples* è impostato a 1;
* *randomize\_print*: *flag* che indica se mostrare *trace* selezionati casualmente (1) o i primi *n\_trace\_to\_print*;
* *delta*: valore che indica la percentuale di *trace outliers* negli *xes*.

3.3 Reti di Petri

Le fasi di *process discovery* e *anomaly detection* sono modellate nel *package* *MachineLearning* (Figura 18) che contiene tre classi:

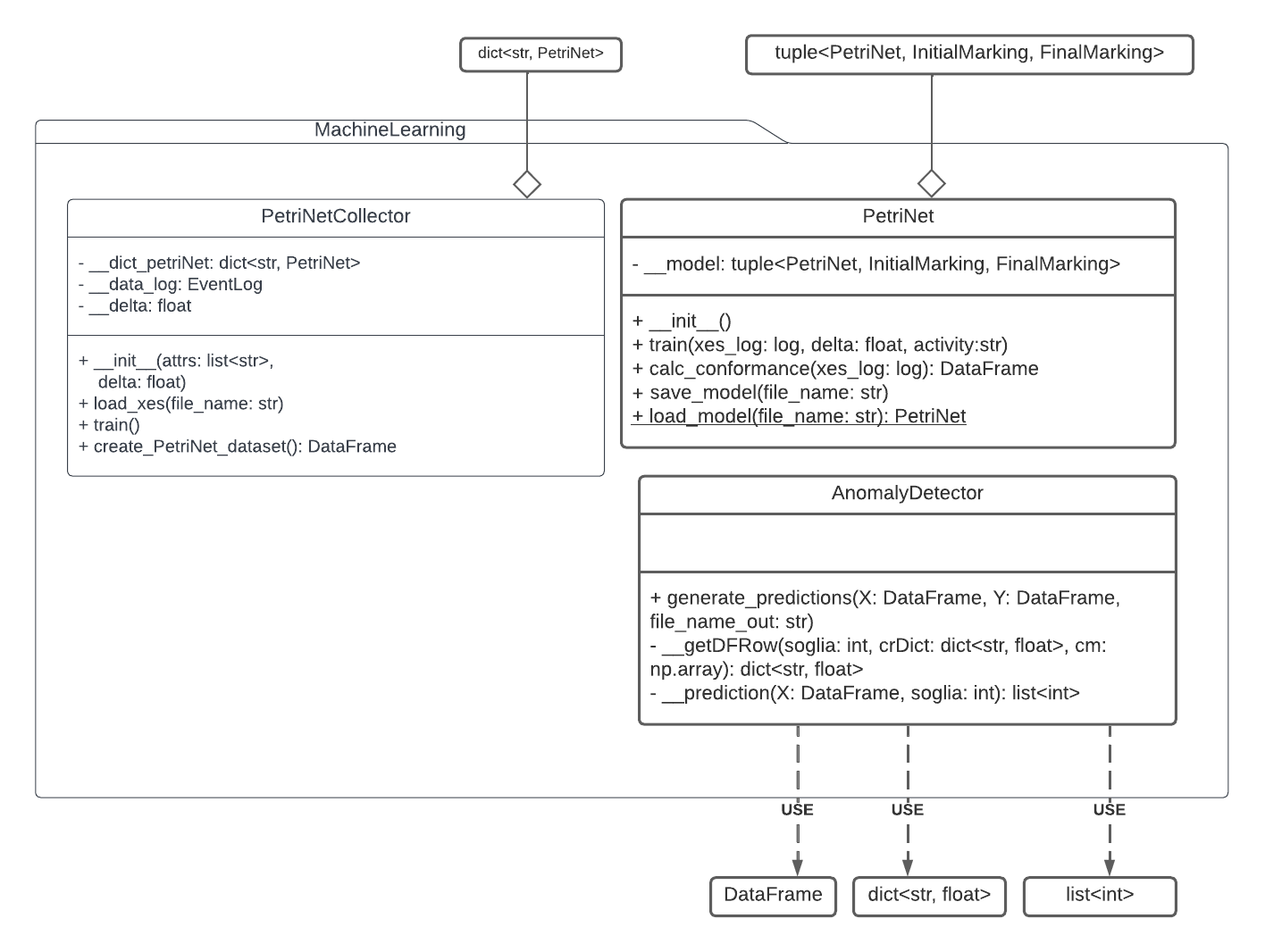


Figura 18. Diagramma delle classi del *package* *MachineLearning*

* *PetriNet* contiene l’attributo model che rappresenta una tupla contenente il modello di una PetriNet addestrata dai trace memorizzati in un file *xes*, e come metodi: il costruttore, il metodo per effettuare l’addestramento, il metodo per il calcolo della conformance di un altro *xes* e i metodi per la serializzazione e deserializzazione del modello;
* *PetriNetCollector* ha lo scopo di contenere tutte le *PetriNets* (una PetriNet per attributo di Event) da addestrare e con cui calcolare i livelli di conformance. La classe include gli attributi:
  + dict\_PetriNet: il dizionario delle *PetriNet* la cui chiave è l’attributo su cui è addestrata e come valore un oggetto di classe PetriNet;
  + *data\_log*: l’*EventLog* attualmente in uso;
  + delta: il parametro che indica la percentuale di *outliers* nello *xes.*

Tale classe include anche i seguenti metodi:

* + costruttore con cui inizializza tutte le PetriNet;
  + *load\_xes*: un metodo per poter effettuare il caricamento del file *xes* da assegnare all’attributo *data\_log*;
  + *train*: il metodo per effettuare l’addestramento delle PetriNet
  + *create\_PetriNet\_dataset*: che genera un dataset concatenando i livelli di conformance delle PetriNet.
* *AnomalyDetector* ha lo scopo di rilevare le anomalie basandosi sul numero di attributi per cui il livello di conformance risulta essere diverso da *1* (conformance massima). La classe include i seguenti metodi:
  + *generate\_predictions*: genera le predizioni ed effettua un report le cui colonne sono: soglia, TP, FN, FP, TN, precision (positive), recall (positive), fscore (positive), precision (negative), recall (negative), fscore (negative), accuracy, macroF, WeigthedF;
  + *getDFRow*: data la soglia, il dizionario derivante dal classification report e la matrice di confusione, restituisce una riga per il metodo *generate\_predictions*;
  + *prediction*: dato il dataset e la soglia (il numero massimo di attributi per cui il livello di conformance è diverso da 1) predice per ogni esempio se è anomalo o meno.

La fase di *process discovery* genera una rete di Petri. Una rete di *Petri* è una delle varie rappresentazioni matematiche di un sistema distribuito discreto. Come un linguaggio di modellazione, esso descrive la struttura di un sistema distribuito come un grafo bipartito con delle annotazioni. Una rete di *Petri* *PT* (*Place*/*Transition*) è un grafo orientato con due tipi di nodi, posti e transizioni, connessi da archi diretti. I posti sono rappresentati graficamente da cerchi e le transizioni da rettangoli. Un arco può unire solamente nodi di tipo diverso, quindi possono esserci archi solo tra posti e transizioni. Un posto da cui un arco parte per finire in una transizione è detto posto di input della transizione; un posto in cui un arco arriva partendo da una transizione è detto posto di output della transizione. I posti possono contenere un certo numero di token o marche. Una distribuzione di token sull'insieme dei posti della rete è detta marcatura. Le transizioni agiscono sui token in ingresso secondo una regola, detta regola di scatto (in inglese *firing*). Una transizione è abilitata se può scattare, cioè se ci sono token in ogni posto di input. Quando una transizione scatta, essa consuma token dai suoi posti di input, esegue dei task e posiziona un numero specificato di token in ognuno dei suoi posti di uscita.[20] La *PetriNet* utilizzata in questa tesi viene generata dalla libreria *pm4py* in *Python*.

3.4 *Fitness* di un *Trace*

Nel *process mining*, dopo che è stato addestrato un modello, si può determinare il livello di *fitness* di un ulteriore *trace* con il modello. Uno degli algoritmi per determinare il livello di *fitness* è tramite l’*Alignment based Fitness*. Un modello è “conforme” ad un *log* se tutti i *trace* all’interno del *log* possono essere riprodotti dal modello. Una metrica di *fitness* quantifica la conformità di un log in un modello.[21] Si può determinare, quindi, se un trace risulta essere normale o anomalo se il livello di *fitness* di questo *trace* con il modello addestrato su degli *xes* con *trace* normali risulta essere rispettivamente o compreso in .

4. Validazione empirica

4.1 CICD 2017

La sperimentazione è stata effettuata tramite il dataset di file “pcap” generati dal *Canadian Institute for Cybersecurity* (CIC-IDS2017) del 2017.[22] Il dataset è composto da cinque file *pcap*, i cui pacchetti sono stati catturati nelle giornate lavorative da lunedì, 3 Luglio 2017 al venerdì, 7 Luglio 2017. Il lunedì non sono stati effettuati attacchi, quindi si ha un file *pcap* con pacchetti normali, invece, dal martedì al venerdì sono stati effettuati attacchi di tipo diverso nel seguente modo:

* martedì: FTP-Patator (9:20 – 10:20 a.m.) e SSH-Patator (14:00 – 15:00 p.m.;
* mercoledì: DoS slowloris (9:47 – 10:10 a.m.), DoS Slowhttptest (10:14 – 10:35 a.m.), DoS Hulk (10:43 – 11 a.m.), DoS GoldenEye (11:10 – 11:23 a.m.) e Heartbleed Port 444 (15:12 - 15:32);
* giovedì: Web Attack – Brute Force (9:20 – 10 a.m.), Web Attack – XSS (10:15 – 10:35 a.m.), Web Attack – Sql Injection (10:40 – 10:42 a.m.), Meta exploit Win Vista (14:19 -14:35), Infiltration – Cool disk – MAC (14:53 p.m. – 15:00 p.m.) e Infiltration – Dropbox download - Win Vista (15:04 – 15:45 p.m.);
* venerdì: Botnet ARES (10:02 a.m. – 11:02 a.m.), Port Scan Firewall Rule on (13:55 - 14:35), Firewall rules off (14:51-15:29) e DDoS LOIT (15:56 – 16:16).

Ad ogni fascia oraria, e ad ogni tipo di attacco è associata una coppia indirizzo *IP* attaccante e indirizzo *IP* attaccato, in questo modo è possibile distinguere le connessioni di rilevanza per gli attacchi da quelle di altri *host* di connessioni normali.

4.2 Risultati dell’estrazione dei *trace*

L’estrazione dei *trace* ha avuto i risultati descritti nella tabella 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Giorno | Numero di *trace* normali | Numero di *trace* anomali | Lunghezza media dei *trace* normali | Lunghezza media dei *trace* anomali |
| Lunedì | 249242 | 0 | 1.5 | N/A |
| martedì | 204901 | 6462 | 1.54 | 1 |
| mercoledì | 212678 | 14117 | 1.5 | 13.48 |
| Giovedì | 173874 | 69550 | 1.52 | 1.13 |
| venerdì | 190523 | 205890 | 1.5 | 1.25 |

Tabella 1. Risultati dell’estrazione dei *trace* in termini di numero di *trace* per classe e lunghezza media dei *trace* per ogni classe

4.3 Etichettatura dei *trace*

L’etichettatura dei *trace* è stata possibile mediante uno *script* che segue le regole indicate sul sito e riportate al capitolo 4.1. L’etichettatura in questa tesi è consistita nel generare un ulteriore file *conn\_labeled.log*, a cui è stata aggiunta la colonna *label* che risulta essere *Normal* o *Anomaly*. Seguendo le specifiche dei creatori del dataset, in alcune fasce orarie non risultano esserci attacchi del tipo descritto. Non potendo sapere se gli attacchi sono stati effettuati in orari diversi rispetto a quelli elencati, o sfasati di qualche ora, è stata presa la decisione di rimuovere dal dataset le connessioni originate dagli *host* coinvolti negli attacchi che vengono rilevate al di fuori degli orari specificati.

4.4 Metriche di valutazione e setting sperimentale

La sperimentazione è suddivisa in *training* e *testing*. La fase di apprendimento è stata effettuata utilizzando solo lo *xes* della giornata di lunedì, poiché è la giornata dove non sono presenti connessioni anomale. Per il test sono stati generati i file *xes* delle giornate dal martedì al venerdì, e sono stati effettuati quattro *test* diversi, il cui addestramento utilizzato è quello di lunedì per tutti le validazioni, reso possibile caricando il modello addestrato dopo la prima validazione. La sperimentazioneè stata impostata in questo modo:

1. si generano i file *xes* dai *conn\_labeled.log* per ogni giornata;
2. viene letto il file *xes* del lunedì per generare il modello della PetriNet e caricare in memoria le *label* dei *trace*;
3. per ogni attributo degli *Event*:
   1. se è presente su disco il modello della PetriNet addestrata su quell’attributo come *Activity* allora viene caricata;
   2. altrimenti viene addestrato un modello con quell’attributo come *Activity* e successivamente viene salvato su disco per futuri usi;
4. viene letto il file *xes* di uno dei giorni dal martedì al venerdì per generare il *dataset* per *l’Anomaly Detector*;
5. Viene generato il dataset che è un *DataFrame* della libreria di *Python pandas*, le cui colonne sono gli attributi usati come activity e i corrispondenti valori di *fitness* calcolati, e le righe sono i *trace* per cui sono stati calcolati i livelli di *fitness* per ogni attributo;
6. Viene dato in input il *dataset* all’*Anomaly Detector* il quale genera il report. Il report è riferito ad una giornata ed è una tabella dove le colonne sono le metriche di valutazione e le righe sono i livelli di soglia utilizzati per determinare se i *trace* sono anomali o meno.

I risultati della sperimentazione devono essere valutati secondo delle metriche di valutazione standard. La metrica utilizzata per il livello di *fitness* degli *xes* è stata l’*Alignment based fitness,* mentre quelle usate per la valutazione[23] dell’*Anomaly Detector* sono:

* TP: *true positive,* il numero di esempi normali, che sono stati predetti come normali;
* FN: *false negative,* il numero di esempi normali, che sono stati predetti come anomali;
* FP: *false positive,* il numero di esempi anomali, che sono stati predetti come normali;
* TN: *true negative,* il numero di esempi anomali, che sono stati predetti come anomali;
* precision (positive): il rapporto tra il numero degli esempi predetti correttamente come normali sul totale degli esempi predetti normali
* recall (positive): il rapporto tra il numero degli esempi predetti correttamente come normali sul totale degli esempi effettivamente normali
* fscore (positive): misura che tiene conto della precision (positive) e recall (positive), calcolando la media armonica di precisione e recupero;
* precision (negative): il rapporto tra il numero degli esempi predetti correttamente come anomali sul totale degli esempi predetti anomali;
* recall (negative): il rapporto tra il numero degli esempi predetti correttamente come anomali sul totale degli esempi effettivamente anomali;
* fscore (negative): misura che tiene conto della precision (negative) e recall (negative), calcolando la media armonica di precisione e recupero[24];
* accuracy: rapporto tra il numero degli esempi predetti correttamente sul numero degli esempi;
* macroF: media aritmetica dei valori di f-score positivi e negativi[25];
* WeigthedF: media ponderata dei valori di f-score positivi e negativi, pesati considerando il supporto di ogni classe[26].

4.5 Risultati

Bibliografia

[1] - Wil M. P. van der Aalst: Process Mining - Data Science in Action, Second Edition. Springer 2016, ISBN 978-3-662-49850-7, pp. 3-452

[2] – Wireshark – Strumento per l’analisi di pacchetti - [Wireshark · Go Deep.](https://www.wireshark.org/)

[3] – Zeek – Strumento di monitoraggio della sicurezza di rete - [The Zeek Network Security Monitor](https://zeek.org/)

[4] – Scope – Internet Protocol – Darpa Internet Program Protocol Specification - Information Sciences Institute University of Southern California [RFC 791: Internet Protocol (rfc-editor.org)](https://www.rfc-editor.org/rfc/rfc791#section-1.2)

[5] – UDP – RFC 768 - User Datagram Protocol - [RFC 768: User Datagram Protocol (rfc-editor.org)](https://www.rfc-editor.org/rfc/rfc768)

[6] – Scope – Transmission Control Protocol - Darpa Internet Program Protocol Specification - Information Sciences Institute University of Southern California [RFC 793: Transmission Control Protocol (rfc-editor.org)](https://www.rfc-editor.org/rfc/rfc793.html#section-1.2)

[7] – Format - UDP – RFC 768 - User Datagram Protocol - [RFC 768: User Datagram Protocol (rfc-editor.org)](https://www.rfc-editor.org/rfc/rfc768)

[8] – Header Format – Transmission Control Protocol - Darpa Internet Program Protocol Specification - Information Sciences Institute University of Southern California [RFC 793: Transmission Control Protocol (rfc-editor.org)](https://www.rfc-editor.org/rfc/rfc793.html#section-1.2)

[9] – HTTP – Computer science – Britannica - [HTTP | Definition, Meaning, Versions, & Facts | Britannica](https://www.britannica.com/technology/HTTP)

[10] - I CERTIFICATI SSL e i nuovi obblighi del GDPR - [I CERTIFICATI SSL e i nuovi obblighi del GDPR Maggio 2018 I certificati SSL e i siti con protocollo https (nethics.it)](https://www.nethics.it/i-certificati-ssl-e-i-nuovi-obblighi-del-gdpr/)

[11] - Ports for Internet Services - Any Port in a Datastorm - Richard Akerman - [TCP/IP Ports (chebucto.ns.ca)](http://www.chebucto.ns.ca/~rakerman/port-table.html)

[12] – DESCRIPTION – PCAP(3PCAP) MAN PAGE - [pcap(3PCAP) man page | TCPDUMP & LIBPCAP](https://www.tcpdump.org/manpages/pcap.3pcap.html)

[13] – Definition of the CSV Format – Common Format and MIME Type for Comma-Separated Values (CSV) Files - Network Working Group - Y. Shafranovich - [RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files (rfc-editor.org)](https://www.rfc-editor.org/rfc/rfc4180#section-2)

[14] – conn.log – Book of Zeek - [conn.log — Book of Zeek (git/master)](https://docs.zeek.org/en/master/logs/conn.html)

[15] – dns.log – Book of Zeek - [dns.log — Book of Zeek (git/master)](https://docs.zeek.org/en/master/logs/dns.html)

[16] – Attributi del tipo Conn::Info – Zeek Docs - [Conn::Info Zeek Type](https://docs.zeek.org/en/master/scripts/base/protocols/conn/main.zeek.html?highlight=conn%20info%23type-Conn::Info)

[17] – Python – Welcome to Python - [Welcome to Python.org](https://www.python.org/)

[18] – 2.2 Process Mining Pag. 32 - Wil M. P. van der Aalst: Process Mining - Data Science in Action, Second Edition.

[19] – An Introduction to the XES Standard – Fluxicon - Christian W. Günther - [An Introduction to the XES Standard — Flux Capacitor (fluxicon.com)](https://fluxicon.com/blog/2010/09/intro-to-xes/)

[20] – Petri net – Scolarpedia - [Petri net - Scholarpedia](http://www.scholarpedia.org/article/Petri_net)

[21] - Quality Metrics in Process Mining - Alignment-based Metrics in Conformance Checking - B.F. van Dongen and J. Carmona and T. Chatain - [paper22.pdf (ceur-ws.org)](https://ceur-ws.org/Vol-1701/paper22.pdf)

[22] - Intrusion Detection Evaluation Dataset (CIC-IDS2017) – UNB - [IDS 2017 | Datasets | Research | Canadian Institute for Cybersecurity | UNB](https://www.unb.ca/cic/datasets/ids-2017.html)

[23] – Types of Errors – Artificial Intelligence - Foundations of Computational Agents – David L. Poole, Alan K. Mackworth ISBN – 9781107195394 pp. 307 – 309

[24] – F1 score – Wikipedia - [F1 score - Wikipedia](https://it.wikipedia.org/wiki/F1_score)

[25] - Macro Average - Micro, Macro & Weighted Averages of F1 Score, Clearly Explained - Kenneth Leung - [Micro, Macro & Weighted Averages of F1 Score, Clearly Explained | by Kenneth Leung | Towards Data Science](https://towardsdatascience.com/micro-macro-weighted-averages-of-f1-score-clearly-explained-b603420b292f#989c)

[26] - Weighted Average- Micro, Macro & Weighted Averages of F1 Score, Clearly Explained - Kenneth Leung - [Micro, Macro & Weighted Averages of F1 Score, Clearly Explained | by Kenneth Leung | Towards Data Science](https://towardsdatascience.com/micro-macro-weighted-averages-of-f1-score-clearly-explained-b603420b292f#989c)