Università degli Studi di Modena e Reggio Emilia

Facoltà di Ingegneria “Enzo Ferrari”
Tesi di Laurea Specialistica in Ingegneria Informatica

Machine learning algorithms for clustering and correlating security alerts in Intrusion Detection Systems

-

Algoritmi di machine learning per il clustering e la correlazione di alerts in Intrusion Detection Systems

Relatore:
Prof. Michele Colajanni

Correlatore:
Ing. Mirco Marchetti

Tesi di Laurea di
Fabio Manganiello

Anno Accademico 2009-2010

Contents

Italian abstract

Il mondo attuale della sicurezza informatica fa grande uso di strumenti quali gli *Intrusion Detection Systems* (*IDS*), un tipo di software il cui obiettivo è quello di analizzare flussi di traffico alla ricerca di eventuali pattern riconducibili ad attacchi noti o comunque traffico sospetto. Un IDS è generalmente considerato come una seconda linea di difesa contro attacchi informatici, dopo firewall, software anti-malware e l'implementazione di politiche di rete efficaci, in presenza della necessità di provvedimenti nei confronti di traffico potenzialmente dannoso.

In ogni caso, un IDS si concentra su anomalie e attacchi a "basso livello", analizzando i possibili allarmi come oggetti indipendenti l'uno dall'altro e ignorando eventuali collegamenti logici fra alcune coppie. Il risultato di un IDS è quindi un log, analizzabile da un amministratore di sistema, che contiene i dettagli su degli attacchi elementari potenzialmente dannosi, ignorando possibili correlazioni fra questi.

Obiettivi del software

Alcuni malintenzionati potrebbero effettuare attacchi relativamente complessi prima di raggiungere un determinato obiettivo, e le correlazioni fra i diversi passi in questi attacchi non verranno trovate da un IDS classico. Ciò significa che un analista di sistema impiegherà generalmente molto tempo in due attività che potrebbero essere evitate se l'IDS stesso avesse abbastanza conoscenza e "logica" a monte per non trattare gli allarmi come eventi a sé stanti:

- Eliminazione manuale della ridondanza tipica degli alert log (ad esempio un gran numero di "consecutive TCP small segments exceeding threshold" se una macchina nella rete analizzata ha un'implementazione difettosa dello stack TCP/IP), e setacciare gli allarmi che "inquinano" i log con la loro ridondanza considerando che, come che illustrato in [6], poche cause "radici" permanenti sollevano in genere il 90% degli allarmi in un IDS, e queste cause devono essere identificate per semplificare l'analisi dei log;

- Trovare correlazioni fra allarmi riportati dall'IDS senza investigare sulle possibili relazioni reciproche, e identificare quelli effettivamente indipendenti.

Queste attività sono costose in termini di tempo per un amministratore di sistema, e per poter ridurre il tempo richiesto in questi compiti diversi sforzi sono stati compiuti negli ultimi anni, generalmente classificabili all'interno di due principali categorie di approcci:

- Riduzione del numero di allarmi riportati trovando e raggruppando allarmi marcati come "simili" sulla base di un insieme configurabile di criteri e regole (*alert clustering*);

- Correlazione di allarmi probabilmente appartenenti allo stesso scenario di attacco complesso, o aventi relazione di causalità, per poter generare il grafico di un attacco complesso lanciato contro un determinato sistema o una rete (*alert correlation*).

Il software sviluppato e discusso in questo documento può essere classificato all'interno di entrambe le categorie, poiché prima tenta di trovare cluster di allarmi sulla base di un insieme di criteri di raggruppamento configurabili, quindi costruisce i grafi di correlazione che esprimono le relazioni di causalità fra questi insiemi di allarmi. I risultati di queste due operazioni vengono mostrati attraverso un'interfaccia web intuitiva, basata su un web server sviluppato da zero per questo scopo, o attraverso la visualizzazione diretta dei grafi di correlazione da immagini PNG o file PostScript.

Per poter effettuare operazioni di clustering e correlazione su un certo insieme di allarmi sollevati da un IDS, si useranno algoritmi di *machine learning* e intelligenza artificiale. È infatti ampiamente discusso in [10] di come un IDS sia un software praticamente incompleto senza l'applicazione a valle di un insieme di algoritmi di intelligenza artificiale per l'estrazione di nuove informazioni (potenziali cluster o correlazioni) a partire dalla conoscenza acquisita dalle minacce registrate. Infatti, senza questi algoritmi un IDS solleverebbe allarmi associati ad attacchi elementari senza nessuna ulteriore conoscenza sul contesto in cui questi attacchi sono stati effettuati. In particolare, questo software si concentra particolarmente sui seguenti aspetti:

1. Analisi del traffico di rete e parsing degli allarmi sollevati dall'IDS, col fine di associare, se possibile, un allarme al flusso TCP che lo ha generato;

2. Analisi dei gruppi di allarmi simili e mutuo raggruppamento di questi ultimi in cluster omogenei, al fine di:

   (a) Ridurre la ridondanza nei log dell'IDS, semplificando quindi il compito di analisi all'amministratore di sistema;

   (b) Identificare semplicemente possibili cause comuni dietro gli allarmi più diffusi in una certa rete;

   (c) Effettuare un primo tentativo di riconoscere semplici scenari di attacco all'interno dello stesso cluster.

   Un algoritmo di clustering gerarchico è stato usato in questo software, basato su tassonomie di clustering configurabili in base al contesto;

3. Trovare correlazioni fra insiemi di allarmi registrati. Per fare questo, il software usa quattro potenziali indici di correlazione per le coppie di allarmi:

   (a) Un indice di correlazione basato su una *knowledge base*, un modello (*hyperalert*) che descrive generici prerequisiti e conseguenze per un certo tipo di attacco, e può quindi essere usato per costruire grafi di causalità fra gli allarmi;

   (b) Un indice di correlazione *pseudo*-bayesiano, che correla coppie di allarmi considerando la probabilità che capitino nella stessa finestra temporale basandosi sulla conoscenza già acquisita dei possibili scenari;

(c) Un indice di correlazione basato su reti neurali, che calcola la similitudine fra due alert, dopo una fase di addestramento, e li riporta su una mappa bidimensionale (e "clusterizzabile"), trasformando quindi il problema della similarità fra alert in un problema geometrico di distanze fra punti su uno spazio bidimensionale e i loro raggruppamenti più probabili;

(d) Eventuali correlazioni manuali esplicite, fornite dall'utente nel caso in cui il sistema non riconoscesse una correlazione effettiva fra due allarmi, o marcasse come vera una correlazione non esistente.

Al di là di questi indici di correlazione, il software è progettato per essere un framework modulare, rendendo quindi possibile l'aggiunta di ulteriori indici di correlazione attraverso un'interfaccia di programmazione estremamente semplice. L'indice di correlazione complessivo fra due allarmi è calcolato come media pesata di tutti gli indici di correlazione ad esso associati;

4. Provvedere un'interfaccia utente intuitiva per esplorare i grafi di cluster di allarmi, specificare nuove correlazioni, o analizzare il traffico di rete associato a un allarme specifico.

L'obiettivo è quello di estendere e costruire un sistema al di sopra di un IDS, senza impattare sulle sue prestazioni, che semplifichi l'analisi degli allarmi per un analista di sistema trovando automaticamente allarmi simili, riducendo la ridondanza dei log di allarmi, tentando di riconoscere possibili scenari di attacco, e che fornisca un'interfaccia utente intuitiva per esplorare tali risultati senza analizzare manualmente file di testo o output di database, e avente il minor impatto possibile sulle prestazioni dell'IDS.

Sono stati raggiunti diversi obiettivi nella progettazione e nello sviluppo di questo software, se confrontati all'attuale stato dell'arte. Tali miglioramenti verranno brevemente analizzati di seguito.

Miglioramenti algoritmici

Il metodo di clustering degli alert proposto da [6] è stato notevolmente migliorato attraverso:

- Un meccanismo a finestra temporale che evita di raggruppare allarmi sollevati in momenti distanti;
- Una strategia euristica per calcolare la dimensione ottimale dei cluster in base all'eterogeneità degli allarmi, evitando di specificare tale parametro in funzione del tipo di traffico da analizzare.

Gli indici di correlazione fra coppie di alert sono calcolati come medie pesate fra diversi indici di correlazione distinti, sia semi-deterministici che completamente euristici, riuscendo nell'attività di combinare indici associati a diverse strategie di correlazione. Le correlazioni euristiche sono inoltre moltiplicate per un insieme di pesi dinamici nel calcolo della media, calcolati in funzione del traffico di rete acquisito e della conoscenza attuale sugli allarmi sollevati.

Un algoritmo *pseudo*-bayesiano completamente nuovo (nel senso che "ricorda" una correlazione bayesiana, calcolando la probabilità condizionata di un evento sapendo che si è verificato un altro tipo di evento, ma la strategia usata per calcolare questa probabilità non passa per il teorema di Bayes) è anche stato proposto e testato con successo per calcolare la correlazione fra alert basata su vincoli temporali e una funzione di correlazione a decadimento esponenziale (gaussiana) è usata come “peso” numerico in funzione del tempo trascorso fra due alert.

È stata anche utilizzata una *Self-Organizing Map* (*SOM*) per il calcolo dell'indice di correlazione, come discusso in [8] e in altri documenti. Tuttavia le performance e la precisione della rete sono stati notevolmente migliorati grazie a un algoritmo “intelligente” per l'inizializzazione dei pesi, e per la prima volta una strategia di clustering sul layer di output della rete neurale (con una stima euristica sul numero di cluster) è stata usata per identificare i gruppi di allarmi probabilmente appartenenti allo stesso scenario di attacco.

Un linguaggio modulare e personalizzabile basato su XML è anche stato sviluppato e testato con successo per definire tipi di *hyperalert* da usare come base di conoscenza per gli indici di correlazione. Il linguaggio eredita i lati positivi di linguaggi come *CAML* (completamente modulare, espressivo, estensibile, con supporto per macro e variabili definite dall'utente) ed è pulito, facilmente leggibile, e analizzabile da ogni parser XML.

Miglioramenti software

Per la prima volta un motore di clustering e correlazione è sviluppato direttamente al di sopra di Snort, l'Intrusion Detection System più diffuso, come modulo per il suo preprocessore. Il modulo sviluppato è altamente personalizzabile attraverso il file `snort.conf` e raggruppa diverse tecniche di clustering e correlazione di allarmi, fornendo anche una semplice interfaccia web per navigare, modificare e analizzare cluster e grafi di correlazione di allarmi.

Il modulo sviluppato è a sua volta modulare, in quanto dei nuovi indici di correlazione possono essere facilmente aggiunti al motore attraverso un'interfaccia di programmazione intuitiva. Può quindi essere considerato come un framework per l'analisi di security alerts.

Il sistema supporta anche la correlazione manuale, specificabile attraverso un semplice meccanismo in stile “drag'n'drop” attraverso l'interfaccia web. Un linguaggio basato su XML è anche stato proposto per modellare queste correlazioni.

Il software è stabile, anche quando analizza grandi quantità di traffico, e non ha un grande impatto sulle performance dell'IDS.

I risultati ottenuti, sia per il clustering che per la correlazione di alert, sono stati soddisfacenti. Tuttavia un buon modulo per gli hyperalert, specialmente quando gli altri moduli di correlazione euristici non sono ancora sufficientemente “allenati”, è necessario per avere una correlazione più accurata, e anche un buon valore per il parametro di soglia di correlazione nella configurazione, che dipenda dal traffico da analizzare e dalla topologia della rete. Inoltre, una conoscenza incompleta riguardo ad alcuni modelli associati ad allarmi sollevati, unita a un'informazione “storica” su questi allarmi poco dettagliata o non completamente aderente ai risultati attesi, può portare a casi di correlazioni errate (falsi positivi) o correlazioni effettive non trovate (falsi negativi). Questo problema può essere parzialmente arginato usando l'interfaccia web per specificare manualmente correlazioni fra allarmi.

Sviluppi futuri

Diverse funzionalità sono state identificate come potenziali sviluppi futuri del software. Fra queste:

- Migliorare la knowledge base degli hyperalert. Finora nella directory `corr_rules` sono contenuti circa 20 modelli di hyperalert. Questo numero può essere espanso in futuro specificando ulteriori modelli, essendo la precisione nell'identificazione di correlazioni fra alert strettamente dipendente dal numero e dall'accuratezza dei modelli forniti, specialmente in scenari di conoscenza storica insufficiente per produrre dati significativi attraverso gli altri indici di correlazione. È anche doveroso ricordare che la precisione nell'identificazione di correlazioni fra alert dipende anche dalla precisione dell'*IDS* nell'identificare correttamente gli attacchi, minimizzando la quantità di falsi positivi e falsi negativi, poiché il modulo usa esattamente queste informazioni per generare i grafi di correlazione;

- Fornire ulteriori indici di correlazione, per correlare un insieme di alert usando più punti di vista e riducendo quindi la possibilità di errori di correlazione. Un approccio interessante è quello discusso in [2], che sfrutta una rete bayesiana *naive* con finestra temporale senza richiedere alcuna conoscenza al di là dei possibili obiettivi finali di un attaccante per correlare gli alert. Questo algoritmo può essere implementato come modulo aggiuntivo per il software;

- Trovare un algoritmo che calcoli il miglior valore di k nell'equazione 3.10. Questo parametro è attualmente settato in modo statico dall'utente in funzione del tipo di traffico, degli allarmi generalmente sollevati e della loro eterogeneità. Una soluzione futura potrebbe essere quella di trovare un algoritmo che calcoli automaticamente questo parametro usando un'euristica accettabile;

- Integrare il modulo con informazioni provenienti da più sorgenti. Finora il modulo effettua l'analisi su log di Snort, salvati su file di testo o su database. In futuro potrebbe essere interessante integrare queste informazioni con quelle provenienti da altri IDS (come *Prelude*), log di *Apache* o *syslog*.

Sommario

Saranno esaminati i dettagli sull'implementazione di questa soluzione nei prossimi capitoli. Nel capitolo 2 si discuterà dell'attuale stato dell'arte e il lavoro già compiuto nel settore dell'analisi intelligente delle intrusioni, mentre nel capitolo 3 si andrà più in profondità nell'analisi dell'architettura, dei componenti e della logica dietro il software, un'architettura la cui implementazione sarà trattata nel capitolo 4. Nel capitolo 5 si tratteranno i risultati di alcuni test eseguiti sul data set DARPA 99 e un'analisi delle performance, concludendo nel capitolo 6 con una discussione sugli obiettivi raggiunti e le funzionalità che potrebbero essere implementate in futuro.

Chapter 1

Introduction

The world of computer security actively involves nowadays the use of tools like *Intrusion Detection Systems* (*IDS*), a kind of software that can analyze flows of traffic and possibly find in them certain patterns recognized as possible traces of attacks (*alerts*). An *IDS* is usually considered as a second line of defense against malicious attacks, after firewalls, anti-malware software and effective network policies, when some actions are needed in order to recognize and possibly stop some potentially harmful traffic.

A first segmentation over the several types of IDS is done by considering *how* they react when an anomalous activity is found. An IDS can be:

- *passive*, it simply monitors and logs suspicious activities on the network;
- *active*, when it takes some decisions if a suspicious activity is found, for example by isolating part of the network from the outside or by appending the attacker's host to a black list. In this case we usually refer to them as *Intrusion Prevention Systems* (*IPS*).

Another type of logical segmentation, also discussed in [17], is usually done for this kind of software based on *when* it triggers the alerts. The alerts can be triggered:

- Every time an activity that does not match the allowed rules of traffic on a certain network is found. For example, such an IDS could trigger an alert if a user sends an unknown HTTP request type (not GET, POST, HEAD and so on) to a web server, even if actually no traces of attack were found in that request. Since some alerts may be triggered even when no actual attack is there, this IDS could raise a large number of *false positives*;
- Every time an activity actually matches a rule that identifies an attack in it. This approach anyway could be prone to a large number of *true negatives*, since an attack that does not match any trace or rule for known attacks may not be correctly recognized.

However, an IDS usually focuses on low-level attacks and anomalies, processing the alerts independently from each other, despite there may be logical connections among some of them. Therefore the result of an IDS is a log, analyzable by a system analyst, that contains details over possibly harmful elementary attacks, regardless towards the possible connections between them.

Some intruders could perform some complex attacks in order to achieve their goals, and the correlations between these attacks will not be found by an IDS. This means that a security analyst will spend much time in two activities that could be avoided if the IDS itself had enough knowledge and logic behind not to treat alerts as independent events:

- To remove the redundancy in alert logs (for example a huge number of "consecutive TCP small segments exceeding threshold" if a machine on the network has a broken implementation of the TCP/IP stack) and prune away the alerts "polluting" the logs with their redundancy, since, as shown in [6], few persistent root causes usually trigger the 90% of alerts in an Intrusion Detection System and these causes need to be identified in order to simplify the analysis of the logs;
- To find correlations among attacks reported by the IDS without investigating on the possible relations, or verifying if they are actually independant from each other.

These activities are expensive for a system administrator in terms of time, and in order to reduce the required time in these tasks many efforts have been done in the latest years, usually classifiable inside of two main categories of approaches:

- To reduce the amount of reported alerts by finding and grouping together alerts marked as "similar" based on some configurable criteria and rules (*alert clustering*);
- To correlate alerts likely to belong to the same complex attack scenario, or with a relation of causality between them, in order to generate a graph of complex attacks launched against the system or the network (*alert correlation*).

The software discussed in this document can be classified in both the tracks, since it first attempts to find clusters containing homogeneous or similar alerts considering some configurable clustering criteria, then it builds the correlation graphs expressing the causality relations among these clusters of alerts. The results of these operations are then shown to the user through an easy-to-use web interface, based on a web server developed from scratch for this purpose, or through some correlation graphs directly exported as PNG images or PostScript files.

In order to perform clustering and correlation tasks over a certain set of alerts raised by an IDS, we use some *machine learning* and artificial intelligence algorithms. It is in fact discussed in [10] how an IDS is only partly complete without some downstream artificial intelligence algorithms applied in order to extract new information (potential clusters or correlations) from the acquired knowledge over the recognized anomalies. Indeed, without these algorithms an IDS would raise alerts associated to elementary attacks without any further knowledge on the context where these alerts where triggered. In particular, the software discussed in this document mainly focuses on the following aspects:

1. To analyze the network traffic and parse the IDS triggered alerts as well, in order to associate, if possible, an alert to the TCP flow that raised it;
2. To find groups of similar alerts and group them together in homogeneous clusters, in order to:
   (a) Reduce the redundancy in the IDS logs, therefore simplifying the task of a system administrator who will analyze less alerts;
   (b) Easily recognize possible root causes for common alerts in a certain network;

   (c) Possibly achieve in a first attempt to recognize simple attack scenarios in the same cluster.

   A hierarchical clustering algorithm has been used in this software, based on configurable and context-aware clustering taxonomies;

3. To find correlations among sets of acquired alerts. In order to do this, the software uses at least four potential correlation indexes for the pairs of alerts:

   (a) A knowledge base-oriented correlation index, based on a model (*hyperalert*) that describes common prerequisites and consequences for a certain attack, and can therefore be used for building a causality graph among alerts;

   (b) A *pseudo*-bayesian correlation index, that correlates pairs of alerts considering how often they happen in the same *time window*;

   (c) A neural network-based correlation index, that computes the similarity of two alerts after a training phase and maps them to a 2-dimensional (and clusterizable) map, therefore transforming the problem of *alert similarity* into a geometric problem of distances between points and their possible strategies of clustering in a 2-dimensional space;

   (d) Some explicit manual correlations, provided by the user in case the system didn't recognize an actual correlation between two alerts, or it marked as true a non-existing correlation.

   Beyond these correlation indexes, the software has been designed as a modular framework, so it is possible to add more correlation indexes in a really simple way. The overall correlation indexes between two alerts is computed as weighted mean of their correlation indexes;

4. To provide an easy-to-use web-based user interface for exploring graphs of alert clusters, specifying new correlations, or analyze the network traffic associated to a specific alert.

The goal is to extend and build a system on top of an IDS, without impacting on its performance, that simplifies the analysis of alerts for a system administrator. This is done by automatically finding similar alerts, reducing the redundancy in alerts logs, attempting to recognize possible attack scenarios, and providing an easy user interface to browse all of this without manually analyzing plain text files or database outputs. All of this in a view of low impact on the overall performance of the IDS.

Chapter 2

Related works

The problem of grouping and correlating the alerts generated by an Intrusion Detection System in a more "human-oriented" way is actual and widely considered crucial in the world of IT defense systems. Indeed the current widely used Intrusion Detection Systems (*IDS*), like *Snort* or *Prelude*, succeed in recognizing security threats in a system analyzing network traffics with statically provided rules (for example, for checking if a certain packet or a sequence of packets contain a known payload). To a lesser extent they can also be successful in dynamically recognizing some more sophisticated attacks using dynamic preprocessor modules. For instance they can rebuild TCP flows through Snort's *Stream5* module, recognize attacks splitted on more packets or TCP sessions like a *port scan*, or attempt to decode SSL streams. But still the main problem for Intrusion Detection Systems is the overwhelming amount of data they produce in alert logs, often containing redundancy, false positives (or, even worse, true negatives), together with the poor or null capability of grouping similar alerts and/or correlating them. This means that the output of an Intrusion Detection System is often a huge redundant alert log, whose interpretation takes time to a security analyst or makes the security analysis impossible at all.

In these years many attempts and much research have been made with the aim to simplify the task of a system administrator or analyst studying the logs from an Intrusion Detection System. For the purpose of this document we are going to focus our attention on two of the paths followed in this research:

- **Alert clustering**, the set of features and mechanisms that allow the grouping of distinct alerts in more general or wider objects;

- **Alert correlation**, a mechanism that produces a graph of alerts' correlations starting from uncorrelated distinct alerts or group of alerts.

We are going to examine more in depth the work made in these sectors so far.

2.1 Alert clustering

The goals of clustering alerts are:

- To make the analysis of security alerts easier for a system administrator, grouping together alerts of the same type, received from the same machine and/or the same subnet and/or to the same ports, or received in a short time interval, and therefore reducing the redundancy typical in a system that triggers a large amount of alerts;

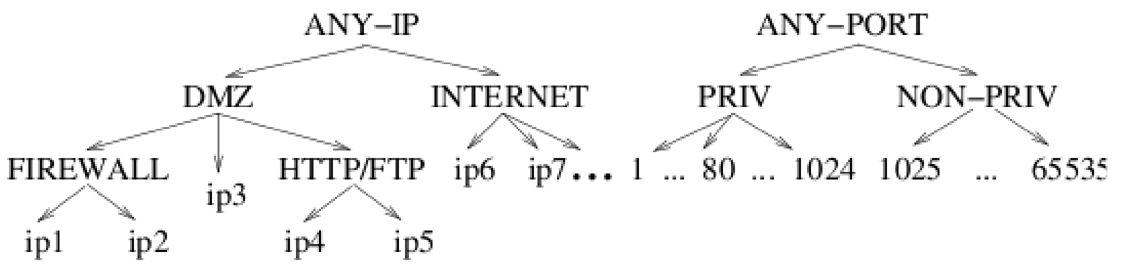

Figure 2.1: Example of a generalization hierarchy for IP addresses and TCP ports

- To identify the root cause of a set of alerts: as shown in [6], few persistent root causes usually trigger the 90% of the alerts in an Intrusion Detection System (for example, a broken implementation of the TCP/IP stack on a network machine can trigger thousands of "Fragmented IP" alerts). These causes usually trigger alerts of the same type, obviously treated as *false positives* from the analyst. Clustering these alerts can reduce a large amount of false positives to a single *grouped alert* object, making easier and less redundant the analysis of a log.

In [9] an approach based on data mining algorithms for supporting alert clusterization is suggested, discarding alerts marked as "normal" by the clusterization algorithm. Moreover, in [1] data mining algorithms are used for detecting anomalous network traffic patterns in real time.

The approach used in the software developed in this document is partly an evolution of the one proposed in [6]. In this approach we need the following definitions:

- **Root Cause**. We name *root cause* of an alert the reason why that alert is triggered by the (*IDS*). For example, a mistake or a failure in the implementation of the TCP/IP stack on a machine in our network fragmenting all the outbound traffic can trigger many "Fragmented IP" alerts on an *IDS*. Similarly, a broken or deprecated implementation of SSL keys on an SSH remote server can trigger many "SSH protocol mismatch" alerts. One of the goals of this approach to alerts clustering is grouping these alerts together, making it easier for an analyst to detect the root cause of the problem instead of having the alert logs "polluted" by many false positives;

- **Alert**. Alerts are event triggered by an *IDS* associated to anomalous operations, or potential attacks. An alert is modelled as a tuple over the Cartesian product $dom(A_1) \times ... \times dom(A_n)$, where $\{A_1, ..., A_n\}$ is the set of attributes of the alert (for example type ID, source and destination IP addresses and ports, timestamp...) and $dom(A_i)$, for all $i = 1...n$, is the domain of the i-th attribute. The value assumed by the instance of alert $\mathbf{a}$ for the attribute i is denoted by $\mathbf{a}[A_i]$;

- **Generalized attribute value**. This concept represents a subset of an attribute domain $dom(A_i)$. For example, *web servers* may identify the subset of IP addresses that host a web server, *DMZ* may identify the subset of IP addresses kept in the DMZ, and *unprivileged ports* may represent the subset of ports *(1025-65536)*. The *extended domain* of an attribute A_i, $Dom(A_i)$, is the union of the values in the "natural" domain of the attribute $dom(A_i)$ and the set of generalized attribute values defined over A_i. Starting from this definition we can define a *generalized alert* as a tuple over $[Dom(A_1) \times ... \times Dom(A_n)] \backslash [dom(A_1) \times ... \times dom(A_n)]$.

A generalized alert $\mathbf{g}$ models any ordinary (ungeneralized) alerts $\mathbf{a}$ whose attribute values are included in the generalized attribute values of $\mathbf{g}$: we write in this case $\mathbf{a} \trianglelefteq \mathbf{g}$:

$$\mathbf{a} \trianglelefteq \mathbf{g} \Longleftrightarrow \forall A_i \mid (\mathbf{a}[A_i] = \mathbf{g}[A_i] \vee \mathbf{a}[A_i] \in \mathbf{g}[A_i]) \tag{2.1}$$

For example, according to Fig. 2.1 we can say that the extended domain for the attribute *port* of an alert is $Dom(PORT) = \{1, ..., 65535, PRIV, NON - PRIV, ANY - PORT\}$.

- **Generalization hierarchies**. For each attribute A_i let $\mathcal{G}_i$ be a direct acyclic graph with a single root node representing the elements in $Dom(A_i)$. The graph $\mathcal{G}_i$ represents an *is-a* taxonomy for the values in the extended domain of A_i. In Fig. 2.1 [6] a sample taxonomy is shown, representing a possible generalization hierarchy for IP addresses in a network and TCP ports. For two elements $x, \hat{x} \in Dom(A_i)$ we say that $\hat{x}$ is *a parent* of x if there is a direct path in the generalization hierarchy $\mathcal{G}_i$ from $\hat{x}$ to x ("direct path" means "a path where the root node of the hierarchy is not an element", unless $\hat{x}$ itself is the root of the hierarchy). In symbols we would write $x \trianglelefteq \hat{x}$. Extending this definition from attribute values to alerts, given two alerts $\mathbf{a}$ and $\hat{\mathbf{a}}$ we say that $\mathbf{a} \trianglelefteq \hat{\mathbf{a}}$, i.e. $\hat{\mathbf{a}}$ *is a parent of* $\mathbf{a}$, if $\mathbf{a}[A_i] \trianglelefteq \hat{\mathbf{a}}[A_i]$ for each attribute A_i.

The *Exact Alert Clustering Algorithm* is an algorithm that, given a log of alerts, groups them into **clusters** such that *all the alerts in the same cluster share the same root cause.* Unfortunately no implementation can be made for this algorithm, as the *root cause* is a concept that makes sense only for a human mind. The software cannot be aware of root causes, so an algorithm grouping *exactly* the alerts sharing the same root cause *exactly* in the same alert is not implementable. We should then loose our clustering constraints, given as plausible the following hypothesis:

Hypothesis 1 *Root causes frequently manifest themselves in* LARGE *alert clusters* ADEQUATELY *modelled by* GENERALIZED ALERTS

This hypothesis starts from the assumption (usually true in the world of intrusion detection) that most of the alerts are false positives. This means that most of the IDS alerts are triggered by the same benign root causes. Under this hypothesis we can loose the concept of *alert clustering*, defining an *approximate alert clustering algorithm* like an algorithm that, given a log of alerts, finds *large* alert clusters *adequately* modelled by *general alerts.*

2.1.1 An ideal approach to alert clustering

In order to verify in our algorithm how close two alerts are we are going to introduce the notion of *dissimilarity*, $d(\cdot, \cdot)$, i.e. a function which takes two alerts as inputs and returns a numerical value that expresses how adequately these alerts can be modelled by a generalized alert. Being *dissimilarity* inversely related to *similarity*, given two alerts $\mathbf{a}_1, \mathbf{a}_2$ we have that a low coefficient for $d(\mathbf{a}_1, \mathbf{a}_2)$ means that these alerts can be easily modelled by a unique general alert.

Having defined the notions of alerts, generalized alerts and generalization hierarchies, we can now easily define the dissimilarity between two elements $x_1, x_2 \in Dom(A_i)$

as the length of the shortest path in $\mathcal{G}_i$ connecting x_1 to x_2 through a common parent p:

$$d(x_1, x_2) = \min\{\delta(x_1, p) + \delta(x_2, p) \mid p \in \mathcal{G}_i, x_1 \trianglelefteq p, x_2 \trianglelefteq p\} \tag{2.2}$$

where $\delta(\cdot, \cdot)$ returns the length of the shortest path between two nodes in $\mathcal{G}_i$.

The next step is to extend the notion of dissimilarity from attributes to alerts. At this point, having defined a generalized alert as an instance over the Cartesian product $X_{i=1}^{n} Dom(A_i)$, the dissimilarity between two alerts $\mathbf{a}_1, \mathbf{a}_2$ can be easily written as

$$d(\mathbf{a}_1, \mathbf{a}_2) = \sum_{i=1}^{n} d(\mathbf{a}_1[A_i], \mathbf{a}_2[A_i]) \tag{2.3}$$

With these definitions we can now define the notion of *heterogeneity* of a cluster of alerts $\mathcal{C}$. The heterogeneity is a function that takes a cluster $\mathcal{C}$ as parameter and returning a small value if $\mathcal{C}$ can be adequately modelled by a generalized alert. Let $\mathbf{g}$ be a generalized alert that is a parent for all the alerts in $\mathcal{C}$. The *average dissimilarity* is computed like the arithmetic mean of the dissimilarity coefficients between all the alerts in $\mathcal{C}$ and $\mathbf{g}$:

$$\bar{d}(\mathbf{g}, \mathcal{C}) = \frac{1}{|\mathcal{C}|} \sum_{\mathbf{a} \in \mathcal{C}} d(\mathbf{g}, \mathbf{a}) \tag{2.4}$$

We can then define the *heterogeneity* of a cluster as its minimum average dissimilarity coefficient. The minimum dissimilarity coefficient coincides with the average dissimilarity coefficient computed over an element $\mathbf{g} \in X_{i=1}^{n} Dom(A_i)$ that minimizes the coefficient itself:

$$H(\mathcal{C}) = \min\{\bar{d}(\mathbf{g}, \mathcal{C}) \mid \mathbf{g} \in X_{i=1}^{n} Dom(A_i),\ \forall \mathbf{a} \in \mathcal{C} : \mathbf{a} \trianglelefteq \mathbf{g}\} \tag{2.5}$$

We define a generalized alert $\mathbf{g}$ such that $\forall \mathbf{a} \in \mathcal{C} : \mathbf{a} \trianglelefteq \mathbf{g}$ and $\bar{d}(\mathbf{g}, \mathcal{C}) = H(\mathcal{C})$ a *cover* for $\mathcal{C}$. A cover is the most adequate model for a cluster. If the generalization hierarchies are trees instead of acyclic graphs, then there is exactly one cover for each alert cluster $\mathcal{C}$. For example, considering the figure 2.1, we have that the node *FIREWALL* is a cover for the cluster made by the nodes *(ip1, ip2)*, and *DMZ* is a cover for the cluster *(FIREWALL, ip3, HTTP/FTP)*. The problem of alert clustering can then be reduced to the following:

Problem 1 *Given an alert log $\mathcal{L}$, a parameter min_size $\in \mathbb{N}$, and a generalization hierarchy $\mathcal{G}_i$ for each attribute A_i, the* IDEAL ALERT CLUSTERING PROBLEM *$(\mathcal{L}, min\_size, \mathcal{G}_1, ...\mathcal{G}_n)$ is to*

- *Find a set $\mathcal{C} \subseteq \mathcal{L}$ that minimizes the heterogeneity $H(\mathcal{C})$*
- *The set $\mathcal{C}$ must satisfy the constraint $|\mathcal{C}| \geq min\_size$*

The problem of clustering can therefore be reduced to the problem of finding, among all the sets $\mathcal{C} \subseteq \mathcal{L}$, with $|\mathcal{C}| \geq min\_size$, the set with the minimum heterogeneity. Once this set is found, the algorithm can be run again on the remaining $\mathcal{L} \setminus \mathcal{C}$ alerts in order to search for other clusters.

The weak spot of the approach illustrated in Problem 1 is that it is an *NP-complete* problem (a proof of this can be found on [6, p.123]), which means that an algorithm able to solve this problem under these conditions simply cannot be developed. We need to loose again the constraints and the expectations over the clustering problem, in order to obtain a working clustering algorithm.

```
Input: $(\mathcal{L}, min\_size, \mathcal{G}_1, ..., \mathcal{G}_n)$
Output: A heuristic solution for the clustering problem
Algorithm:
    // Set the counters of 'grouped' alerts inside each a to 1
    for all alerts $\mathbf{a}$ in $\mathcal{L}$ do $\mathbf{a}$[count] = 1
    while $\forall\ \mathbf{a} \in \mathcal{L} : \mathbf{a}$[count] < $min\_size$ do
        $A_k$ = heuristic_best_attribute($A_1, ..., A_n$)
        for all alerts $\mathbf{a} \in \mathcal{L}$ do
            $\mathbf{a}[A_k]$ = father of $\mathbf{a}[A_k]$ in $\mathcal{G}_k$
        while $\exists\ \mathbf{a}, \mathbf{a}' \in \mathcal{L} : \mathbf{a} \equiv \mathbf{a}'$ do
            $\mathbf{a}$[count] = $\mathbf{a}$[count] + $\mathbf{a}'$[count]
            $\mathcal{L} = \mathcal{L} - \{\mathbf{a}'\}$
        end while
    end while
    return all the generalized alerts $\mathbf{a} \in \mathcal{L}$ having $\mathbf{a}$[count] $\geq$ $min\_size$
```

Figure 2.2: Pseudo-code implementation of the heuristic clustering algorithm

2.1.2 A practical approach to alert clustering

Given the *NP-completeness* of the ideal alert clustering problem, we need to develop an algorithm that finds the clusters $\mathcal{C} \subseteq \mathcal{L}$ which satisfy the constraint $|\mathcal{C}| \geq min\_size$, but not necessarily minimizing the heterogeneity $H(\mathcal{C})$: the target problem is to find the clusters whose heterogeneity is *the closest as possible* to the optimal one that may be found through the ideal alert clustering using a good heuristic function. This is the approach suggested on [6, p.124-125], and also developed in the software illustrated here with some variants. For making the algorithm simpler we assume that all the generalization hierarchies are represented by n-ary trees (this is also the most natural case in most of the contexts), even if [6] also proposes a variation for working with hierarchies modelled by direct acyclic graphs. Under this assumption we know that each subset $\mathcal{C} \subseteq \mathcal{L}$ has *exactly* one cover. Under this hypothesis, we can implement an algorithm like the one shown in fig. 2.2. Through this algorithm, we examine the alerts in an alert log $\mathcal{L}$, choosing at each cycle an attribute $A_k \in \{A_1, ..., A_n\}$ marked as *optimal* by a heuristic function, and replacing in all the alerts $\mathbf{a}[A_k] \in \mathcal{L}$ with the value of the parent of A_k in the hierarchy $\mathcal{G}_k$. After this substitution, the algorithm checks if there are two alerts that became identical (i.e. $\mathbf{a}[A_i] = \mathbf{a}'[A_i]\ \forall\ A_i$). In this case, the two alerts are merged together and the counter of merged alerts is increased as well. The algorithm is repeated as long as $\mathcal{L}$ contains cluster of alerts whose size is less than the one established as *minimum size.*

The only pending feature is a heuristic function which computes the best attribute to be "grouped". In [6, p.124] the following function is developed:

$$F_i = \max\{f_i(v) \mid v \in Dom(A_i)\} \tag{2.6}$$

with

$$f_i(v) = \sum_{\mathbf{a} \in \mathcal{L}\ :\ \mathbf{a}[A_i] = v} \mathbf{a}[count] \tag{2.7}$$

or, in *pseudo*-SQL,

$$f_i(v) = \text{SELECT } sum(count) \text{FROM } \mathcal{L} \text{ WHERE } A_i = v \tag{2.8}$$

The heuristic function computes the coefficients F_i for each attribute A_i, and returns the attribute A_k having the minimal value:

$$A_k = \{A_i \in A_1, ..., A_n \mid F_i = \min\{F_1, ..., F_n\}\} \tag{2.9}$$

In [18] a simpler model is put forward in order to cluster alerts. In this document first a *Security Event Time Window* W_{SE} is defined as the time interval that, given a set of alerts $\{a_1, ..., a_n\}$ raised by the same security event SE (so all the alerts in this set have the same *attack_type*), satisfies the inequation

$$\min\{a_i.timestamp - a_1.timestamp, \forall a_i\} \leq W_{SE} \leq a_n.timestamp - a_1.timestamp \tag{2.10}$$

On the basis of this notion, [18] defined three types of *hyper alerts* (to be intended like *clusters* of alerts, not to be confused with the definition of hyper alert provided in section 2.2):

Definition 1 (Type I Hyperalert) *Given a set of alerts*

$$A = \{a_1, ..., a_n\}$$

*located in the same W_{SE}, we say that A can be modelled by a **Type I Hyper Alert** if*

$$\begin{aligned} &a_1.srcIP = a_2.srcIP = ... = a_n.srcIP = srcIP \\ &a_1.dstIP = a_2.dstIP = ... = a_n.dstIP = dstIP \\ &a_1.alert\_type = a_2.alert\_type = ... = a_n.alert\_type \end{aligned}$$

In this case we identify the hyper alert type as a tuple

$$H = (srcIP, dstIP, AID, T, N)$$

with

$$\begin{aligned} &AID = \{a_i.alert\_ID \mid \forall a_i \in A\} \\ &T = a_1.timestamp \\ &N = |AID| \end{aligned}$$

Definition 2 (Type II Hyperalert) *Given a set of alerts*

$$A = \{a_1, ..., a_n\}$$

*located in the same W_{SE}, we say that A can be modelled by a **Type II Hyper Alert** if*

$$\begin{aligned} &a_1.srcIP = a_2.srcIP = ... = a_n.srcIP = srcIP \\ &a_1.dstIP \neq a_2.dstIP \neq ... \neq a_n.dstIP \\ &a_1.alert\_type = a_2.alert\_type = ... = a_n.alert\_type \end{aligned}$$

In this case we identify the hyper alert type as a tuple

$$H = (srcIP, dstIP, AID, T, N)$$

with

$$\begin{aligned}
&srcIP = a_1.srcIP \\
&dstIP = \{a_i.dstIP \mid \forall a_i \in A\} \\
&AID = \{a_i.alert\_ID \mid \forall a_i \in A\} \\
&T = a_1.timestamp \\
&N = |AID|
\end{aligned}$$

Definition 3 (Type III Hyperalert) *Given a set of alerts*

$$A = \{a_1, ..., a_n\}$$

*located in the same W_{SE}, we say that A can be modelled by a **Type III Hyper Alert** if*

$$\begin{aligned}
&a_1.srcIP \neq a_2.srcIP \neq ... \neq a_n.srcIP \\
&a_1.dstIP = a_2.dstIP = ... = a_n.dstIP = dstIP \\
&a_1.alert\_type = a_2.alert\_type = ... = a_n.alert\_type
\end{aligned}$$

In this case we identify the hyper alert type as a tuple

$$H = (srcIP, dstIP, AID, T, N)$$

with

$$\begin{aligned}
&srcIP = \{a_i.srcIP \mid \forall a_i \in A\} \\
&AID = \{a_i.alert\_ID \mid \forall a_i \in A\} \\
&T = a_1.timestamp \\
&N = |AID|
\end{aligned}$$

The model developed in this document operates a clustering operation into larger hyper alerts over the sets of alerts grouped in the same security event time windows.

2.2 Alert correlation

The next step in making IDS log analysis easier is correlating similar attacks, or attacks belonging to the same scenario. Alerts triggered by an IDS are not, in most of the cases, isolated events. It is very likely, for example, that after performing a port-scan on a host H that finds a known vulnerable remote service listening on the port p an exploit will be launched on *H:p* for taking advantage over that vulnerability. And after this, and after obtaining access to the machine, it is very likely that the attacker will attempt, for example, to mount on H a remote NFS or SMB resource hosted on another machine in the same network, or to access system logs for removing the traces of his attack. So in a real intrusion detection context the alerts raised by an IDS are not strictly independent events, but more or less correlated pieces in a larger *scenario* of a complex attack.

In [12], [4] and [14] an approach is suggested for modelling security alerts raised by an IDS as *hyperalerts*. A hyperalert is the instance of an alert (for example, the instance of a port-scan alert) modelled with

- Its **prerequisites**, i.e. a set of boolean conditions that must be true for the attack to be successful;
- Its **consequences**, i.e. a set of boolean conditions that *may* (note: not *must*) be true after the attack has been performed.

A hyperalert type can then be modelled, as explained in [12, p. 7], as a triple

$$T = (fact, prerequisites, consequences) \tag{2.11}$$

where

- *prerequisites* is the set of predicates that have to be true for the attack to be successful. These are boolean predicates connected by *AND* operators ($\wedge$). Each predicate can be modelled by a more or less complex boolean expression, with *OR* ($\vee$) operators, *NOT* ($\neg$) operators, and so on;
- *consequences* is the set of predicates that *may* be true in case the attack succeeds. These are more or less complex boolean predicates connected by *OR* operators ($\vee$);
- *fact* is usually an *assignment* of the variables used in *prerequisites* and *consequences* for modelling the current hyperalert.

The sets of preconditions and consequences can be modelled

- *statically*, in this case the developer of the application, or a third-part developer, or the analyst itself writes the description of a hyperalert on the basis of what usually occurs when this alert is raised;
- *dinamically*, in this case the description of a hyperalert is dinamically made by the software itself on the basis of the extracted *alert patterns* from the logs.

A *hyperalert* is an instance of an hyperalert type T modelled as the triple in (2.11).

Considering the example provided in [12, p. 7], we can model a hyperalert identifying an attempt to exploit a *sadmind* administration tool vulnerable to buffer overflow in the following way:

$$\begin{aligned} SadmindBufferOverflow \quad = \quad & (\{VictimIP, VictimPort\}, \\ & \{ExistsHost(VictimIP) \wedge \\ & \qquad VulnerableSadmind(VictimIP, VictimPort)\}, \\ & \{GainRootAccess(VictimIP)\}) \end{aligned}$$

An instance h of this hyperalert is an assignment of the variables in the *fact* for modelling an actual *alert* triggered by the IDS, with an associated couple *[begin_time, end_time]* expressing the timestamps of the alert itself. For example:

$$\begin{aligned} h(SadmindBufferOverflow) &= (\{10.8.0.1, 600\}, \\ &\quad \{ExistsHost(10.8.0.1) \wedge \\ &\qquad VulnerableSadmind(10.8.0.1, 600)\}, \\ &\quad \{GainRootAccess(10.8.0.1)\}), \\ &\quad [begin\_time = t_1, end\_time = t_2] \end{aligned}$$

In [4] a richer and more sophisticated model is proposed for modelling hyperalerts through **CAML** (*Correlated Attack Modelling Language*), a language developed with the purpose of modelling in the most complete and rich way a hyperalert as a *module* with its prerequisites, consequences and facts. In fig. 2.3 a CAML module is represented for modelling a *sadmind* buffer overflow alert. The syntax of CAML is quite rich, and allows to express constraints over temporal variables, hosts, services, files, users or user-specified predicates, and also predicates over relations between predicates themselves (a condition must be verified, for example, before, after or at the same time of another), or intersections or unions of predicates. It is possible to write *attack patterns* as well, as *template* general-purpose modules that can describe larger sets of alerts, for example an attack pattern matching several types of DoS or DDoS attacks or several types of web XSS exploitations.

Being able to express a hyperalert in function of its prerequisites and consequences, we can now define when two hyperalerts h_1 and h_2 are correlated, or when a hyperalert h_1 *prepares* for a hyperalert h_2. Let us denote as $P(h_i)$ the prerequisites of a hyperalert h_i, and as $C(h_i)$ its consequences. We can then say that h_1 *prepares* for h_2, or $h_1 \longrightarrow h_2$, if, according to [12, p. 10], there exist $p \in P(h_2)$ and $C \subseteq C(h_1)$ such that for all $c \in C$ we have $c.end\_time < p.start\_time$ and the conjunction of all the predicates in C implies p. In other words, h_2 is a consequence of h_1 if all the consequences of h_1 happened earlier than h_2 and it is possible "to draw" arrows between some of the consequences of h_1 and some of the prerequisites of h_2. In a formal way:

$$\begin{aligned} h_1 \longrightarrow h_2 \quad \text{iff} \quad & \exists p \in P(h_2), C \subseteq C(h_1) \mid \\ & \forall c \in C : c.end\_time < p.start\_time, \ \bigwedge_{c \in C} c \models p \end{aligned}$$

Let us focus again on the *sadmind* overflow example. We know that the hyperalert $h_{sadmindPING}$, triggered if a potential attacker checks if a host is up and if it runs a service vulnerable to *sadmind*, *prepares* a hyperalert $h_{sadmindBOF}$ if $h_{sadmindPING}$ happened before $h_{sadmindBOF}$ and, for example, $h_{sadmindPING}$ has the consequence

$$VulnerableSadmind(VictimIP, VictimPort)$$

and $h_{sadmindBOF}$ has the same predicate as prerequisite. How large the set C must be and how much the two hyperalerts should "match" for being considered as actually correlated is still an open issue and it will be focused in the next chapter.

In [12, p. 10] a constraint over the times of the two hyperalerts is also suggested. In this approach a simple clustering operation is performed by collecting more alerts of the same type in the same hyperalert, but a hyperalert should be a coherent object collecting alerts of the same type satisfying a *temporal location* constraint. Given a duration D (e.g. 100 seconds), a hyperalert must satisfy the constraint $\max\{t.end\_time \mid \forall t \in h\} - \min\{t.begin\_time \mid \forall t \in h\} < D$. Intuitively, it does not

```
module Sadmind-Buffer-Overflow (
activity:
        r1: Event (
               Source (
                     Node (Address(s: address)))
               Target (
                     Node (Address(t: address))
                     Service (tp: port))
               Classification (
                     origin == "cert"
                     name == "CA-2001-11"))
pre:
        p1: HasService (
               Node (Address(address == t))
               Service (port == tp))
        p2: Depends (
               Target ( Service (
                     implement == "Solaris"
                     ver1: "7")))
        VersionCmp (ver1, "7") < 0
        Subset ( r1, p1 )
post:
        Event: (
               starttime == r1.starttime
               endtime == DEFAULT_ENDTIME
               Source (
                     Node ( Address( address == s)))
               Target (
                     Node ( Address( address == t)))
               Classification (
                     origin == "vendor-specific"
                     name == "Root-Access"))
)
```

Figure 2.3: Example of a CAML module modelling a *sadmind* buffer overflow hyperalert

$$
\begin{array}{lll}
\pi[i,j] \models p & \text{iff} & \pi(i) = p,\ p \in P \\
\pi[i,j] \models \neg p & \text{iff} & \neg(\pi[i,j] \models p) \\
\pi[i,j] \models \varphi \wedge \psi & \text{iff} & \pi[i,j] \models \varphi \wedge \pi[i,j] \models \psi \\
\pi[i,j] \models \varphi \vee \psi & \text{iff} & \pi[i,j] \models \varphi \vee \pi[i,j] \models \psi \\
\pi[i,j] \models \Box\varphi & \text{iff} & (\forall k \in [i,j])(\pi[k,j] \models \varphi) \\
\pi[i,j] \models \Diamond\varphi & \text{iff} & (\exists k \in [i,j])(\pi[k,j] \models \varphi) \\
\pi[i,j] \models \bigcirc\varphi & \text{iff} & \pi[i+1,j] \models \varphi,\ i < j \\
\pi[i,j] \models \varphi \ ; \psi & \text{iff} & (\exists k \in [i,j))(\pi[i,k] \models \varphi \wedge \pi[k+1,j] \models \psi) \\
\pi[i,j] \models \varphi \oplus \psi & \text{iff} & \pi[i,j] \models (\varphi \wedge \neg\psi) \vee (\neg\varphi \wedge \psi) \\
\pi[i,j] \models \varphi^n & \text{iff} & \pi[i,j] \models \underbrace{\varphi \ ; \varphi \ ; \dots ; \varphi}_{n}
\end{array}
$$

Figure 2.4: Semantics of the Signature Interval Temporal Logic

make much sense to group under the same hyperalert two DoS alerts happened with a delay of one year from each other.

In [12, p. 13] an assumption is made over the predicates of the hyperalerts. In particular, it is stated that given a set of predicates

$$P = \{p_1(x_{11}, ..., x_{1k_1}), ..., p_m(x_{m1}, ..., x_{mk_m})\}$$

for any set of instantiations of the variables $x_{11}, ..., x_{1k_1}, ..., x_{m1}, ..., x_{mk_m}$ deriving all the predicates implied by P followed by the instantiation of the variables leads to the same result as instantiating all the variables and then deriving all the predicates implied by the instantiated predicates. Under this assumption, [12, p. 14] guarantees that the approach analyzed so far for correlating hyperalerts on the basis of their preconditions and consequences is *sound* and *correct* (THEOREM 3.19, [12]).

Moreover, being the approach suggested in [12] based on a DBMS storing the information over hyperalerts, finding potentially correlated hyperalerts is a problem reduced to the following SQL statement:

```
SELECT DISTINCT p.HyperAlertID, c.HyperAlertID
FROM PrereqSet p, ExpandedConseqSet c
WHERE p.EncodedPredicate = c.EncodedPredicate
AND c.end_time < p.begin_time
```

2.2.1 Formal models for multi-event attack signatures

In the context of a scenario of multi-event attacks a formal model may be useful to describe the relations among distinct events. In [13] a model based on the *Interval Temporal Logic ITL* is put forward. A *temporal logic* is an extension of *propositional logic* with the temporal operators *always* ($\Box$), *sometime* ($\Diamond$), *at the next moment* ($\bigcirc$). *ITL* can be considered like an extension of temporal logic where the notion of temporal satisfaction *by a point* is replaced by the notion of temporal satisfaction *by an interval*, that is a finite discrete linear structure. The model of *ITL* developed in [13] is named *Signature ITL* (*SigITL*). Let P be a set of atomic propositions: we can define the **syntax** of *SigITL* as follows:

$$\varphi ::= P \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \Box\varphi \mid \Diamond\varphi \mid \bigcirc\varphi \mid \varphi \ ; \ \varphi \mid \varphi \oplus \varphi \mid \varphi^n \tag{2.12}$$

Let π be a finite sequence, we denote by $\pi(i)$ the i^{th} event of the trace π. With $\pi[i,j]$ we denote the interval of π from the event i to the event j, having $i \leq j$. If a formula φ holds in the interval $\pi[i,j]$ we denote this by $\pi[i,j] \models \varphi$. Given these assumptions, in fig. 2.4 we can define the **semantics** of *SigITL*.

We can then express the event rules in SigITL in the following way:

- A sequence of events must occur in a sequential order: $\Diamond(p_1 \ ; \ p_2 \ ; \ ... \ ; \ p_n)$
- A sequence of events must occur, regardless of its order: $\Diamond p_1 \wedge \Diamond p_2 \wedge ... \wedge \Diamond p_n$
- Exactly one of a sequence of events $\bar{p}$ must occur: $\Diamond(p_1 \oplus p_2 \oplus ... \oplus p_n)$
- At least n repetitions of the event p_1 must occur: $\Diamond(p_1)^n$

The operator $\Diamond$ (sometime) next to each rule means that we are not necessarily interested in the occurrence of any event from a multi-event signature at the first time point only. Through this logic it is possible to express in a complete way the constraints over concurrence, sequence, ordering, occurrence and non-occurrence of events in a set, for example, of alerts raised by an IDS.

Chapter 3

Software architecture

The purpose of the developed software is mainly to cluster and correlate alerts triggered by an *IDS*, and to present them through an easy-to-use web interface. This is very useful and time-saving for the security analyst, who can save much work in identifying groups of similar alerts, common causes and possible correlations in a complex attack scenario.

This software is based on the following features in order to reach this aim:

- Fetch network traffic and alerts generated by an *IDS* (stored on plain logs or on a database) as inputs;

- Organize the single network packets into larger network streams and merge this information into the knowledge coming from the *IDS* (potential security alerts);

- Use several user-provided clusterization classes (over IP classes, port subsets, time intervals...) for clustering the alerts into larger grouped subsets;

- Correlate the clustered alerts using both exact and heuristic correlation indexes (the overall correlation index between two distinct clustered alerts will be computed as weighted average over the provided correlation indexes);

- Store the information over alert clusters, traffic streams and alert correlations (databases and log files) and provide a user interface for easily view and browse the correlation networks (static graphs, in PS or PNG format, and a dynamic web-based interface);

- Offer the possibility to store the traffic stream associated to a specific security alert in a format (PCAP, [28]) widely compatible with most of the network sniffers and analyzer software around, in order to allow a deeper inspection of suspicious network traffic;

- Offer the possibility to provide user feedbacks over wrong correlations (or *un*-correlations) made by the software, so that the system can adjust the correlations from its own mistakes.

A high-level schema of the software architecture is shown in fig. 3.1.

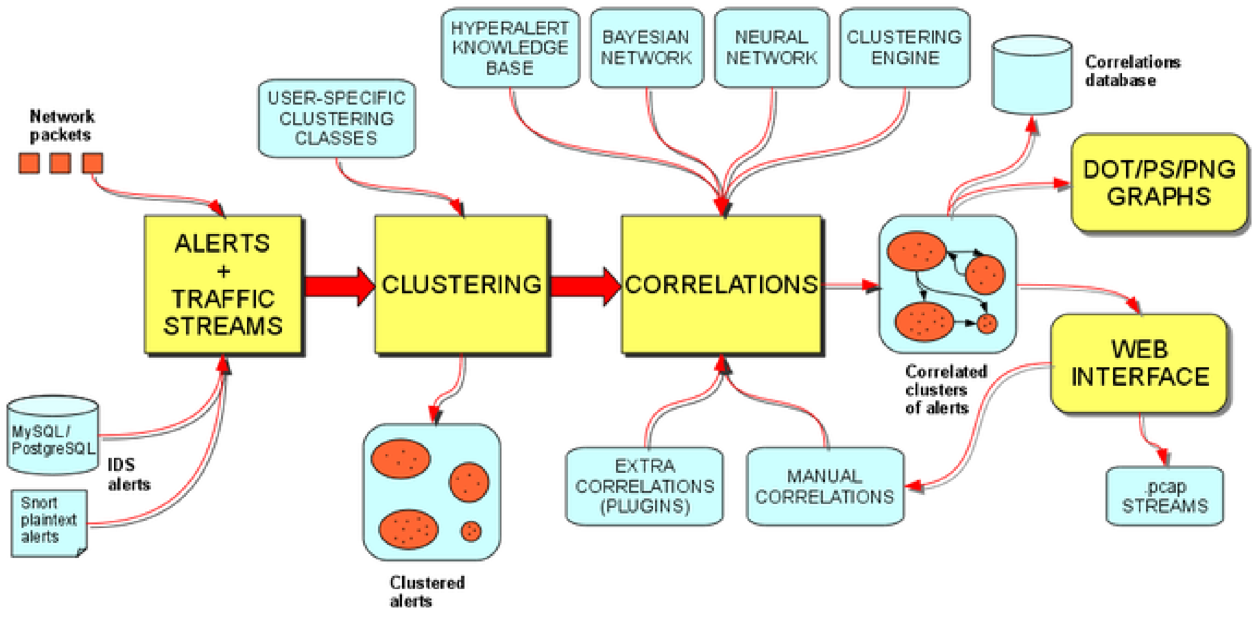

Figure 3.1: High level architecture of the software

3.1 Parsing *IDS* alerts and network traffic

The first step consists in merging together the pieces of information from network traffics and *IDS* alerts.

The packets fetched from the network interface are not necessarily ordered (on the contrary, they usually are not), so in order to associate an *IDS* alert to a traffic stream it is necessary to organize the packets received from the network interface into a data structure containing flows that group together the packets logically belonging to the same network session. A hash table is used for this purpose, having each element identified by a key

$$hash\_key = (srcIP, dstPort) \tag{3.1}$$

This key is used instead of the quadruple $(srcIP : srcPort, dstIP : dstPort)$ in order to allow a first "primitive" clustering, since the packets from the same host to the same destination port will be grouped in the same traffic flow, even when they come from a different source port. So the hash table will contain a set of *linked lists*, each representing an ordered set of packets identified by (3.1).

The main steps performed by the algorithm used for storing a sequence $\mathcal{P}$ of (unsorted) packets into a hash table $\mathcal{T}$ having each element identified by (3.1) are the following:

1. Initialize $\mathcal{T}$ as an empty hash table;
2. For each packet $pkt \in \mathcal{P}$ initialize key as a pair $(pkt.srcIP, pkt.dstPort)$;
3. If $\mathcal{T}$ already contains key, then get the flow associated to it and insert pkt in chronological order (this means that if there is a packet more recent than pkt, pkt will be inserted before this packet, otherwise it will be appended at the end

Input: ($\mathcal{P}$ set of packets)
Output: (hash table $\mathcal{T}$ containing the ordered flows of packets)
Variables: ($key, pkt, flow, element$)
Algorithm:

```
1: $\mathcal{T} \leftarrow \{\}$
2: for all $pkt \in \mathcal{P}$ do
3:    $key \leftarrow (pkt.srcIP, pkt.dstPort)$
4:    {Check if the table already contains the key of the current packet. If so, read the
      flow associated to that key}
5:    if $key \in \textsc{Keys}(\mathcal{T})$ then
6:       $flow \leftarrow \mathcal{T}[key]$
7:       for all $element \in flow$ do
8:          {Insert the new packet in the flow in chronological order}
9:          if $element.\text{timestamp} > pkt.\text{timestamp}$ then
10:            $pkt.\text{prev} \leftarrow element.\text{prev}$
11:            $element.\text{prev.next} \leftarrow pkt$
12:            $pkt.\text{next} \leftarrow element$
13:            break
14:         end if
15:      end for
16:      if $element = flow.\text{last\_element}$ then
17:         $flow.\text{last\_element.next} \leftarrow pkt$
18:         $pkt.\text{next} \leftarrow$ null
19:      end if
20:   else
21:      {If the key was not found, insert the new flow having the key of the current
         packet}
22:      $\textsc{Keys}(\mathcal{T}) \leftarrow \textsc{Keys}(\mathcal{T}) \cup \{key\}$
23:      $pkt.\text{next} \leftarrow$ null
24:      $\mathcal{T}[key] \leftarrow \{pkt\}$
25:   end if
26: end for
```

Figure 3.2: Algorithm used for storing a sequence $\mathcal{P}$ of packets into a sorted hash table

of the flow), otherwise create a new flow in $\mathcal{T}$ identified by *key* and having, for now, *pkt* as unique element.

A more detailed description of this algorithm is shown in the pseudo-code represented in 3.2).

A concurrent process parses the security alerts triggered by the *IDS* (they can be read from a plain text log or from a database). A security alert is then modelled as a tuple

$$\text{alert} = (alertID, alertType, srcIP, srcPort, dstIP, dstPort, timestamp, ...) \quad (3.2)$$

and an abstract *alert log* representation $\mathcal{L}$ is kept in memory as list of such tuples. Every time a new alert a is appended to $\mathcal{L}$ a lookup process in $\mathcal{T}$ starts as well, in order to find a network stream s having the same key $(srcIP, dstPort)$ of a. If such stream of packets is found, we mark s as "sequence of network packets associated to the alert a".

It is clear that the operation of organizing the packets into a data structure may generate a huge hash table stored in memory on a system with a large amount of inbound traffic, an unneeded waste of memory since we are not interested in *all* the traffic streams on the system but only on the ones associated to *IDS* triggered security alerts. In order to avoid such a waste of memory a boolean flag ("*is_observed*") is set on a traffic stream when it is associated to a security alert. Unless this flag is set, a stream is removed from the hash table when a packet with *RST* or *FIN* flags is received inside of that stream, or when the latest received packet is older than a configurable threshold.

3.2 Alert clustering

The next step after parsing *IDS* alerts into an abstract data structure and integrating them with traffic information is to group them into user-specific clusters. The algorithm used in this software is close to the one proposed in [6] with some improvements.

First of all a hierarchy tree is built over the clustering classes provided by the user, as shown in fig. 2.1. The hierarchies can be built over IP addresses and subnets (root node: 0.0.0.0/0), ports and port ranges (root node: 1-65536) and timestamps (root node: -2^{31}-2^{31}). The alerts are then grouped in clusters according to the following algorithm (the details in *pseudo-code* are shown in fig. 3.3):

1. Consider the set of user-provided classes $\mathcal{C}$ associated to the attribute a, for example, if a is the IP address, we may have a set of classes

   $$\mathcal{C} = (10.8.0.0/16, 192.168.1.0/24, 1.2.3.4, ...)$$

   each class has a range (values covered by the class, with a minimum and a maximum value) and the attribute a has a special class, identified as *root*, whose range covers each value that a may assume (again, if a is the IP address, we may identify 0.0.0.0/0 as *root node* of our hierarchy);

2. For each class $c_1 \in \mathcal{C}$ find a class c_2 such that

   - $c_1.\textsc{Range} \subseteq c_2.\textsc{Range}$
   - $|c_2.\textsc{Range}| = \min\{|c.\textsc{Range}| \ : \ c_1.\textsc{Range} \subseteq c.\textsc{Range},\ c \in \mathcal{C}\}$
   - $c_1 \neq c_2$

   i.e. c_2's range must include the range of c_1, c_2's range must the smallest one having this property and, of course, $c_1 \neq c_2$. c_2 is defined as *cover* of c_1, and c_1 will then be a child node of c_2 in the clusterization hierarchy $\mathcal{H}$;

3. If no node c_2 eligible as cover for c_1 is found, c_1 becomes a child node for the attribute a's root node.

[6] proves that, if the clusterization classes are defined in a way that generates an acyclic hierarchy graph, then each class different from the root node has *exactly* one cover.

Input: $\mathcal{C}$ set of user-provided clustering classes for the attribute a (IP address, TCP port, timestamp, ...)
Output: $\mathcal{H}$ an n-ary tree representing the clusterization hierarchy for the attribute a
Variables: $(c_1, c_2, bestParent, minRange)$
Algorithm:

```
1: $\mathcal{H} \leftarrow \{\}$
2: {All the classes initially have no children}
3: for all $c_1 \in \mathcal{C}$ do
4:    Children($c_1$) $\leftarrow \{\}$
5: end for
6: for all $c_1 \in \mathcal{C}$ do
7:    $minRange \leftarrow \text{MAX}_{int}$
8:    $bestParent \leftarrow$ null
9:    for all $c_2 \in \mathcal{C}, c_1 \neq c_2$ do
10:      {Check whether $c_2$ can be a "good parent" for $c_1$, i.e. the range represented by $c_1$ is included in the one represented by $c_2$}
11:      if $c_2$.Range $\subseteq$ $c_1$.Range then
12:         {Check whether the $c_2$'s range is the minimal one containing $c_1$'s range so far}
13:         if $(c_2.\text{Range}_{max} - c_1.\text{Range}_{max}) + (c_1.\text{Range}_{min} - c_2.\text{Range}_{min}) < minRange$ then
14:            $minRange \leftarrow (c_2.\text{Range}_{max} - c_1.\text{Range}_{max}) + (c_1.\text{Range}_{min} - c_2.\text{Range}_{min})$
15:            $bestParent \leftarrow c_2$
16:         end if
17:      end if
18:   end for
19:   {If no clustering class $c_2$ contains $c_1$, then $c_1$ is "adopted" as child of the root node over the attribute $a$}
20:   if $bestParent =$ null then
21:      $bestParent \leftarrow$ Root($a$)
22:   end if
23:   Children($bestParent$) $\leftarrow$ Children($bestParent$) $\cup \{c_1\}$
24: end for
25: $\mathcal{H} \leftarrow \{$Root($a$)$\}$
```

Figure 3.3: Algorithm used for building a clustering hierarchy $\mathcal{H}$

Input: $(value, \mathcal{H})$
Output: (the best node in $\mathcal{H}$ matching $value$)
Variables: $(next, child)$
Algorithm: getBestNode$(value, node \leftarrow \mathcal{H})$

```
1: $next \leftarrow$ **null**
2: **if** $node =$ **null then**
3:     **return null**
4: **end if**
5: **for all** $child \in$ Children$(node)$ **do**
6:     **if** $val \in child$.Range **then**
7:         $next \leftarrow child$
8:         **break**
9:     **end if**
10: **end for**
11: **if** $next =$ **null then**
12:     **return** $node$
13: **end if**
14: **return** getBestNode$(value, next)$
```

Figure 3.4: Algorithm used for assigning to each attribute a_j of an alert A_i the best "cover" node in the hierarchy $\mathcal{H}_j$

Given a set of attributes $\mathbf{a} = (a_1, ..., a_n)$ with their clustering hierarchies $\mathcal{H} = (\mathcal{H}_1, ..., \mathcal{H}_n)$ and a set of alerts $\mathbf{A} = (A_1, ..., A_m)$ (each alert is an instance of the attribute tuple $\mathbf{a}$), the purpose of the algorithm is now to associate, to each attribute a_j of each alert A_i, a node c_k in the hierarchy $\mathcal{H}_j$ that best models or generalizes the value assigned to the attribute a_j in A_i. The recursive algorithm is shown in detail in fig. 3.4. In words, it can be described as it follows:

1. Start the algorithm passing as parameters the attribute value to be matched and the root node of the associated hierarchy $\mathcal{H}_j$;
2. For each c_k child node of the current node, check if $value \in c_k$.Range;
3. If so, call the algorithm passing as parameters *value* and c_k;
4. If $\nexists c_k \mid value \in c_k$.Range, then the node that best matches *value* is the current one.

After each attribute a_j of each alert A_i has a node c_k associated in the hierarchy $\mathcal{H}_j$, the algorithm proposed in fig. 2.2 (from [6]) can be called in order to cluster the alerts.

The algorithm developed in this document improves the one proposed in [6] by making the clusterization more dynamic without needing to explicitely provide a *min_size* parameter for an alert cluster's minimum size. In order to compute this parameter in function of the current set of alerts we introduce the concept of *hererogeneity* $H(\mathcal{L})$ of the alert set $\mathcal{L}$. Provided that we want the alerts of the same type and received at a time interval less than a certain threshold to be classified in the same cluster (the type of an alert is provided by the *IDS*, for example *buffer overflow*, or *port scan*, or *attempted SQL injection*), and we do not want clusters containing alerts of different

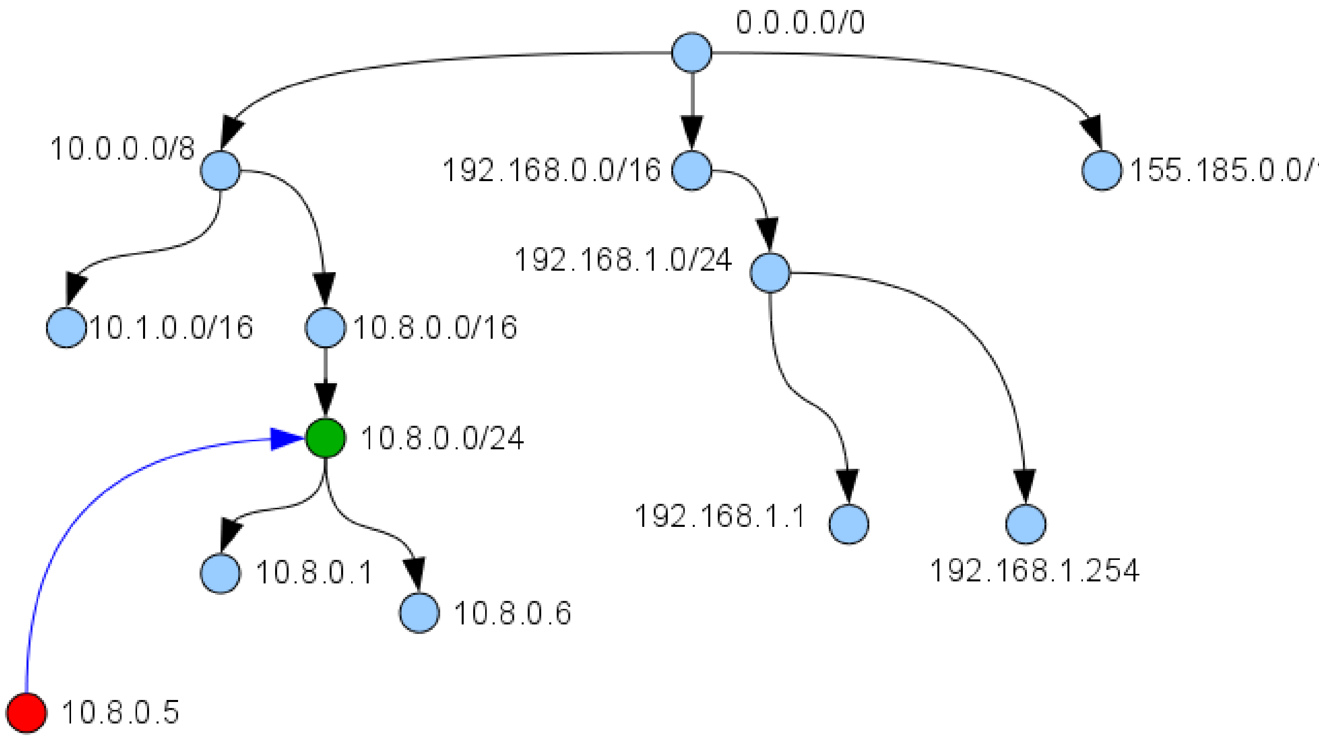

Figure 3.5: Matching of an item inside of an IP clusterization hierarchy

types, and knowing that $\mathcal{L}$ has $N_{distinct}$ distinct alert types, we compute an estimate of the heterogeneity of $\mathcal{L}$ as

$$H(\mathcal{L}) = \frac{N_{distinct}}{|\mathcal{L}|} \tag{3.3}$$

In pseudo-SQL:

SELECT COUNT(DISTINCT *alert_type*)/COUNT(*)
AS $H(\mathcal{L})$ FROM *alert_log* $\mathcal{L}$

We have

$$\frac{1}{|\mathcal{L}|} \leq H(\mathcal{L}) \leq 1 \tag{3.4}$$

with $H(\mathcal{L}) = \frac{1}{|\mathcal{L}|}$ when all the alerts in $\mathcal{L}$ are of the same type, and $H(\mathcal{L}) = 1$ when all the alerts have different types.

The optimal minimum size for the clustering algorithm is computed as inverse of the heterogeneity computed in (3.3):

$$min\_size = \frac{1}{H(\mathcal{L})} \tag{3.5}$$

When $H(\mathcal{L}) = \frac{1}{|\mathcal{L}|}$ (all the alerts have the same type) we have $min\_size = |\mathcal{L}|$: this means that the best cluster is a cluster grouping all the alerts in $\mathcal{L}$. When $H(\mathcal{L}) = 1$ (all the alerts have different types) we have $min\_size = 1$, i.e. the best choice is to take $|\mathcal{L}|$ clusters each containing a single alert.

Once we have the clustering hierarchies, each attribute a_j of each alert A_i associated to a node in these hierarchies, and a min_size parameter estimated in function of the heterogeneity of our set of alerts, we can run the algorithm proposed in fig. 2.2 for grouping the alerts into clusters.

Another improvement to the clustering algorithm discussed in [6] is an additional temporal constraint. In fact, we usually do not want to place in the same cluster two *buffer overflow* alerts over the same vulnerable application, sent from the same subnet to the same host and the same port, but considering that one of these was triggered one year later than the first one. So, given two alerts A_1 and A_2 with attribute instances $A_1 = (a_{1_1}, a_{1_2}, ..., a_{1_n})$ and $A_2 = (a_{2_1}, a_{2_2}, ..., a_{2_n})$, before or after executing a *generalization* step like the one proposed in fig. 2.2 (i.e. substituting to each attribute its parent in the clustering hierarchy until a cluster whose size is *min_size* is formed), we can group A_1 and A_2 in the same cluster if and only if

- $a_{1_i} = a_{2_i} \; \forall i = 1..n$
- $|A_1.time - A_2.time| \leq T_{max}$

where T_{max} is provided by the user (one minute, one day, one month, one year...) and represents the maximum amount of time that should occur between two alerts for them to be considered as parts of the same alert cluster.

3.3 Alert correlation

The software proposed in this document is a *framework* that supports multiple indexes for correlating two or more alerts. Given a set of correlation indexes $Corr = \{corr_1, corr_2, ..., corr_n\}$, the overall correlation index between two alerts A_1, A_2 is computed as weighted average over the correlation indexes of A_1 and A_2, i.e. each correlation index has a *weight*, that can be constant or vary in function of the number of steps made by the index's learning algorithm, in function of the information provided to the index, and so on:

$$corr(A_1, A_2) = \frac{1}{|Corr|} \sum_{i=1}^{n} w_i \cdot corr_i(A_1, A_2) \tag{3.6}$$

Given a set of alerts $\mathbf{A} = \{A_1, A_2, ..., A_m\}$, we compute $corr(\mathbf{A})$ as

$$corr(\mathbf{A}) = \{corr(A_i, A_j), \forall i, j = 1...m, \; i \neq j\} \tag{3.7}$$

In order to verify if two alerts are actually correlated (i.e. in order to "draw" an arch from A_i to A_j) we compute the average value and the standard deviation over this set:

$$\mu_{corr} = \frac{1}{N} \sum_{i=1,j=1}^{N} corr(A_i, A_j) \tag{3.8}$$

$$\sigma_{corr} = \sqrt{\frac{1}{N} \sum_{i=1,j=1}^{N} (corr(A_i, A_j) - \mu_{corr})^2} \tag{3.9}$$

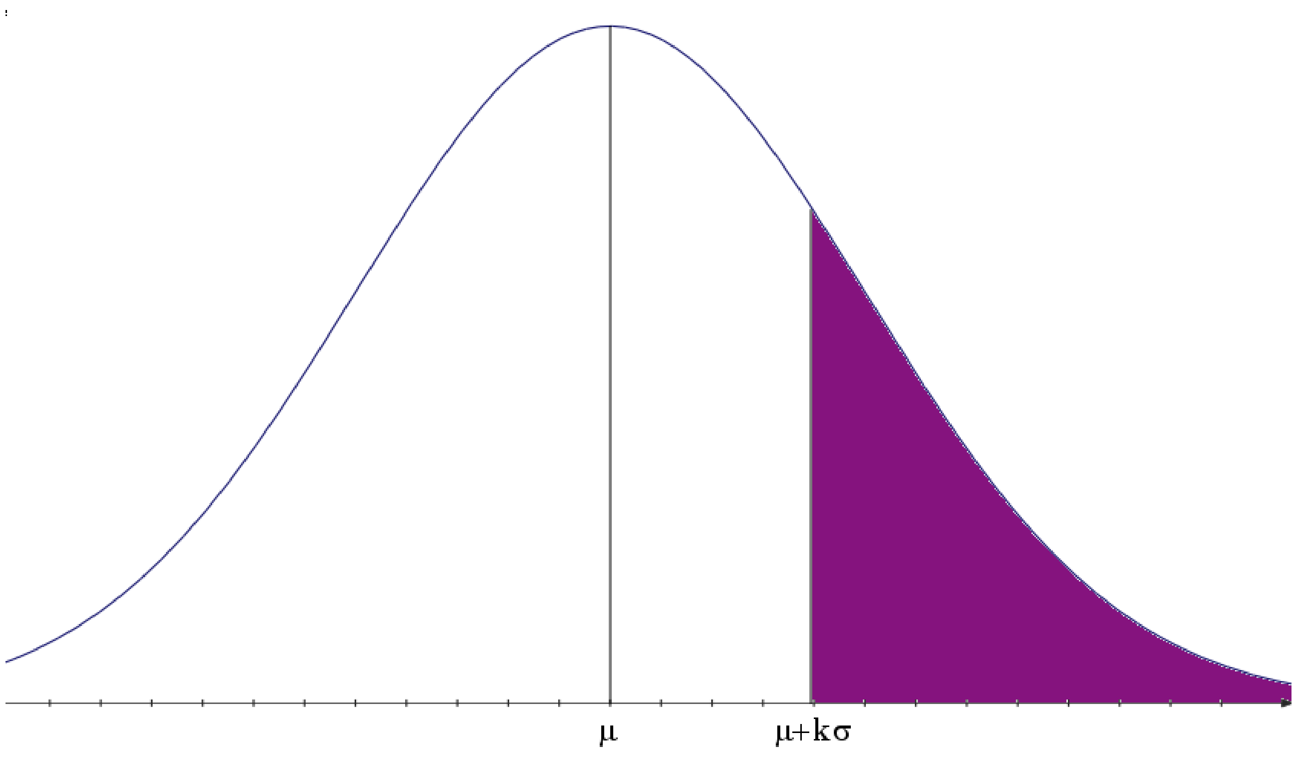

Figure 3.6: Representation of the gaussian function $f(x) = \frac{1}{\sqrt{2\pi}\sigma_{corr}} \exp\left(\frac{(corr(A_i,A_j)-\mu_{corr})^2}{2\sigma_{corr}}\right)$ The alerts marked as "correlated" are the ones having $corr(A_i, A_j) \geq \mu_{corr} + k\sigma_{corr}$

with $N = |corr(\mathbf{A})|$.
We then mark two alerts A_i, A_j as *correlated* if their correlation value $corr(A_i, A_j)$ is greater than a configurable threshold, in particular

$$\text{correlated}(A_i, A_j) \quad \text{iff} \quad corr(A_i, A_j) \geq \mu_{corr} + k\sigma_{corr},\ k \in \mathbb{R} \tag{3.10}$$

The parameter k is named *correlation threshold parameter* and expresses how strict or loose the constraint for correlating two alerts must be (see fig. 3.6). This parameter is specified by the user in function of the strictness needed for the correlation (a value close to 0 will correlate the alerts whose correlation index is greater or equal to the average correlation index, a value of $k < 0$ will also correlate alerts whose correlation index is less than the average, a value of $k \gg 0$ will result in a graph with few correlation archs between alerts). Empirically, a value of $k \simeq 1$ is a good compromise for the correlation.

3.3.1 Hyperalert knowledge base correlation

The first index used in this software comes from the explicit knowledge base of known hyperalerts, through an approach close to the one proposed in [12] and shown in section 2.2.

A knowledge base, already provided as part of the software, or created by the user, or derived from a model of history alerts, provides a model for each alert type T as

$$T = (fact, prerequisites, consequences) \tag{3.11}$$

The meanings of *prerequisites* and *consequences* of an alert are defined in section 2.2, i.e. we define as *prerequisites* the set of boolean conditions that must be verified for the alert to be triggered, as *consequences* the set of boolean expressions which *may* become true once the alert has been triggered, and as *fact* an assignment of the variables in the current alert context.

Once we have a model for two alerts A_1 and A_2 as *hyperalerts* (i.e. we know their prerequisites and their consequences), we compute the correlation index between A_1 and A_2 (or, even better, the index that expresses the probability that A_1 *prepares* A_2) as

$$corr(A_1, A_2) = 2\frac{|Pre(A_2) \cap Post(A_1)|}{|Pre(A_2) \cup Post(A_1)|} \tag{3.12}$$

where $Pre(A_i)$ identifies the set or prerequisites associated to the alert A_i and $Post(A_i)$ identifies its set of consequences. The coefficient 2 that multiplies this quantity is necessary for having $0 \leq corr(A_1, A_2) \leq 1$, since an item that occurs both in $Pre(A_2)$ and $Post(A_1)$ will be taken twice in the union set at the denominator and only once in the intersection set at the numerator.

Note that $corr(A_1, A_2) \neq corr(A_2, A_1)$ (this coefficient, differently from some others, is not commutative): this happens because $corr(A_1, A_2)$ expresses the probability to have an alert A_2 raised *after* an alert A_1, while $corr(A_2, A_1)$ expresses the probability to have an alert A_1 raised after an alert A_2. Suppose, for example, we have two alert types *PortScan* and *ShellcodeThroughCGI* defined as it follows:

$$\begin{aligned}
&Pre(PortScan) = \{HostExists(TargetHost)\}\\
&Post(PortScan) = \{HostIsUp(TargetHost),\\
&\qquad HostHasService(TargetHost, ANY\_SERVICE)\}\\
&Pre(ShellcodeThroughCGI) = \{HostIsUp(TargetHost),\\
&\qquad HostHasService(TargetHost, HTTP),\\
&\qquad HostRunsVulnerableCGIApps(TargetHost)\}\\
&Post(ShellcodeThroughCGI) = \{GainRemoteAccess(AttackerHost, TargetHost)\}
\end{aligned}$$

Briefly, in this model we assume that

- A precondition for a port scan to be successful is that the host exists;
- A consequence of a port scan is that the attacker knows that the specified host is up and running;
- Another consequence of a port scan is that the attacker knows that the host has some services running on some ports;
- A precondition for a shellcode injection in a buffer overflow prone CGI application is that the host is up, it has an HTTP server running and this server runs at least one vulnerable CGI application;
- A consequence of an exploited vulnerable CGI application is that the attacker gains remote access to the target machine.

A semantic engine can find by deduction that

$$HostHasService(TargetHost, ANY\_SERVICE) \models HostHasService(TargetHost, HTTP)$$

and if $TargetHost$ is the same in both the models then these predicates are equivalent. Using the formula expressed in eq. 3.12 we can find that

$$corr(PortScan, ShellcodeThroughCGI) = \frac{2 \cdot 2}{5} = 0.8 \tag{3.13}$$

This is the probability, based on the provided hyperalert model, that an attacker could launch a shellcode over a CGI web application after he performed a portscan that revealed the TCP port 80 open.

On the other hand, we have

$$corr(ShellcodeThroughCGI, PortScan) = 0 \tag{3.14}$$

since $Pre(PortScan) \cap Post(ShellcodeThroughCGI) = \emptyset$.

We assume that the correlation index expressed by this hyperalert knowledge has always unitary weight, since we consider the information fetched by the knowledge base as always true or plausible.

3.3.2 *Pseudo*-bayesian correlation

Another correlation index used in this software is the one that expresses the pseudo-*bayesian* conditioned probability of two alerts types $p(A|B)$ in function of the provided alert history. Given an alert log $\mathcal{L}$, two alert types $A, B \subseteq \mathcal{L}$ (for example $A = PortScan$ and $B = ShellcodeThroughCGI$), a set of alerts $a_1, a_2, ..., a_m \in A$ and $b_1, b_2, ..., b_n \in B$ we define the set $Corr(A, B)$ containing the potentially correlated alerts of type A and B as

$$Corr(A, B) = \{(a, b) \in A \times B : |t_a - t_b| \leq T_{max}\} \tag{3.15}$$

where t_a, t_b are respectively the timestamps of a and b and T_{max} is a configurable *time threshold* for considering a and b potentially correlated (one minute, one hour, one month...).

We also define the subsets A_{corr} and B_{corr} as

$$A_{corr} = A \cap Corr(A, B) = \{a \in A \mid a \in Corr(A, B)\} \tag{3.16}$$

and

$$B_{corr} = B \cap Corr(A, B) = \{b \in B \mid b \in Corr(A, B)\} \tag{3.17}$$

Given these definitions, we can define the *probability of an alert of type A given an alert of type B*, or $p(A|B)$, as the expression

$$p(A|B) = p(Corr(A, B)) - \frac{|\bar{A}_{corr}|}{|A|} \tag{3.18}$$

The reasoning behind this model is that, if two types of alert often happen in the same temporal context (for example if an *ICMP ping* is often followed by a *TCP port scan* in the short time), these alerts are very likely to be correlated.

We also define

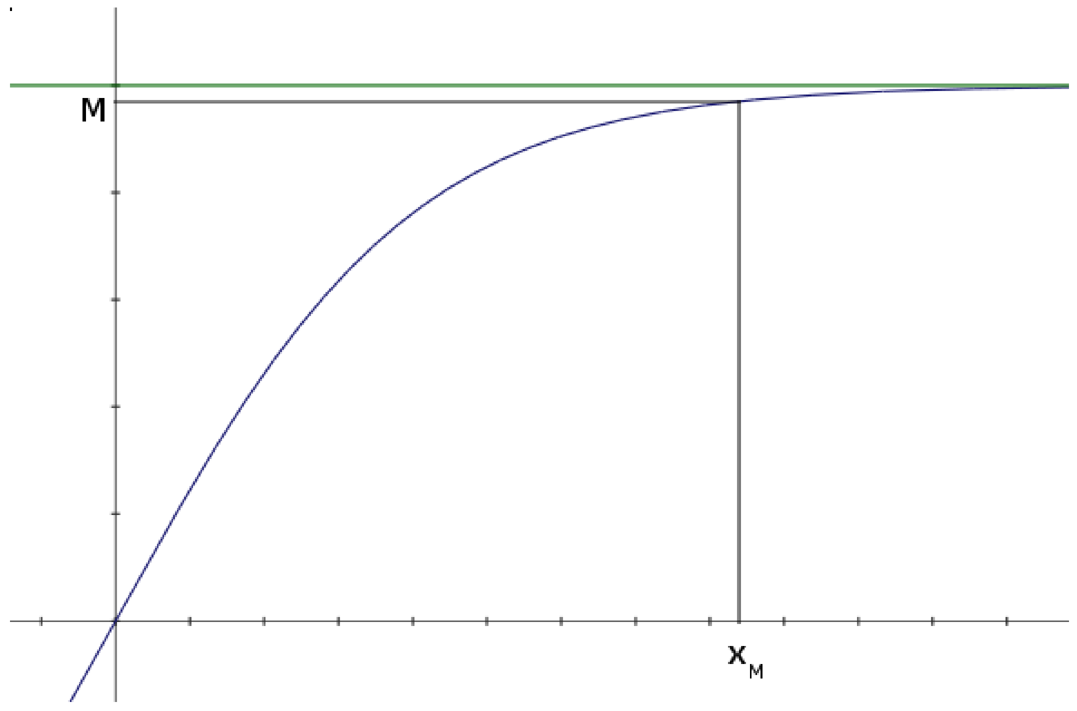

Figure 3.7: Plot of the hyperbolic tangent function $w(x) = \tanh\left(\frac{x}{k}\right)$, used as weight function for the *pseudo*-bayesian and neural correlation indexes

So the set $\bar{A}_{corr}$ identifies the instances of alerts of type A that are not correlated to any alert of type B, i.e. the instances of A that do not belong to $Corr(A, B)$. The quantity $|\bar{A}_{corr}|/|A|$ in eq. 3.18 then represents the fraction of alerts of type A not correlated to any alert of type B and it is used as a kind of *penalty coefficient*, this means that if we have a relatively large number of alerts A correlated to B, but the number of alerts A not correlated to B is much larger, the probability $p(A|B)$ cannot be high.

The probability of $Corr(A, B)$ is then computed as

$$p(Corr(A, B)) = \frac{1}{|Corr(A, B)|} \sum_{(a,b) \in Corr(A,B)} e^{-\frac{(t_a - t_b)^2}{K}} \tag{3.19}$$

Each term of the sum represents the time correlation between two alert instances a and b, correlation that decreases exponentially following a Gaussian decay in function of the quantity $|t_a - t_b|$.

The coefficient K represents *how large* the Gaussian "bell" should be, and it is computed in function of the parameter (user-provided) T_{max}, i.e. the maximum amount of time between two alerts for being considered correlated. In particular, we want to compute K so that the value of the function

$$f(x) = e^{-\frac{x^2}{K}} \tag{3.20}$$

with $x = |t_a - t_b|$, is C for $x = |t_a - t_b| = T_{max}$, with C defined as *cutoff value* (in this software $C = 0.01$, this means that the correlation between two alerts with a temporal distance T_{max} will be 1%):

$$f(T_{max}) = C \longrightarrow e^{-\frac{T_{max}^2}{K}} = C \tag{3.21}$$

We get an exponential equation with K as variable. Solving it, we obtain

$$K = -\frac{T_{max}^2}{\log C} \tag{3.22}$$

The problem now is how to weigh the index coming from this correlation approach. We cannot give a unitary weight to this index like we did for the one resulting from the operations over the hyperalert models, as here we are managing knowledge parsed out of an alert log, that may represent a partial, incomplete or wrong information. The idea is to give a low weight ($\simeq 0$) to this correlation index when the alert log $\mathcal{L}$ does not contain a relatively large number of alerts for performing some bayesian reasoning, and increase the weight (getting asymptotically closer and closer to a unitary weight) when the amount of information in the alert log increases. In order to do this, given the number x of alerts in the log as variable, the software uses the *hyperbolic tangent* tanh as weight function $w(x)$:

$$w(x) = \tanh\left(\frac{x}{k}\right) = \frac{e^{\frac{x}{k}} - e^{-\frac{x}{k}}}{e^{\frac{x}{k}} + e^{-\frac{x}{k}}} \tag{3.23}$$

The parameter k is computed in function of another parameter, x_M, that expresses the number of alerts that the log $\mathcal{L}$ should contain for letting the weight function be M (in this software $M = 0.95$, this means that the bayesian correlation will have a weight of 95% when $\mathcal{L}$ contains x_m alerts):

$$w(x_M) = M \longrightarrow \frac{e^{\frac{x_M}{k}} - e^{-\frac{x_M}{k}}}{e^{\frac{x_M}{k}} + e^{-\frac{x_M}{k}}} = M \tag{3.24}$$

We get an equation where k is the variable. The solution of this equation can only be computed through a numerical approximation. In particular, let $t = x_M/k$. For a fixed value of M (e.g. $M = 0.95$) an approximated solution t_{sol} can easily be computed through numerical methods, as shown in [37]. Once we have this solution associated to a certain value of M we can easily compute k as

$$k = \frac{x_M}{t_{sol}} \tag{3.25}$$

In other words, x_M expresses how fastly the weight of the bayesian correlation index should be increased when the number of alerts in the alert log increases.

3.3.3 Neural network correlation

An interesting approach for correlating alerts, also proposed in [8], [15], [19], [5] and others uses a *Self-Organizing Map* (*SOM*) for correlating alerts. A *SOM* (see [25] and [7]) is an artificial neural network that uses unsupervised learning algorithms for providing a discrete two-dimensional representation of a generic n-dimensional set of data. The network *maps*, after a training phase, each n-dimensional input element to a neuron on the output matrix: the "distance" between two generic elements in the input set can be computed as Euclidean distance ([22]) or Manhattan distance ([23]) between the output neurons on which these elements are mapped.

An example schema of a *SOM* is shown in fig. 3.8. For a generic network with K input neurons and a matrix of $M \times N$ output neurons we have $K \cdot M \cdot N$ links from each neuron in the input layer to each neuron on the output layer, each link has a weight that identifies the "strongness" of that link, and usually each neuron on the output matrix is connected to all of its adjacent neurons (it can also be connected to all of the other neurons, in more complex implementations).

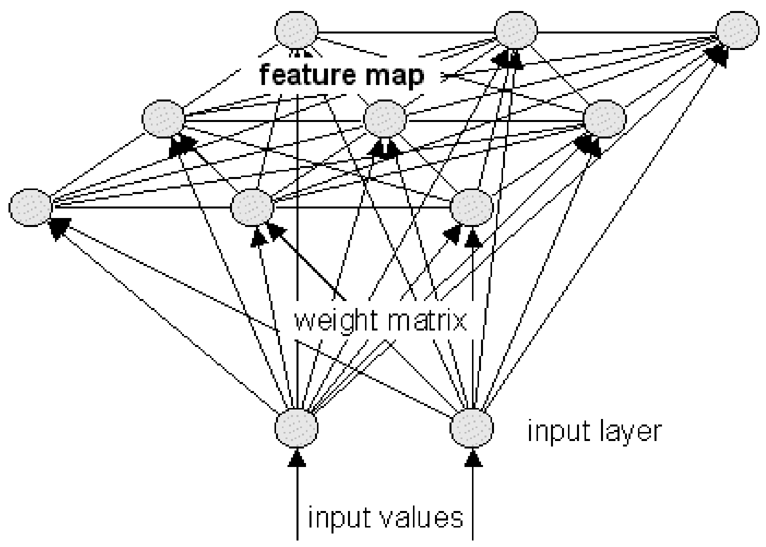

Figure 3.8: Sample schema of a *Self-Organizing Map* (*SOM*)

In this software a set of alerts $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_h\}$ is modelled as input data for the neural network. Each alert is a tuple

$$\mathbf{x}_i = \{alertType_i, srcIP_i, dstIP_i, srcPort_i, dstPort_i, timestamp_i\} \tag{3.26}$$

where each element is a numerical value normalized in $[0, 1]$.

The first step needed for using a *SOM* consists in initializing the weights of these links. Differently from a supervised neural network with back-propagation, were the initialization can be more or less random and then the weights can be adjusted by comparing the obtained results with the expected data, a *SOM* operates through *unsupervised* algorithms. This means that a poor or random initialization of the weights can lead to association mistakes hard to adjust, or requiring a very large number of training steps for being fixed. This computational training effort can be avoided with a smart initialization of the weights of the links from the input to the output layer. [11] and others propose an approach for weight initialization based on *Principal Component Analysis* (*PCA*), a method usually used for identifying the principal differences between signals and then make a dimensional reduction of them. The purpose is to find the *eigenfaces* face vectors in the reduced space) by finding the projection axes having the largest variance in the projected face images. The operation is then repeated in the orthogonal still uncovered subspace, until we have not a large variance in the set anymore. The problem is solved by computing the eigenvalues of the correlation matrix $\mathbf{R} \in \mathbb{R}^{K \times K}$, with K size of the input layer of the network:

$$\mathbf{R} = E\left\{(\mathbf{x} - \bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}})^T\right\} \tag{3.27}$$

with $\mathbf{x}$ representing the vector of inputs and $\bar{\mathbf{x}}$ its mean. A good strategy for initializing the values of the link weights (see [25]) is to choose them among the vectors in the subspace generated by the eigenvector associated to the maximum eigenvalue of the correlation matrix. This is not the used approach in this software anyway, since the calculation of the correlation matrix, the algorithm for finding its eigenvalues (or at least its maximum eigenvalue) and from there the resolution of the system for getting the associated eigenvector is computationally quite expensive. The strategy for the initialization of the weights in this software follows then the approach proposed in [16].

Given a *SOM* with an output matrix of $M \times N$ neurons and a training set $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_h}\}$, where each element is a vector of size K, the initialization of the neural weights follows these steps:

1. **Initialization of the weights on the four corners** of the output matrix. We first choose the two vectors $\mathbf{x}_a, \mathbf{x}_b \in \mathbf{X}$ having the maximum distance (computed, in this case, as K-dimensional Euclidean distance):

$$\mathbf{x}_a, \mathbf{x}_b \in \mathbf{X} \mid d(\mathbf{x}_a, \mathbf{x}_b) = \max\{d(\mathbf{x}_i, \mathbf{x}_j), \forall i, j = 1...h\} \tag{3.28}$$

   with

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{p=1}^{K} (x_{ip} - x_{jp})^2} \tag{3.29}$$

   The values of these vectors will be the values of the weight vectors, respectively, for the lower left corner $\mathbf{w}_{M1}$ and the the upper right corner $\mathbf{w}_{1N}$ (i.e. $\mathbf{w}_{M1} = \mathbf{x}_a$, $\mathbf{w}_{1N} = \mathbf{x}_b$).

   After this operation, the vector $\mathbf{x_c} \in \mathbf{X} - \{\mathbf{x_a}, \mathbf{x_b}\}$ having the maximum distance from $\mathbf{x_a}$ and $\mathbf{x_b}$ is chosen and used for initializing the weights of the upper-left corner $\mathbf{w}_{11}$. Finally the vector $\mathbf{x_d} \in \mathbf{X} - \{\mathbf{x_a}, \mathbf{x_b}, \mathbf{x_c}\}$ having the maximum distance from $\mathbf{x_a}$, $\mathbf{x_b}$ and $\mathbf{x_c}$ is chosen and used to initialize the weights of the bottom-right corner $\mathbf{w}_{MN}$.

2. **Initialization of the weights on the four edges**. The idea is to map the most distant elements in the training set on the four corners, as far as possible from each other, and fill the remaining neurons' weights by linear interpolation.

   The vectors of weights associated to the neurons on the four edges are initialized through the following relations:

$$\mathbf{w}_{1j} = \frac{j-1}{N-1}\mathbf{w}_{1N} + \frac{N-j}{N-1}\mathbf{w}_{11} \text{ for } j = 2, ..., N-1 \tag{3.30}$$

$$\mathbf{w}_{Mj} = \frac{j-1}{N-1}\mathbf{w}_{MN} + \frac{N-j}{N-1}\mathbf{w}_{M1} \text{ for } j = 2, ..., N-1 \tag{3.31}$$

$$\mathbf{w}_{i1} = \frac{i-1}{M-1}\mathbf{w}_{M1} + \frac{M-i}{M-1}\mathbf{w}_{11} \text{ for } i = 2, ..., M-1 \tag{3.32}$$

$$\mathbf{w}_{iN} = \frac{i-1}{M-1}\mathbf{w}_{MN} + \frac{M-i}{M-1}\mathbf{w}_{1N} \text{ for } i = 2, ..., M-1 \tag{3.33}$$

3. **Initialization of the remaining neurons** through the following relation:

$$\begin{aligned} \mathbf{w}_{ij} &= \frac{(j-1)(i-1)}{(N-1)(M-1)}\mathbf{w}_{MN} + \frac{(j-1)(M-i)}{(N-1)(M-1)}\mathbf{w}_{1N} + \\ & \frac{(N-j)(i-1)}{(N-1)(M-1)}\mathbf{w}_{M1} + \frac{(N-j)(M-i)}{(N-1)(M-1)}\mathbf{w}_{11} \end{aligned}$$

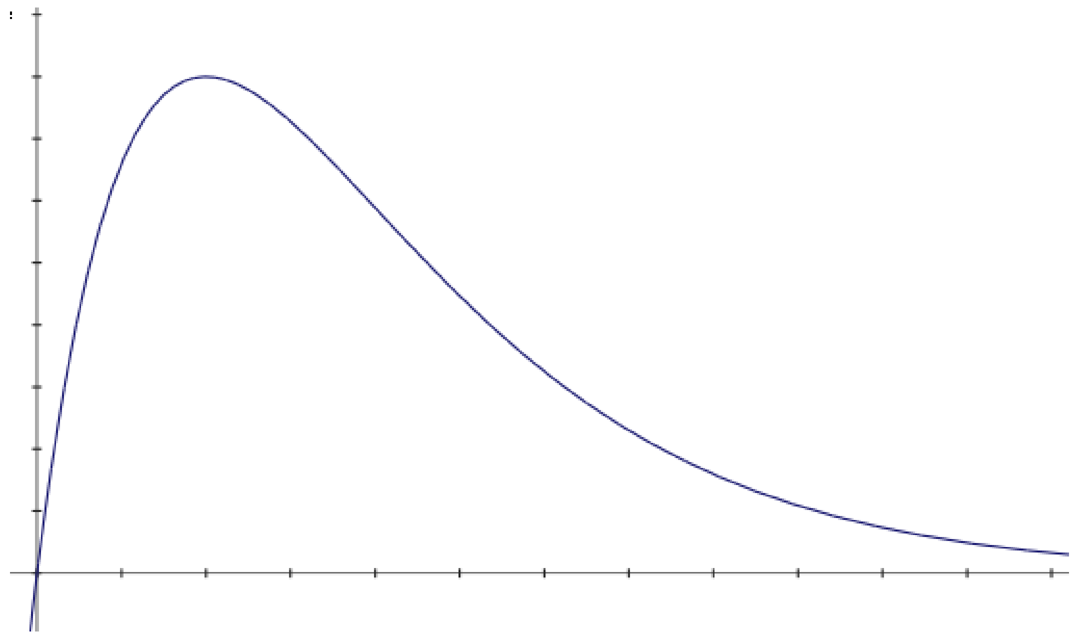

Figure 3.9: Plot of the function $\alpha(t)$ that expresses the learning rate in function of the current learning step

After initializing its weights, the network needs to be trained using, preferably, the same training set $\mathbf{X} = \{\mathbf{x}_1, ..., \mathbf{x}_h\}$ used for the initialization of the weights. For each training vector $\mathbf{x}_i \in \mathbf{X}$, $i = 1...h$ the algorithm finds the neuron that is most likely to "represent" it as

$$||\mathbf{x}_i - \bar{\mathbf{w}}|| = \min ||\mathbf{x}_i - \mathbf{w}_{jk}|| \tag{3.34}$$

for $j = 1...M$ and $k = 1...N$ and with the operator $||\cdot||$ representing the squared Euclidean distance. $\bar{\mathbf{w}}$ represents the vector of weights of the "best" output neuron, i.e. the neuron whose weights are closer to the input element $\mathbf{x}_i$.

At the learning step t then all the weights of the neural network are updated through the following relation:

$$\mathbf{w}_{jk}(t) = \mathbf{w}_{jk}(t-1) + \delta(\bar{\mathbf{w}}, \mathbf{w}_{jk})\alpha(t)\left(\mathbf{x}_i - \mathbf{w}_{jk}(t-1)\right) \tag{3.35}$$

for $j = 1...M$ and $k = 1...N$. The function δ represents the relation between the generic neuron weight vector $\mathbf{w}_{jk}$ and the neuron currenctly marked as "best" in function of their distance If the best neuron has coordinates (μ, ν) on the output matrix then the function δ is computed as

$$\delta(\bar{\mathbf{w}}, \mathbf{w}_{jk}) = \frac{1}{(|j-\mu| + |k-\nu|)^4 + 1} \tag{3.36}$$

The term in round parentheses at the denominator is the *Manhattan* distance between the neuron at (j, k) and the *best* neuron. The idea is that the influence of a training adjustment has a relatively strong impact on the weights of the best neuron associated to a training element and a moderate impact on its adjacent neurons, and this weight drastically decreases for the other neurons of the network (the distance function will be 1 for the neuron associated to $\bar{\mathbf{w}}$, 1/2 for its adjacent neurons, 1/17 for the neurons on the same diagonals, and so on). A Gaussian function can also be used for the distance:

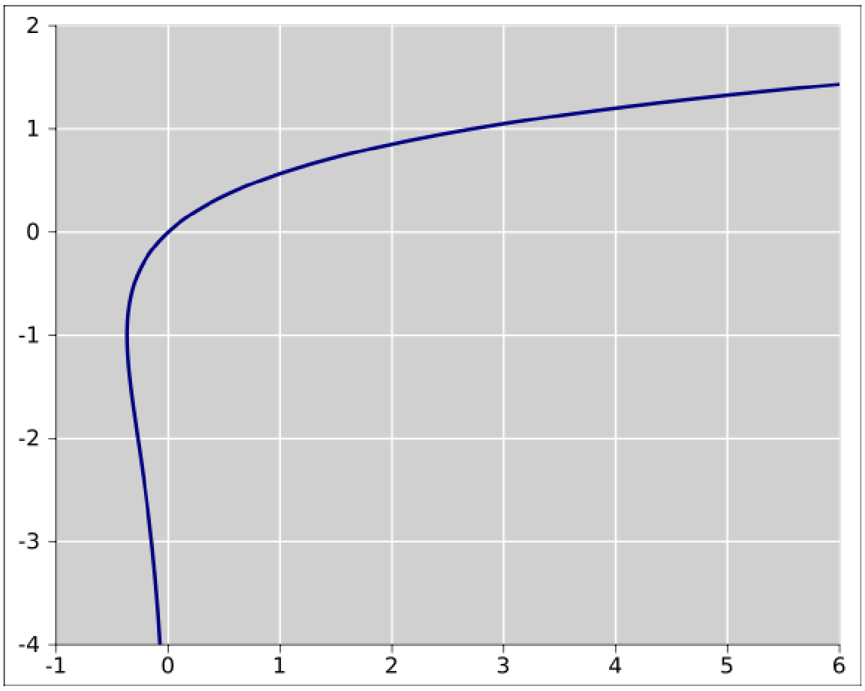

Figure 3.10: Plot of the Lambert W function, used for solving the equation 3.40 and getting the value of T. The upper branch for $W > -1$ is the function W_0 (principal branch), the lower branch with $W \leq -1$ is its analytic continuation W_{-1} (from [24])

$$\delta(\bar{\mathbf{w}}, \mathbf{w}_{jk}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(|j-\mu| + |k-\nu|)^2}{2\sigma^2}\right) \tag{3.37}$$

The function $\alpha(t)$ identifies the *learning rate* of the network at the learning step t. The idea is to keep this value relatively high ($\simeq 1$) at the beginning of the training phase and decrease it when more adjustments have been performed on the neural network. Anyway this policy (learning rate function initially close to 1 and then monotonically decreasing to 0) makes the network strictly dependant on the training elements at the beginning of the training phase. In order to avoid this asymmetry the software uses a different approach for the learning rate function, close to the one proposed in [20]. The learning rate is computed, in function of the current learning step t, as

$$\alpha(t) = M\frac{t}{T} \exp\left(1 - \frac{t}{T}\right) \tag{3.38}$$

where M is the maximum value that the learning rate should assume (0.8 in this software) and T is a parameter that expresses *how fastly* the learning rate function should tend to 0. The plot of $\alpha(t)$ is shown in fig. 3.10.

The parameter T is then computed in function of another (user-defined) parameter t_c, that represents the number of training steps that should be taken for having $\alpha(t) = C$ with C fixed (in this software $C = 0.01 \cdot M$, so t_c is the number of steps to take for having the value of the learning rate as 1% of its maximum value). A large value of t_c generates a learning rate function that tends very slowly to 0, while a small value generates a learning rate that soon reaches its peak then quickly moves to 0.

Our purpose is then to compute the parameter T in $\alpha(t)$ given t_c, given

$$\alpha(t_c) = C \tag{3.39}$$

We get

$$\frac{t_c}{T} + \log T = \log \frac{Mt_c e}{C} \tag{3.40}$$

that is a logarithmic equation having T as variable. An approximated solution of this equation is shown in [38] (with $t_c = 3$ and $Mt_c e/C = 10$ as sample values). The solution we are interested in among the two proposed ones is the minimum one:

$$T = \frac{Mt_c e}{C} \exp\left(W_{-1}\left(-\frac{C}{Me}\right)\right) \tag{3.41}$$

where W_{-1} is the analytic continuation of the Lambert W function (see [24]). An approximation of this function is computed in this software through its Taylor series as proposed in [3]:

$$W_{-1}(z) = \sum_{k=0}^{+\infty} \mu_k p^k \tag{3.42}$$

with

$$p = -\sqrt{2(ez+1)} \tag{3.43}$$

$$\mu_k = \frac{k-1}{k+1}\left(\frac{\mu_{k-2}}{2} + \frac{\alpha_{k-2}}{4}\right) - \frac{\alpha_k}{2} - \frac{\mu_{k-1}}{k+1} \tag{3.44}$$

$$\alpha_k = \sum_{j=2}^{k-1} \mu_j \mu_{k-j+1} \tag{3.45}$$

$$\mu_0 = -1,\ \mu_1 = 1,\ \alpha_0 = 2,\ \alpha_1 = -1 \tag{3.46}$$

The first terms of the series (3.42) are

$$W_{-1}(z) = -1 + p - \frac{1}{3}p^2 + \frac{11}{72}p^3 - \frac{42}{540}p^4 + \frac{769}{17280}p^5 - ... \tag{3.47}$$

The software computes the first 1000 terms of the series, a quite high precision (around the 5th or 6th decimal digit) is needed for the value since we are going to use it as exponent in (3.41). Anyway this is only computed on the first iteration and then it stays constant for all the learning process.

After the learning phase the network is ready to associate alerts given as tuples

$$\text{alert} = \ (alertType, srcIP, srcPort, dstIP, dstPort, timestamp) \tag{3.48}$$

with values normalized in $[0, 1]$. The network has as many input neurons as the number of parameters in the alert tuples and the dimensions $M \times N$ of the output layer are set by the user. The correlation index between two alerts A and B is computed as Euclidean distance between their associated neurons on the map:

$$Corr(A, B) = \sqrt{(\mathcal{N}_i(A) - \mathcal{N}_i(B))^2 + (\mathcal{N}_j(A) - \mathcal{N}_j(B))^2} \tag{3.49}$$

where $\mathcal{N}(X) = (\mathcal{N}_i(X), \mathcal{N}_j(X))$ is the set of coordinates where the network maps the input value X on the output matrix.

The weight of this correlation index is computed in the same way shown in (3.23), where the variable is the number of alerts from the alert log used as training set.

The network is actually re-trained at regular intervals using the alerts received in the log while then as updated training set, and incrementing the size (used as variable in (3.23)) of the training set.

3.3.4 Correlation clustering

The idea behind the *SOM* correlation index is to "map" each cluster of alerts on a 2-dimensional output layer, where the distance between each pair of points expresses the correlation between the cluster of alerts associated to those points. It is possible to execute a clustering algorithm over this 2-dimensional space. This operation aims to find alerts belonging to the same attack scenario, since two clusters of alerts mapped on two near points on the neural network output layer are very likely to belong to the same attack scenario.

In order to do this, the software keeps track, for each point of the neural network's output layer, of the alerts and clusters of alerts associated to it, it considers each pair (i, j), $1 \leq i \leq out\_rows, 1 \leq j \leq out\_cols$ having at least one associated alert or cluster of alerts as part of an input 2-dimensional dataset, and it performs a k-means [32] clusterization over this dataset. The output of this algorithm is the set of clusters that best group the given dataset, so each cluster contains the set of points (and their associated alerts or clusters of alerts) likely to belong to the same attack scenario.

Formally speaking, we name as D the dataset containing the pairs (i, j), $1 \leq i \leq out\_rows, 1 \leq j \leq out\_cols$ on the output matrix having at least one associated alert or cluster of alerts, then we initialize k centers for this dataset, each center identifying a cluster of data. The initialization of these centers follows a heuristic similar to the one used for initializing the weights on the *SOM* network, based on the fact that the centers of the clusters are more likely to be found in the proximity of points relatively distant from each other. So the pairs $\mathbf{x}_\mu = (i_\mu, j_\mu)$ and $\mathbf{x}_\nu = (i_\nu, j_\nu)$, with $\mathbf{x}_\mu, \mathbf{x}_\nu \in D$, are chosen as centers for the first two clusters:

$$\mathbf{x}_\mu, \mathbf{x}_\nu : ||\mathbf{x}_\mu - \mathbf{x}_\nu|| = \max ||\mathbf{x}_i - \mathbf{x}_j||, \forall\ 1 \leq i, j \leq |D|, i \neq j \tag{3.50}$$

The third center is chosen as the point having the maximum distance from both the assigned centers, the fourth one as the one having the maximum distance from the three assigned ones, and so on.

Once assigned the centers for the dataset, each element chooses its closest center as the one having the minimum distance from it. Doing so, at the generic iteration t of the algorithm a set $S_i^{(t)}$ containing the points associated to the i-th center, for $1 \leq i \leq k$, is defined as

$$S_i^{(t)} = \left\{ \mathbf{x}_j : ||\mathbf{x}_j - \mathbf{m}_i^{(t)}|| \leq ||\mathbf{x}_j - \mathbf{m}_{i^*}^{(t)}|| \text{ for all } i^* = 1..k \right\} \tag{3.51}$$

where $\mathbf{m}_i^{(t)}$ represents the coordinates of the i-th center at the step t. The centers of each of these sets are then updated as mean of the points contained inside of them:

$$\mathbf{m}_i^{(i+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} \mathbf{x}_j \tag{3.52}$$

The algorithm is repeated until $\mathbf{m_i^{(t)}} = \mathbf{m_i^{(t+1)}}$ $\forall i = 1..k$, i.e. no more changes are performed in the coordinates of the centers. This ending point is guaranteed to be reached [32]. At this step each set S_i contains the points associated to the i-th center, i.e. the i-th cluster of points. The complexity of the algorithm is [32] $\mathbf{O}(n^{dk+1} \log n)$, where d is the dimension of each item of the dataset ($d = 2$ in our case, as we are clustering points on a 2-dimensional output layer), k is the number of clusters in the dataset and n is the number of items.

The drawback of this algorithm, anyway, is that it requires the number k of clusters in the dataset, a piece of information often (like in this case) not available a priori. In order to overtake this difficulty our approach performs the clustering algorithm for each $k = 1..n$, and for each of this it computes a heuristic index through the **Schwarz criterion** [36] that expresses how "well-modelled" the dataset is by that number of clusters. For each number of cluster k we compute how well it models the given dataset through the following numerical index:

$$\mu_k = k \log n + \sum_{\mathbf{x}_j \in D} (\mathbf{x}_j - \mathbf{m}_i)^2 \tag{3.53}$$

where $\mathbf{m}_i$ is the center associated to the generic point $\mathbf{x}_j \in D$. In other words, this index computes the *average distorsion* represented by the "mapping" of a point to another point, also considering the chosen number k of centers (otherwise the optimal number of centers would always be $k = n$). The value k that best models the dataset D is the one having $\mu_k \leq \mu_{k^*}, \forall k^* = 1..n$, i.e. the one having the minimal Schwarz index.

After performing this algorithm on our dataset we obtain the sets of points most likely to belong to the same cluster, and since each point has one or more associated alerts or clusters of alerts we obtain the alerts or clusters of alerst most likely to be strictly correlated, i.e. belonging to the same attack scenario.

3.3.5 Manual correlations

Another important feature of the software is the possibility that it gives to the user to fix wrong correlations, or to provide some correlations not found by the software itself. These changes will always have the priority over the other indexes, so two alerts will not be correlated even if their overall correlation index is very high if the user specified that couple as "not correlated". On the other hand, two alerts will be correlated even when their overall correlation index is low if the user marked them as correlated.

3.3.6 Extra correlation indexes

The software is a modular *framework* for correlating alerts, so it is possible to extend its functionalities by adding more correlation indexes implemented as software modules, provided that they have a function that given two alerts returns a correlation index and another one that returns the current weight of that index.

Chapter 4

Software implementation

The software is built as a module on top of Snort [21], an open source intrusion detection system (*IDS*), and mainly developed in C language (except for the scripts that are part of the web interface, developed in *HTML/AJAX* for the client-side part and in *Perl CGIs* for the server part).

Being built as preprocessor module for Snort the features of *packet sniffing* are inherited from the *IDS* itself, as well as the configuration that can be performed using `snort.conf` file. The module is splitted into the following logical parts:

1. **Module configuration** (implementation contained in file `spp_ai.c`);
2. **Packet sniffing**, re-construction and sort of TCP flows (implementation contained in file `stream.c`);
3. **Alert parsing** from Snort `alert` file or from a database, and merging of traffic flows information, if available, inside alert objects (implementation contained in file `alert_parser.c`);
4. **Database management**, providing a transparent interface for interacting with databases regardless of the running *DBMS* (implementation contained in files `db.c` and, for the single *DBMSs*, `mysql.c` and `postgresql.c`);
5. **Clustering logic**, using the approach discussed in 3.2 for clustering the alerts into larger groups of similar items (implementation contained in file `cluster.c`);
6. **History management**, serializing the alert information over a binary file (and providing the functions for deserializing this file as well) and/or storing their information on a database (implementation contained in file `alert_history.c`);
7. **Correlation engine**, a sub-module that organises the several alert correlation indexes computing the overall correlations and offers the capability for adding multiple user-provided correlation modules (implementation contained in files `correlation.c` and `modules.c`);
8. **Hyperalert knowledge base *(KB)* management**, a sub-module that manages the knowledge about hyperalerts (preconditions and consequences of alerts) dynamically provided by the user or by the software itself, and computes on the basis of those a *KB-based* correlation index for each available pair of alerts (implementation contained in file `correlation.c`);

9. ***Pseudo*-bayesian correlation**, a sub-module that computes the correlation index between two alerts based on the acquired history of alerts and the probability of observing a certain alert given that another alert was observed in the same time window (implementation contained in file `bayesian.c`);

10. **Neural network-based correlation**, a sub-module that computes the correlation index among two or more alerts using a previously trained *Self-Organizing Map* (*SOM*), implementation contained in file `neural.c`. A small C library (*fsom*) was developed for managing the *SOM*, its code is available in the directory `fsom`: this library can also be used outside of this software;

11. **Clustering of potentially correlated clusters of alerts** (i.e. groups of alerts potentially belonging to the same attack scenario), a sub-module that performs k-means clusterization algorithm over the outputs of the *SOM* in order to find clusters of potentially correlated clusters of alerts, or alerts likely belonging to the same attack scenario (implementation contained in file `neural.c`). A small C library has been developed for performing the k-means algorithm over the neural network's output, using *Schwarz criterion* [36] for computing the optimal number of clusters: this library can also be used outside of this software;

12. **Storing of clustered alerts and correlations information** on an output database (implementation contained in file `outdb.c`);

13. **Web interface** for exploring alert clusters and the correlation graph through a web browser. A small web server was developed for this purpose (implementation contained in file `webserv.c`) running as concurrent thread in the software;

14. **Storing of network traffic associated to single alerts**, if available, in .pcap format, browsable using software like *tcpdump*, *Wireshark*, *Snort* itself and so on. This is done through the web interface, so that a single alert can be chosen inside of a cluster belonging to the correlation graph and its associated network traffic can be selected for download and inspected more deeply (implementation contained in file `htdocs/pcap.cgi`);

15. **Forcing of manual correlations or *uncorrelations***: the user can provide, using the web interface, information over correlations between alerts not found by the module or over pairs of alerts marked as "correlated" by the module but actually not correlated. These "forced" correlations will be passed back to the correlation module and will have the precedence over any other correlation index (implementation contained in file `htdocs/correlate.cgi`).

4.1 Module configuration

Being a module for Snort IDS, the configuration of this software can be directly implemented in `snort.conf` configuration file. A sample configuration is shown in fig. 4.1 (these lines can be simply pasted in your `snort.conf` file), and more details are available in `README` file. The meaning of these options is the following:

- `alertfile`: Path to the plain-text file where Snort stores its alerts;

- `alert_bufsize`: Size of the buffer that contains the alerts that will be sent to the *serializer* thread for being stored on the history file. The alerts are stored when this buffer is full or when a certain time interval has elapsed;

```
 preprocessor ai:
alertfile "/your/snort/dir/log/alert"
alert_bufsize 30
alert_clustering_interval 300
alert_correlation_weight 5000
alert_history_file "/your/snort/dir/log/alert_history"
alert_serialization_interval 3600
bayesian_correlation_interval 1200
bayesian_correlation_cache_validity 600
cluster ( class="dst_port", name="privileged_ports", range="1-1023" )
cluster ( class="dst_port", name="unprivileged_ports", range="1024-65535"
)
cluster ( class="src_addr", name="local_net", range="192.168.1.0/24" )
cluster ( class="src_addr", name="dmz_net", range="155.185.0.0/16" )
cluster ( class="src_addr", name="vpn_net", range="10.8.0.0/24" )
cluster ( class="dst_addr", name="local_net", range="192.168.1.0/24" )
cluster_max_alert_interval 14400
clusterfile "/your/snort/dir/log/clustered_alerts"
corr_modules_dir "/your/snort/dir/share/snort_ai_preproc/corr_modules"
correlation_graph_interval 300
correlation_rules_dir "/your/snort/dir/etc/corr_rules"
correlated_alerts_dir "/your/snort/dir/log/correlated_alerts"
correlation_threshold_coefficient 1
database ( type="dbtype", name="snort", user="snortusr",
password="snortpass", host="dbhost" )
database_parsing_interval 30
hashtable_cleanup_interval 300
manual_correlations_parsing_interval 120
max_hash_pkt_number 1000
neural_clustering_interval 1200
neural_network_training_interval 43200
neural_train_steps 10
output_database ( type="dbtype", name="snort", user="snortusr",
password="snortpass", host="dbhost" )
output_neurons_per_side 20
tcp_stream_expire_interval 300
use_stream_hash_table 1
webserv_banner "Snort AIPreprocessor module"
webserv_dir "/prefix/share/htdocs"
webserv_port 7654
```

Figure 4.1: Sample configuration of the module in file `snort.conf`

- `alert_clustering_interval`: Interval, in seconds, between two executions of the thread for clustering alerts;
- `alert_correlation_weight`: When the number of alerts specified by this option is stored in the history (history binary file or output database), then the weight of the heuristic correlation indexes will be 95% of its maximum value. This expresses "how quickly" the weight of heuristic correlation indexes should increase in function of the stored alerts knowledge. This is the x_M parameter shown in eq. 3.23 for the *pseudo*-bayesian and the neural correlation indexes;
- `alert_history_file`: Path to the file where the history of the acquired alerts will be stored (and loaded);
- `alert_serialization_interval`: Interval, in seconds, between two executions of the thread that serializes the alerts to the binary history file. The serialization will be performed when this interval is elapsed or when the buffer containing the alerts to be stored is full (in both the cases the timeout counter is reset);
- `bayesian_correlation_interval`: Interval, in seconds, between two executions of the thread that computes the *pseudo*-bayesian correlation between the pairs of stored alerts;
- `bayesian_correlation_cache_validity`: The history of alerts used for computing the *pseudo*-bayesian correlation, in order to improve the performance of the software, is not loaded at each execution of the thread but saved in a cache, managed as a *hash table*. This value expresses the interval, in seconds, before a certain instance of this cache goes outdated and needs to be refreshed;
- `cluster`: This option expresses a clustering class to be used by the hierarchical clustering algorithm explained in 3.2. In particular, each occurrence of this option represents a node in the clustering hierarchy, having as sub-options
  - `class`, it can be *src_port, dst_port, src_addr, dst_port* or *timestamp*;
  - `name`, a label that identifies the current cluster class;
  - `range`, the range modelled by the clustering class, it can be, for example, "80" or "1-1024" for a range of ports, "192.168.1.1" or "10.8.0.0/16" for a range of IP addresses, and "Mon-Fri" or "January" for a range of timestamps.
- `cluster_max_alert_interval`: Maximum interval, in seconds, that should occur between two alerts for being considered as part of the same cluster, if all the other clustering conditions are also verified;
- `clusterfile`: File that will contain the information, in plain text, over the clusters of alerts;
- `corr_modules_dir`: Directory containing the extra modules for extra correlation indexes between alerts;
- `correlation_graph_interval`: Interval in seconds between two executions of the thread that re-builds the alert correlation graph;
- `correlation_rules_dir`: Directory containing the rules, in XML format, representing the knowledge of the hyperalert models;

- `correlated_alerts_dir`: Directory that will contain the graph representing the information over correlated alerts, in DOT/PS/PNG format;
- `correlation_threshold_coefficient`: This is the k coefficient shown in eq. 3.10 and it expresses how "sensible" the software should be in correlating alerts depending on the average correlation value and its standard deviation;
- `database`: If this option is used, the alerts triggered by Snort will be parsed from this database instead of the plain-text log file. This has the sub-options
  - `type`, the DBMS used for storing Snort alerts (so far *MySQL* and *PostgreSQL* are supported;
  - `name`, name of the database;
  - `user`, username for accessing the database;
  - `password`, password for accessing the database;
  - `host`, host where the database is stored.
- `database_parsing_interval`: Interval, in seconds, between two executions of the thread that reads the newly stored alerts in the alert database;
- `hashtable_cleanup_interval`: Interval, in seconds, between two executions of the thread that cleans the hash table containing the acquired TCP streams. A TCP stream is only removed if it is not marked as "suspicious" (i.e. it is not associated to an alert) and the latest received packet in that stream is older than the `tcp_stream_expire_interval`;
- `manual_correlation_parsing_interval`: Interval, in seconds, between two executions of the thread that parses the manual correlations (or "uncorrelations") set by the user;
- `max_hash_pkt_number`: Maximum number of packets that each element of the stream hash table should hold, set it to 0 for no limit;
- `neural_clustering_interval`: Interval in seconds between two executions of the thread for clustering (using k-means) the alerts on the output layer of the neural network in order to recognize likely attack scenarios. Set this to 0 if you want no clusterization;
- `neural_network_training_interval`: Interval, in seconds, between two executions of the thread that trains the *SOM* network. This value is usually quite high since the training of the neural network is an expensive operation and it should only be executed when we have a relatively large set of new alerts to be used as new training set;
- `neural_train_steps`: Number of steps for training the neural network with the new training set;
- `output_database`: If this option is set (*strongly suggested*), the information over clusters and correlations will also be saved on an output database (if this is not specified, the information about clusters will only be stored on a plain-text file and the information about correlation will be stored as graph in DOT, PS and PNG formats). The syntax and semantics of the sub-options used here are the

same of the ones used for the `database` option, but the database where this information is stored is not necessarily the same (indeed, it is strongly suggested to keep the generic information over Snort-generated alerts and the one over module-generated correlations in two distinct databases);

- `output_neurons_per_side`: Number of neurons on each side of the *SOM* neural network. A larger value can map two very similar alerts on two distinct neurons, but the computational work will be more expensive;

- `use_stream_hash_table`: Set this option to 0 if you do not want to use the hash table for storing the streams of packets associated to alerts, this is a good choice on a system where many alerts are triggered;

- `tcp_stream_expire_interval`: Interval, in seconds, that should occur before a certain TCP stream is marked as "expired" and so ready for being removed from the hash table at the next execution of the cleanup thread. This interval is computed as difference between the current timestamp and the timestamp of the latest received packet in the stream;

- `webserv_banner`: Message to be printed in the *Server* HTTP header of the responses and at the bottom of the error pages;

- `webserv_dir`: Directory that will contain the web pages and CGI scripts from the module;

- `webserv_port`: Port where the web server will listen onto. If this parameter is set on 0, the web server will not be started at all. In this case, the web pages and CGI scripts should be copied to the directory of another web server (Apache or lighttpd, for example) in order to use the web interface.

These options are parsed from `snort.conf` in the initialization phase, and placed in a shared object of type `AI_config` (see the file `spp_ai.h` for the implementation) visible to all the modules.

4.2 Packet sniffing and sort of TCP streams

The function `AI_process()` in file `spp_ai.c` has the purpose of processing the single TCP packets using Snort built-in packet sniffer and passing them, as `SFSnortPacket*` objects, to the module for the management of the table of streams implemented in file `stream.c` through `AI_pkt_enqueue` function. This function enqueues, using the algorithm described in fig. 3.2, each packet to the corresponding TCP stream, managed as a linked list of `pkt_info*` objects inside a hash table sorted by the key pair $< src\_addr, dst\_port >$. If no stream with the corresponding key pair is found, a new one is created. The hash table is managed using the small *uthash* C library [40].

The `pkt_info` and `pkt_key` structures are defined as follows:

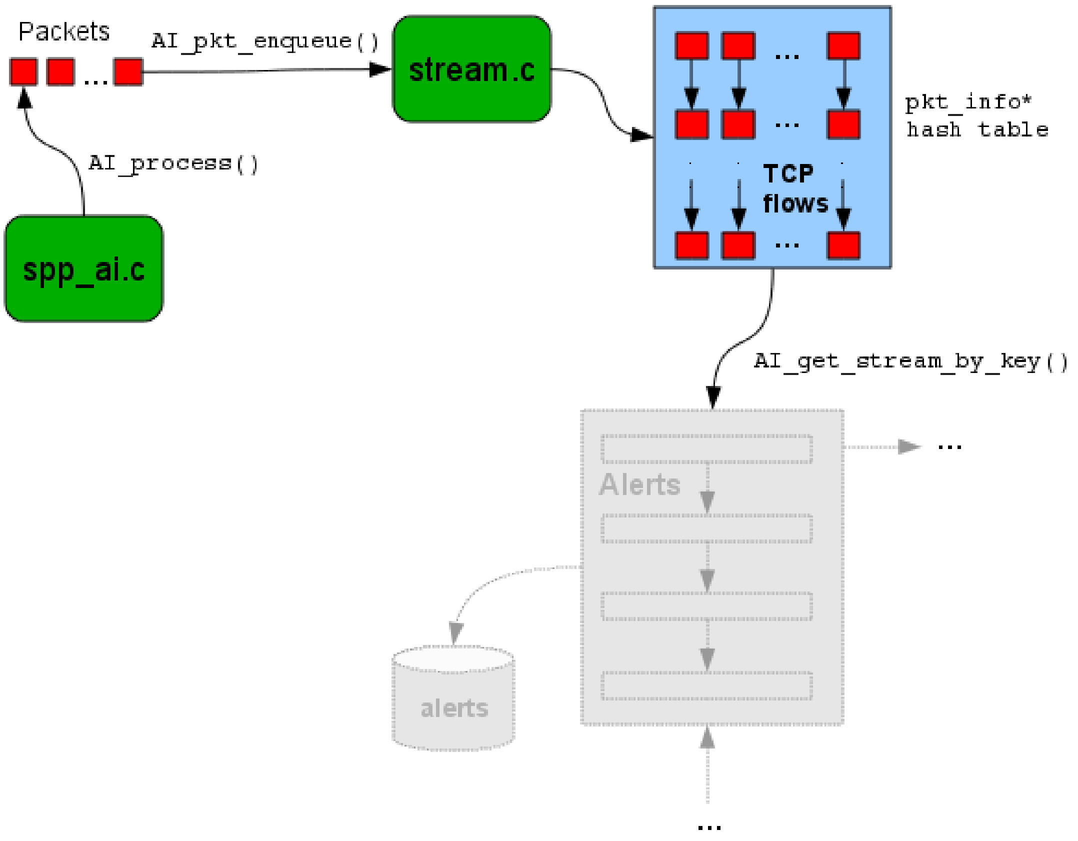

Figure 4.2: Sequence of operations that brings from individual network packets to a table of sorted TCP streams to be integrated in the alerts raised by Snort

```
struct pkt_key {
     uint32_t src_ip;
     uint16_t dst_port;
};

struct pkt_info {
     struct pkt_key key;
     time_t timestamp;
     SFSnortPacket *pkt;
     struct pkt_info *next;
     BOOL observed;
     UT_hash_handle hh;
};
```

Every time a new alert is raised the `AI_get_stream_by_key()` method is invoked on the hash table in order to fetch, if it exists, the corresponding TCP flow identified by the pair $< src\_addr, dst\_port >$ associated to the alert. If this stream exists, this is integrated in the `AI_snort_alert*` object representing the alert itself (see the file `spp_ai.h` for the implementation of this structure), and the stream itself is marked as "observed" (boolean `observed` flag in `pkt_info` structure), meaning that it should not be removed from the hash table as long as the associated alert is allocated. A thread is then periodically invoked on the hash table, with the purpose of removing the streams that

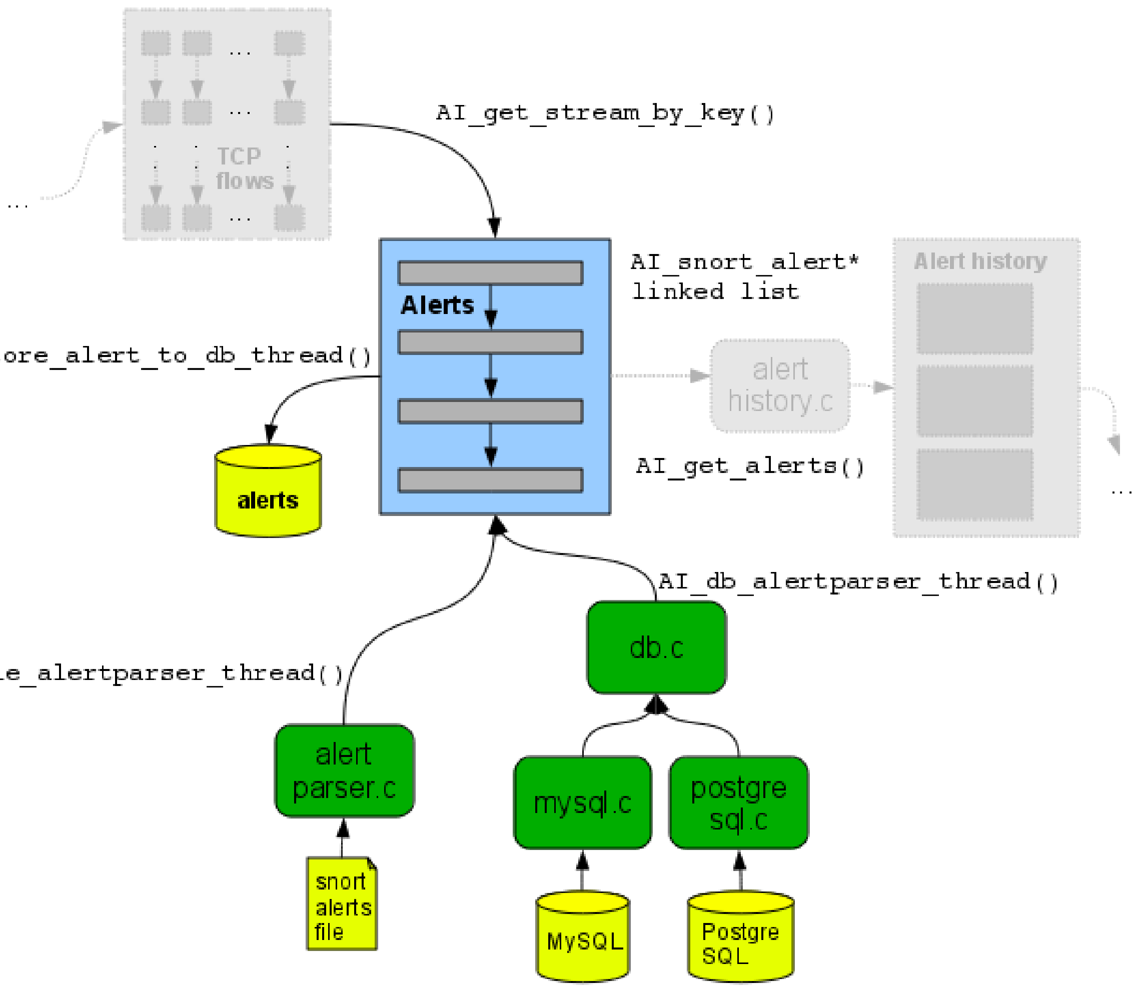

Figure 4.3: Sequence of operations for parsing the alerts both from Snort's plain text `alerts` file and Snort's databases, integrating them with TCP stream information and finally storing them to an output database

1. Have not been marked as *observed*, i.e. are not associated to a security alert;

2. Have received a TCP packet with the flags `RST` or `FIN` set, meaning that this TCP communication has been terminated;

3. Have received their more recent packet at a timestamp t such that

$$current\_timetime - t \geq \texttt{tcp\_stream\_expire\_interval} \qquad (4.1)$$

   see the configuration option `tcp_stream_expire_interval`.

This thread is executed at constant intervals specified by the `hashtable_cleanup_interval` option.

4.3 Alert parsing and database management

The alerts are periodically read from the Snort's `alert` plain-text file using a built-in parser (see the function `AI_file_alertparser_thread()` in the file `alert_parser.c`), or from Snort's alert databases, if available and the `database` option in the configuration file has been properly set (see the function `AI_db_alertparser_thread()` in the file `db.c`). These two threads are mutually exclusive for avoiding duplicated alerts (if the user decides to parse the alerts from the plain-text log then he cannot parse them from the database log and vice versa). The database is re-read for new alerts each `database_parsing_interval` seconds, while the alert file is just read when a new alert is appended using *inotify* [31] Linux kernel's mechanism for placing a watch on the file that triggers the calling process only when some other process has performed a write operation on that file.

Every time an alert is received the `pkt_info` hash table managed in `stream.c` is queried through the function `AI_get_stream_by_key()` for checking if a TCP stream exists for the pair $< src\_addr, dst\_port >$, and, if so, that stream is integrated in the `AI_snort_alert` structure that models the current alert. Moreover a thread is executed every `alert_serialization_interval` seconds for serializing the current alerts information on a binary history file used by the correlation modules downstream for correlating acquired alerts. This thread is invoked at fixed intervals or when a buffer containing the acquired alerts is full (it contains `alert_bufsize` items).

The current linked list of alerts represented by an `AI_snort_alert*` object can be returned using the `AI_get_alerts()` function, both from the file `alert_parser.c` (for alert parsing from the plain text log) and `db.c` (for alert parsing from the database). A mutual exclusion (*mutex*) mechanism is used in order to guarantee that one thread per time can access the linked list of alerts.

A high level interface is specified for interacting with the database in a transparent way, regardless of the DBMS running at the bottom level. When the module is compiled, it can be configured, via `./configure` script, for interacting with a specific database (so far *MySQL* and *PostgreSQL* are supported), by specifying, for example, the options `--with-mysql` or `--with-postgresql`. These options are mutually exclusive, i.e. a module configured for operating with MySQL cannot operate also with PostgreSQL. In fact, the transparent interface is built by defining, according to the chosen DBMS, some macros that override the ones from the library for interacting with that DBMS, for example

```
#ifdef HAVE_LIBMYSQLCLIENT

#include <mysql/mysql.h>
#define DB_init mysql_do_init
#define DB_query mysql_do_query
#define DB_fetch_row mysql_fetch_row
...
#elif HAVE_LIBPQ

#include <postgresql/libpq-fe.h>
#define DB_init postgresql_do_init
#define DB_query postgresql_do_query
#define DB_fetch_row postgresql_fetch_row
...

#endif
```

The implementation of these "wrapper" functions is defined, for example, in files `mysql.c` and `postgresql.c`. After these functions have been defined, and the macros

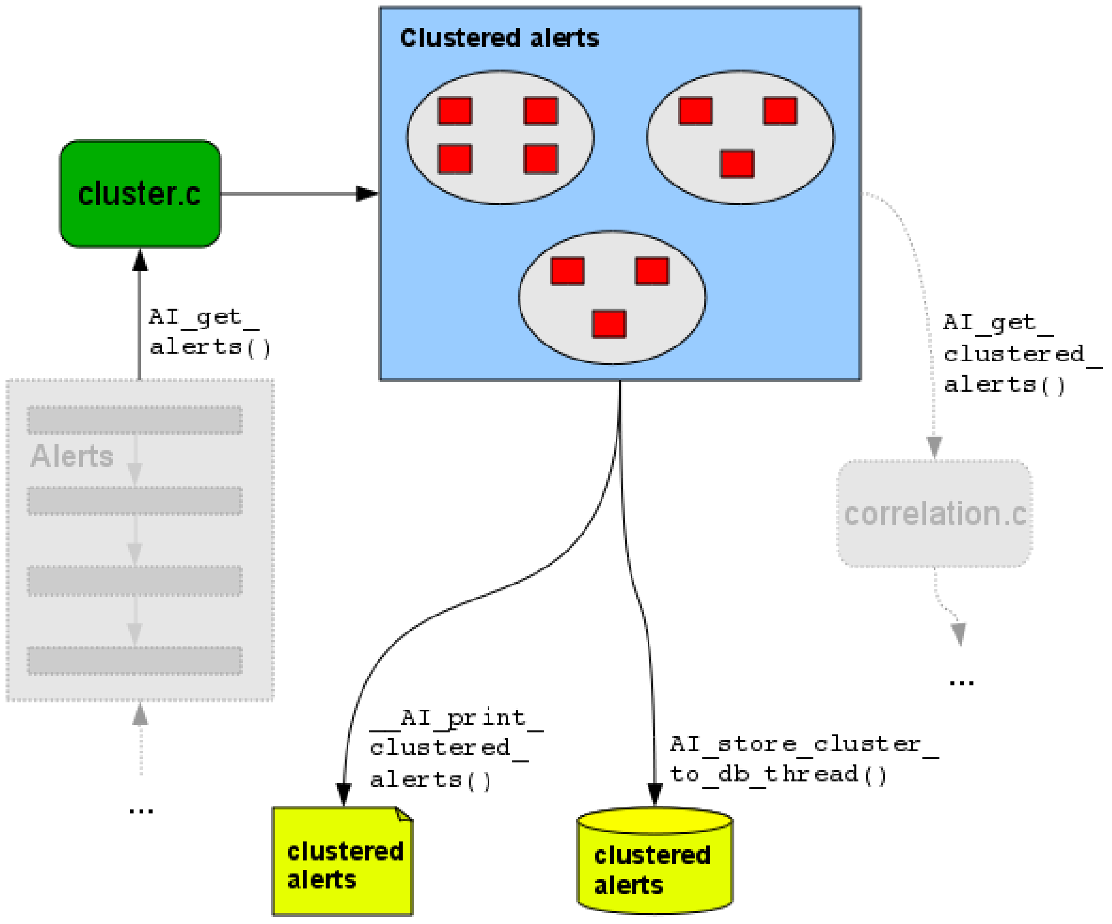

Figure 4.4: Sequence of operations performed by the clustering module

that call the right function according to the DBMS for which the module has been configured are well defined too, the operations on the database can be performed simply by calling, for example, the functions `DB_init()` or `DB_query()` regardlessly of the DBMS running.

Finally the thread `AI_store_alert_to_db_thread()` in `outdb.c` has the purpose of storing the information about the acquired alerts and the associated network streams to a database, if specified by the configuration option `output_database`. This information is then integrated downstream with the one coming from the clusterization and correlation processes.

4.4 Clustering logic

The alerts are periodically clustered by a clustering thread using the algorithm suggested in fig. 2.2. When the software is started the clustering hierarchies are built as well through the algorithm shown in fig. 3.3, obtaining the trees representing the hierarchies for a certain attribute like shown in fig. 3.2. This is done by the function `AI_hierarchies_build()` in `cluster.c`, that then starts the `AI_cluster_thread()` for periodically clustering, each `alert_clustering_interval` seconds, the alerts provided by the *alert* module through the function `AI_get_alerts()`. The clustered alerts will be printed on the log file specified by the `clusterfile` configuration option, in a Snort-like syntax, through the function `AI_print_clustered_alerts()` in `cluster.c`.

The logical steps are shown in fig. 4.4. An example extract from the file containing the information over the alert clusters is the following:

```
[**] [1:469:4] ICMP PING NMAP  [**]
[Priority: 2]
[Grouped alerts: 3] [Starting from: Tue Oct 26 19:49:22 2010]
[10.8.0.0/24] -> 10.8.0.6

[**] [122:1:0] (portscan) TCP Portscan  [**]
[Priority: 3]
[Grouped alerts: 2] [Starting from: Tue Oct 26 19:49:22 2010]
10.8.0.1 -> [10.8.0.0/24]

[**] [1:1418:13] SNMP request tcp  [**]
[Priority: 2]
[Grouped alerts: 1] [Starting from: Tue Oct 26 19:49:24 2010]
10.8.0.1:46832 -> 10.8.0.6:161

[**] [1:1421:13] SNMP AgentX/tcp request  [**]
[Priority: 2]
[Grouped alerts: 1] [Starting from: Tue Oct 26 19:49:24 2010]
10.8.0.1:46832 -> 10.8.0.6:705

[**] [1:1420:13] SNMP trap tcp  [**]
[Priority: 2]
[Grouped alerts: 1] [Starting from: Tue Oct 26 19:49:26 2010]
10.8.0.1:46832 -> 10.8.0.6:162
```

If the `output_database` configuration option is specified, a concurrent thread running in function `AI_store_cluster_to_db_thread()` in file `outdb.c` also stores the cluster information on the output database, whose structure will be analyzed later. These two functions use a *mutex* mechanism for avoiding to store the information when another thread is still modifying them.

4.5 History management

The history of acquired alerts, grouped by alert type, is held in a binary file whose path is specified in `alert_history_file` configuration option. This file is used by the correlation algorithms downstream, in particular the one computing the correlation index among two alerts using a temporal *pseudo*-bayesian network. The logical steps performed by the algorithm are shown in fig. 4.5.

The alert history is serialized as hash table on the binary history file according to the following structure:

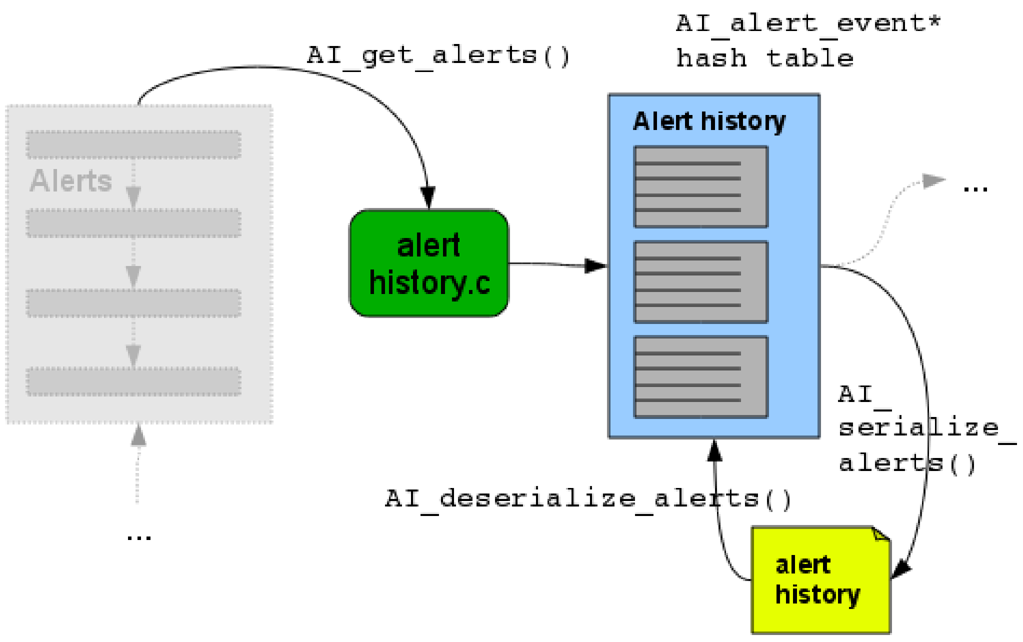

Figure 4.5: Sequence of operations performed by the history management module

```
typedef struct {
     int gid;
     int sid;
     int rev;
} AI_alert_event_key;

typedef struct _AI_alert_event {
     AI_alert_event_key key;
     unsigned int count;
     time_t timestamp;
     struct _AI_alert_event *next;
     UT_hash_handle hh;
} AI_alert_event;
```

where the key represented by the tuple $\{gid, sid, rev\}$ is the identifier that Snort assigns to an alert type (ICMP ping, port scan, shellcode, ...) and *count* represents the number of alerts having that type. The file will be then represented as a sequence like

```
gid1 sid1 rev1 count1 timestamp11 timestamp12 timestamp13 ...
gid2 sid2 rev2 count2 timestamp21 timestamp22 timestamp23 ...
...
gidn sidn revn countn timestampn1 timestampn2 timestampn3 ...
```

The module loads the history information saved on the binary file into a `AI_alert_-event` type hash table when loaded through the function `AI_deserialize_alerts()` in `alert_history.c`. After this operation this structure is held in memory and, unless a thread explicitely deallocates it through the function `AI_alerts_hash_free()`, the binary history file will not be parsed anymore. The thread that acquires new alerts can then send to this module a set of alerts to be serialized on the history file through the function `AI_serialize_alerts()`. This usually happens when the buffer containing the alerts to be serialized in `alert_parser.c` is full (it contains at least `alert_bufsize`

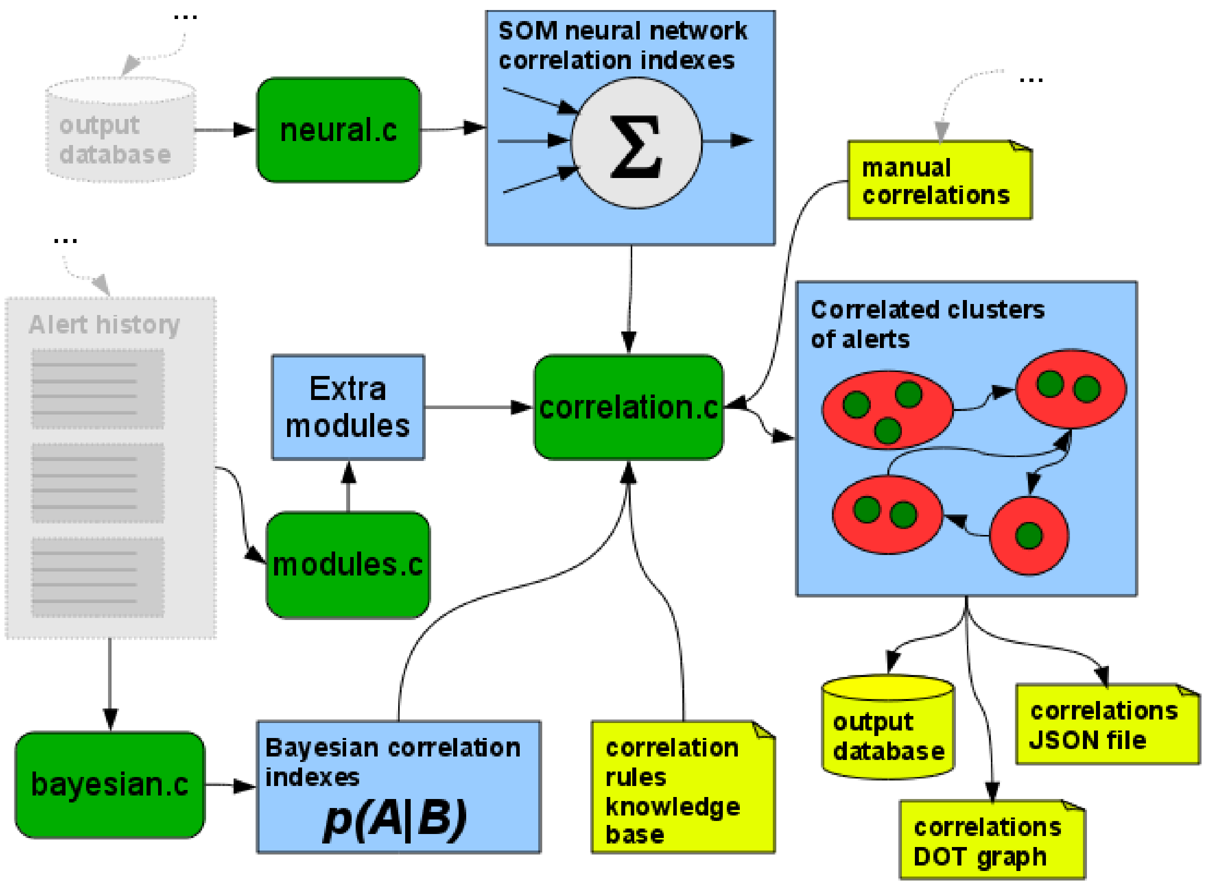

Figure 4.6: Logic of the correlation module

elements) or the *serialization timeout* specified in `alert_serialization_interval` has been reached.

4.6 Correlation engine

The correlation engine is contained in file `correlation.c` and it contains the logic for computing the correlation coefficients between pairs of alert clusters as the weighted mean value of several correlation indexes through the equation shown in eq. 3.6. The logical schema of the correlation module is shown in fig. 4.6. The correlation values are held in a hash table where the key is a pair of alerts (an alert history containing n alerts will generate a correlation hash table whose size is $n(n-1)$) defined like:

```
typedef struct {
     AI_snort_alert *a;
     AI_snort_alert *b;
} AI_alert_correlation_key;

typedef struct {
     AI_alert_correlation_key key;
     double correlation;
     UT_hash_handle hh;
} AI_alert_correlation;
```

```
 <?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hyperalert[
<!ELEMENT hyperalert (snort-id, pre, post, desc)>
<!ELEMENT snort-id (#PCDATA)>
<!ELEMENT desc (#PCDATA)>
<!ELEMENT pre (#PCDATA)>
<!ELEMENT post (#PCDATA)>
]>
```

Figure 4.7: Data Type Definition (*DTD*) for the XML format that expresses the knowledge base for a certain hyperalert

In the following sections we are going to examine how the several correlation indexes are computed.

4.6.1 Hyperalert knowledge base correlation

The first correlation index between two alerts is computed through the mechanism described in section 3.3.1, based on a knowledge base provided by the software itself, by third parts or by the user containing information on how an alert is structured, which are the preconditions needed for it to be raised and which can be its consequences. This knowledge consists in a set of *XML* files provided in directory `correlation_rules_dir` which express the set of preconditions necessary for a certain alert type to be raised and the consequences potentially triggered by this alert. The Data Type Definition (*DTD*) for this XML file is shown in fig. 4.7.

The XML that describes a hyperalert according to this DTD is structured as it follows:

- The field `snort-id` represents the ID that Snort associates to that alert type (in format *gid.sid.rev*), the XML file should be named after this ID as well;
- The field `description`, not mandatory, contains a human-readable description of that hyperalert;
- The `pre` tag contains a precondition needed for that alert, so each file can have one, more or no `pre` tag, each tag containing a precondition;
- The `post` tag contains a consequence that may be true after that alert; each XML can contain one, more or no `post`, despite it makes sense to have at least one consequence or more in every hyperalert model, since a hyperalert model without specified consequences is hard to “connect” to other alerts.

The tags `pre` and `post` take as argument a function name and its parameters, according to the format

```
FunctionName(argument1, argument2, ...)
```

The syntax of a function name and its arguments can be freely decided by the user, provided that this syntax will be used in all the XML hyperalert models for that function name otherwise the preconditions-consequences matching will fail.

The following macros are allowed as well:

- `+SRC_ADDR+`: representing the address that triggered the alert;
- `+DST_ADDR+`: representing the target address of the alert;
- `+ANY_ADDR+`: identifying any IP address;
- `+SRC_PORT+`: identifying the source port of the alert, if it exists;
- `+DST_PORT+`: identifying the destination port of the alert, if it exists;
- `+ANY_PORT+`: identifying any TCP/UDP port.

IP addresses and ports can also be written as ranges, for example *192.168.1.0/24* or *1-1024*. These macros and ranges will be expanded at runtime for matching pairs of predicates.

Two examples, one representing the knowledge that models an alert of type *ICMP ping* and the other one modelling an alert of type *TCP portscan*, can be the following (both available by default in directory `corr_rules`):

```
<!-- 1-469-4.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hyperalert PUBLIC
   "-//blacklight//DTD HYPERALERT SNORT MODEL//EN"
   "http://0x00.ath.cx/hyperalert.dtd">

<hyperalert>
  <snort-id>1.469.4</snort-id>
  <desc>ICMP PING NMAP</desc>
  <post>HostExists(+DST_ADDR+)</post>
</hyperalert>


<!-- 122-1-0.xml -->

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hyperalert PUBLIC
   "-//blacklight//DTD HYPERALERT SNORT MODEL//EN"
   "http://0x00.ath.cx/hyperalert.dtd">

<hyperalert>
  <snort-id>122.1.0</snort-id>
  <desc>(portscan) TCP Portscan</desc>
  <pre>HostExists(+DST_ADDR+)</pre>
  <post>HasService(+DST_ADDR+, +ANY_PORT+)</post>
</hyperalert>
```

The file modelling the port scan attack, for example, should be named `122-1-0.xml`, since the format for the correlation rules files is `gid-sid-rev.xml`.

In the example above, the function *HasService* should always take two parameters, an IP address as first one and a port number as second one, in all of its occurrences, *HasService(10.8.0.1, 80)* and *HasService(+DST_ADDR+, +ANY_PORT+)* for example are valid representations, while *HasService(1234)* will not be considered the same function used above. This happens because the syntax and the semantic of each predicate is free, but if one format is chosen for a certain predicate, that format should be used in all the occurrences of that predicate in order to avoid matching failures, and also no parameter overloading is allowed.

The correlation engine will recognize, for example, a match between the predicates *HasService(+DST_ADDR+, +ANY_PORT+)* and *HasService(192.168.1.2, 80)*, since the latter one is a logical subset deducted by the first one, same for *HostExists(192.168.1.0/24)* and *HostExists(192.168.1.3)*.

The function `__AI_hyperalert_from_XML()` in `correlation.c` has the purpose, given an alert ID, to fetch the corresponding XML file, if exists, that models its preconditions and consequences, and keep it in memory inside of a hash table modelled as

```
typedef struct {
     unsigned int gid;
     unsigned int sid;
     unsigned int rev;
} AI_hyperalert_key;

typedef struct {
     AI_hyperalert_key key;
     char **preconds;
     unsigned int n_preconds;
     char **postconds;
     unsigned int n_postconds;
     UT_hash_handle hh;
} AI_hyperalert_info;
```

The parser in this thread uses *libxml2* for parsing the XML files, and the parsed information is held in a `AI_hyperalert_info` hash table so that it does not need to be parsed again. The acquired pair of alerts are then compared and, for those having a hyperalert description stored in the hash table, their preconditions and consequences are compared. This action is performed in function `__AI_kb_correlation_coefficient()`, that takes two alert descriptions for which a hyperalert model exists in the hash table as arguments and then, for each pair $\{precondition, consequence\}$,

1. Checks if the names of the two functions match (if not, go on with the next pair) through the function `__AI_get_function_name()`;

2. Extracts the arguments of each of the two functions (stopping if the number of arguments is not the same) through the function `__AI_get_function_arguments()`;

3. Expands of the macros, *+SRC_ADDR+*, *DST_ADDR*, *+SRC_PORT+* and *+DST_PORT* will be replaced by the corresponding values associated to that alert; this operation is performed in function `__AI_macro_subst()`;

4. Compares the pairs of arguments, expanding each of them if it represents a macro or an IP/port range. For example, if the i-th argument of the function is a port and this argument has value *+ANY_PORT+* or $1 - 1024$ for the first alert and 80 for the second one, a match will be counted. If the two predicates have a generic i-th argument that is not matched, the algorithm is stopped.

If this algorithm reaches the end, the specified pair of predicates includes two equal items, and the counter of equal predicates will be increased by one. For each pair of alerts $\{A_1, A_2\}$ each pair of preconditions and consequences is compared using this algorithm, and we express the knowledge base correlation index between A_1 and A_2, i.e. the probability that A_1 *prepares* A_2 through the relation already analyzed in section 3.3:

$$corr(A_1, A_2) = 2\frac{|Pre(A_2) \cap Post(A_1)|}{|Pre(A_2) \cup Post(A_1)|} \tag{4.2}$$

4.6.2 *Pseudo*-bayesian correlation

The file `bayesian.c` includes the implementation for the algorithm discussed in chapter 3. In particular, the implementation holds a cache as hash table containing each pair of raised alerts. When a pair of alerts is not found in this cache, the algorithm queries the history file through the method `AI_get_alert_by_key()`, obtaining the list of alerts of type a and type b already occurred and their timestamps. For each pair of alert types $\{a, b\}$ the correlation value is computed through the method suggested in chapter 3 and stored in the hash table. If the correlation value for the pair $\{a, b\}$ was already computed and is already in the hash table, then it does not need to be computed again. The hash table is periodically *refreshed* each `bayesian_correlation_cache_validity` seconds. After this time interval we consider the correlations computed on the latest alert history as obsoleted and they need to be computed again.

4.6.3 Neural network correlation

The file `neural.c` contains the implementation for the correlation mechanism proposed in section 3.3.3. The *SOM* is managed through the mini-library `fsom` [41], a C library developed for the purpose of this software but re-usable in many applications for the management of Self-Organizing Maps.

The concurrent thread `AI_neural_thread()`, executed each `neural_network_training_-interval` seconds, has the purpose of training the network basing on the alerts received since the last training execution. These alerts are read from the output database, converted to a valid dataset (i.e. a vector of vectors of numerical values) through the function `__AI_alert_to_som_data()` and then used for training the network through the function `som_train()` contained in `fsom/fsom.c`. The current representation of the network is then saved to the binary file `netfile` through the function `som_-serialize()`. The same file will be deserialized on the next execution of the software through the method `som_deserialize()` so that the knowledge over the training phases of the neural network is kept among different executions of the software.

After the training phase, each alert is presented to the neural network, converted to a numerical valid dataset and mapped on the output layer, through the algorithm illustrated in section 3.3.3. The correlation between each pair of alerts is then computed as Euclidean or Manhattan distance between the points on the output layer where these alerts are mapped.

4.6.4 Correlation clustering

As shown in section 3.3.4 it is possible, by setting the option `neural_clustering_interval` to a non-zero value to perform the clustering over the output layer of the *SOM* neural network in order to attempt to find alerts or clusters of alerts belonging

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE clusters[
   <!ELEMENT cluster (alert)*>
   <!ELEMENT alert EMPTY>

   <!ATTLIST cluster id CDATA \#REQUIRED>
   <!ATTLIST alert
       desc CDATA
       gid CDATA #REQUIRED
       sid CDATA #REQUIRED
       rev CDATA #REQUIRED
       src_ip CDATA #REQUIRED
       src_port CDATA
       dst_ip CDATA #REQUIRED
       dst_port CDATA
       xcoord CDATA
       ycoord CDATA
    >
]>
```

Figure 4.8: DTD model for the XML file representing the clusters of alerts grouped from the *SOM* output layer

to the same attack scenario. In order to do this the code uses a structure named `AI_alerts_per_neuron` as hash table. The key of this table is the coordinates of the point on the *SOM*'s output layer, the content is the list of alerts or clusters of alerts mapped on these coordinates. The coordinates of the neurons having at least one associate alert are then clustered using k-means. A small C library, *fkmeans* [42], has been developed for this purpose, and it is also re-usable in other contexts since it was designed as a library for clustering any set of data by any dimension using k-means. The source code is contained in the directory `fkmeans` and the implementation is in the file `fkmeans/kmeans.c`. The numerical set containing the coordinates is passed as argument to the function `kmeans_auto()`, that performs the k-means clustering for each $1 \leq k \leq N$ and returns the configuration of the clusters having the minimal Schwarz [36] index (i.e. the one best representing the dataset). The alerts associated to the pairs of coordinates in the dataset are then clustered as well, and the resulting clusters are then stored on an XML file saved in `webserv_dir/neural_clusters.xml`. The DTD specification for this XML file is shown in fig. 4.8.

4.6.5 Manual correlations

It is possible, through the web interface provided by the software, to manually set two pairs of clustered alerts as *correlated* if they were not set or *not correlated* if they were set as correlated but they actually are not. These explicit correlations or *un*-correlations will be held in two XML files, ontained in `webserv_dir/manual_correlations.xml` and `webserv_dir/manual_uncorrelations.xml` respectively. The *DTD* for these files is shown in fig. 4.9. An example can be the following:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE correlations[
    <!ELEMENT correlations (correlation)*>
    <!ELEMENT correlation (from, to)>
    <!ELEMENT from EMPTY>
    <!ELEMENT to EMPTY>

    <!ATTLIST from
        sid CDATA #REQUIRED
        gid CDATA #REQUIRED
        rev CDATA #REQUIRED
    >

    <!ATTLIST to
        sid CDATA #REQUIRED
        gid CDATA #REQUIRED
        rev CDATA #REQUIRED
    >
]>
```

Figure 4.9: *DTD* for the manual correlations and *un*-correlation XML files

```
<!DOCTYPE hyperalert PUBLIC
   "-//blacklight//DTD MANUAL CORRELATIONS//EN"
   "http://0x00.ath.cx/manual_correlations.dtd">

<correlations>
  <correlation>
    <from gid="122" sid="1" rev="0"/>
    <to gid="1" sid="469" rev="4"/>
  </correlation>
</correlations>
```

These files are auto-generated by the web interface and they have the maximum priority over all the other correlation indexes, this means that two alerts will not be marked as correlated even if their correlation index is high when their Snort identificators are marked as not correlated in `manual_uncorrelations.xml`. Vice versa two alerts will always be marked as correlated even when their correlation index is low when their correlation is contained in `manual_correlations.xml`.

These files are periodically parsed each `manual_correlations_parsing_interval` seconds by the thread `__AI_manual_correlations_parsing_thread()` in `correlation.c`, using *libxml2*, and the parsing results are then stored in two hash tables, `manual_correlations` and `manual_uncorrelations`.

4.6.6 Extra correlation indexes

The correlation engine is designed as a modular and extensible *framework* that, beyond the correlation mechanisms already provided by default, offers the possibility to the user to use more correlation modules. These modules will be shared object files (extension `.so`) contained in directory `corr_modules_dir` that should be compiled from this directory adding an entry to the sample `Makefile` here contained, for example

```
gcc -I. -I.. -I../uthash -I../fsom -I../base64 -I../include \
  -D_GNU_SOURCE -Wall -pedantic -pedantic-errors -std=c99 \
  -fPIC -shared -rdynamic -o libsf_ai_corr_index_new.so \
  libsf_ai_corr_index_new.c
```

In order to be correctly loaded a new correlation index should

- Include the file `"spp_ai.h"` from the upper directory;
- Contain a function named `AI_corr_index()` that, given two alerts pointers as `const AI_snort_alert*` structures, returns a `double` value that represents their correlation index;
- Contain a function named `AI_corr_index_weight()` taking no argument and returning a `double` value that represents the weight of that correlation index in a certain moment.

An example module is provided in `libsf_ai_corr_example.c` file in the directory `corr_modules`. This file has a correlation function that always returns 0.5 for each pair of alerts and always returns a null weight, so it's completely harmless.

The extra modules will be read from the extra modules directory when the software starts, and the corresponding symbols of the needed functions will be extracted using `dlsym()` function in `dlfcn.h`. These functions will be then saved in two arrays of functions respectively returned by the functions `AI_get_corr_functions()` and `AI_get_corr_weights()`, and each of these will be a correlation index inserted into the average mean computation in `correlation.c`

4.7 Output database

If the option `output_database` is specified in the configuration, the information about alerts, packet streams, clusters and correlations will be saved on a database as well. This strategy is strongly suggested as the database is much easier to analyse and parse than huge log files, and some correlation indexes like the neural one require the presence of alerts and cluster of alerts stored on the database in order to work properly.

The *Entity-Relationship* (*E/R*) schema of this database is shown in fig. 4.10.

For using this database it is needed to:

- Compile the module with the support for a DBMS (off by default), passing to the `./configure` script the option `--with-mysql` or `--with-postgresql` (the support for more DBMSs will be implemented in future);
- Create the database that will hold the module information, for example

  ```
  $ mysql -uuser -ppass -e "create database snort_ai_module"
  $ psql -U user -c "create database snort_ai_module"
  ```

- Create the tables for the database using the SQL files in `schemas/` directory, for example

  ```
  $ mysql -uuser -ppass snort_ai_module < schemas/mysql.sql
  $ psql -U user -W snort_ai_module < schemas/postgresql.sql
  ```

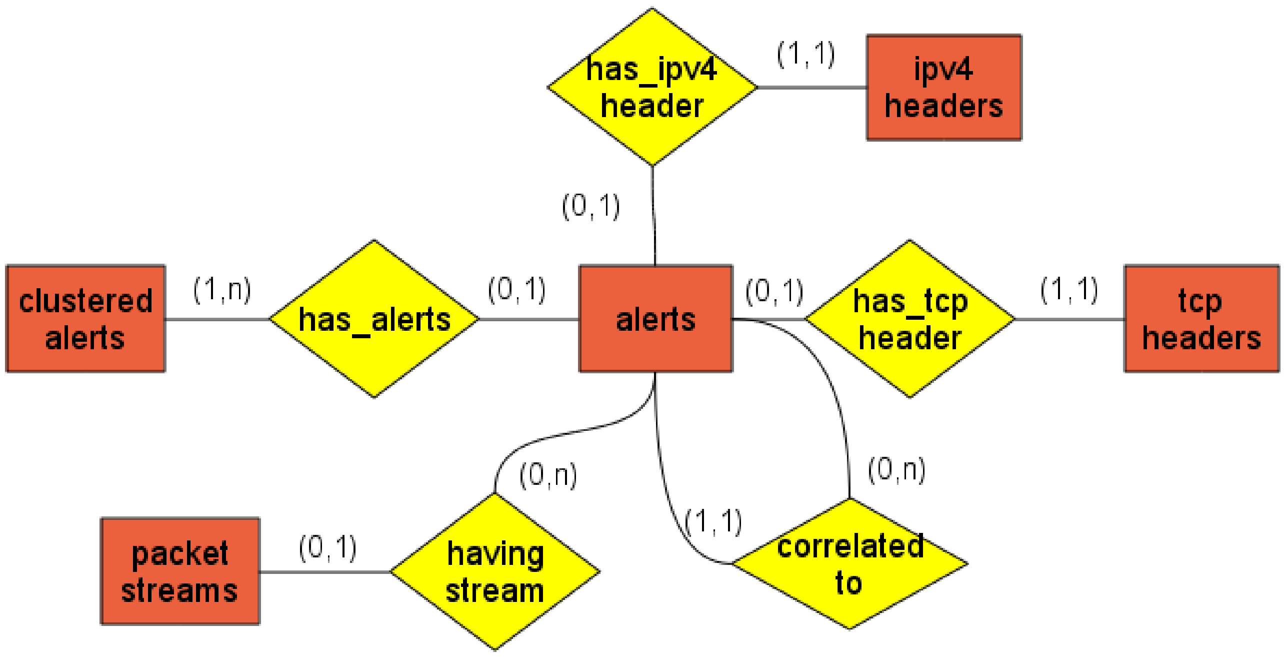

Figure 4.10: *Entity-Relationship* (*E/R*) schema of the output database for storing alerts, clusters, correlations and packet streams

The structure of the database is the following (file `mysql.sql`):

```
DROP TABLE IF EXISTS ca_ipv4_headers;
CREATE TABLE ca_ipv4_headers (
   ip_hdr_id      integer     auto_increment,
   ip_tos         integer,
   ip_len         integer,
   ip_id          integer,
   ip_ttl         integer,
   ip_proto       integer,
   ip_src_addr    varchar(32),
   ip_dst_addr    varchar(32),

   primary key(ip_hdr_id)
);

DROP TABLE IF EXISTS ca_tcp_headers;
CREATE TABLE ca_tcp_headers (
   tcp_hdr_id      integer    auto_increment,
   tcp_src_port    integer,
   tcp_dst_port    integer,
   tcp_seq         integer,
   tcp_ack         integer,
   tcp_flags       integer,
   tcp_window      integer,
   tcp_len         integer,

   primary key(tcp_hdr_id)
```

```
);

DROP TABLE IF EXISTS ca_clustered_alerts;
CREATE TABLE ca_clustered_alerts (
   cluster_id        integer      auto_increment,
   clustered_srcip   varchar(255) default null,
   clustered_dstip   varchar(255) default null,
   clustered_srcport varchar(255) default null,
   clustered_dstport varchar(255) default null,

   primary key(cluster_id)
);

DROP TABLE IF EXISTS ca_alerts;
CREATE TABLE ca_alerts (
   alert_id       integer     auto_increment,
   gid            integer,
   sid            integer,
   rev            integer,
   priority       integer,
   description    varchar(255),
   classification varchar(255),
   timestamp      datetime,
   ip_hdr         integer default 0,
   tcp_hdr        integer default 0,
   cluster_id     integer default 0,

   primary key(alert_id),
   foreign key(ip_hdr) references ca_ip_headers(ip_hdr_id),
   foreign key(tcp_hdr) references ca_tcp_headers(tcp_hdr_id),
   foreign key(cluster_id) references ca_clustered_alerts(cluster_id)
);

DROP TABLE IF EXISTS ca_packet_streams;
CREATE TABLE ca_packet_streams (
   pkt_id         integer     auto_increment,
   alert_id       integer,
   pkt_len        integer,
   timestamp      datetime,
   content        longblob,

   primary key(pkt_id),
   foreign key(alert_id) references ca_alerts(alert_id)
);

DROP TABLE IF EXISTS ca_correlated_alerts;
CREATE TABLE ca_correlated_alerts (
   alert1            integer,
   alert2            integer,
   correlation_coeff double,
   primary key(alert1, alert2),
   foreign key(alert1) references ca_alerts(alert_id),
   foreign key(alert2) references ca_alerts(alert_id)
);
```

This schema is guaranteed to be in *BCNF* [26] normal form.

So in order to fetch the pairs of correlated (clusters of) alerts having a correlation index above a certain threshold we simply need to issue the following query:

```
SELECT a1.*, a2.*
FROM ca_correlated_alerts c JOIN ca_alerts a1 JOIN ca_alerts a2
ON c.alert1=a1.alert_id AND c.alert2=a2.alert_id
WHERE correlation_coeff > 0.5
```

Or, if we want to select all the alerts newer than a certain threshold and also their TCP/IP associated information (this query is executed in `neural.c` for fetching the alerts triggered after the latest training phase and also their network details), we can execute the following:

```
SELECT a.*, ip.*, tcp.*
FROM (ca_alerts a LEFT JOIN ca_ipv4_headers ip
   ON a.ip_hdr=ip.ip_hdr_id)
LEFT JOIN ca_tcp_headers tcp
ON a.tcp_hdr=tcp.tcp_hdr_id
WHERE unix_timestamp(a.timestamp) > t
```

4.8 Graph interface

The correlations between clusters of alerts will be stored by the module, through the function `__AI_correlated_alerts_to_dot()` in file `correlation.c`, in a file in *DOT* language [27] that models it as a graph where each node represents a cluster of alerts and each edge represents a relation of correlation. An example of graph expressed in this language can be the following:

```
digraph G  {
    "ICMP ping nmap" -> "TCP portscan";
    "TCP portscan" -> "Shellcode NOOP execution";
    "TCP portscan" -> "C99 shell RFI attempt";
    "TCP portscan" -> "SSH login attempt failure";
    "Shellcode NOOP execution" -> "RPC mountd export request";
}
```

This file, saved in `correlated_alerts_dir/correlated_alerts.dot`, will be periodically updated every time a new correlation is found.

The library `graphviz` [39] (an optional dependency for this software) allows to generate from this file, through the function `gvRenderFilename()`, a visual graph, in PostScript and PNG formats (these files will be respectively placed in `correlated_alerts_dir/correlated_alerts.ps` and `correlated_alerts_dir/correlated_alerts.png`). An example visualization of the graph generated from a *DOT* file is shown in fig. 4.11. If `graphviz` is not available or installable on the system (`apt-get install`

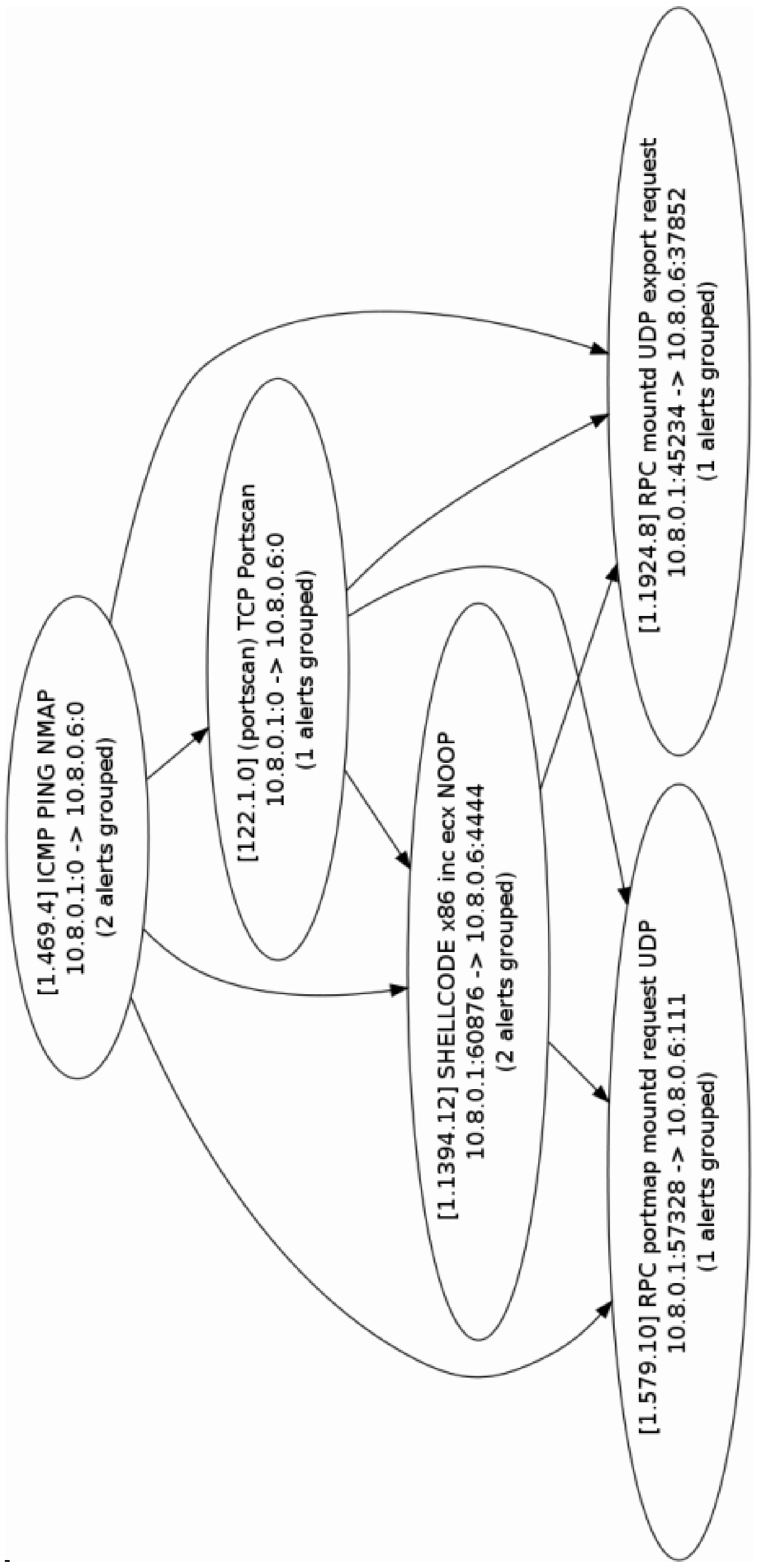

Figure 4.11: Example of a correlation graph generated by the module using the library *graphviz*'s *DOT* renderer

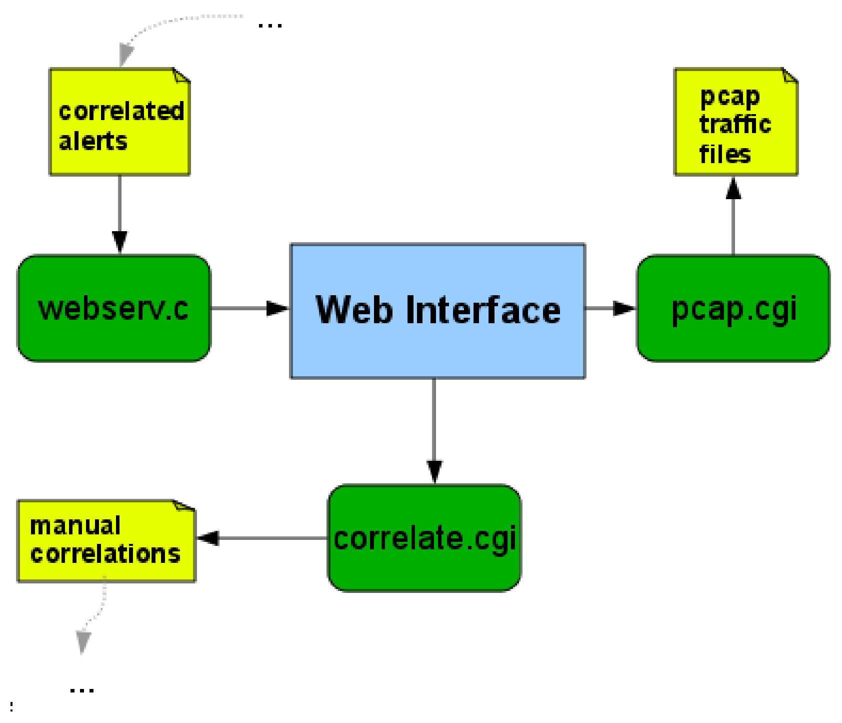

Figure 4.12: Schema of the parts of the module's web interface

`libgraphviz-dev` on a Debian-based system) this dependency can be "switched off" using the option `--without-graphviz` to the `./configure` script, but doing so it would be necessary to use external software or renders for obtaining a visual representation of the *DOT* file.

4.9 Web interface

A tiny web server named *up* (from the name of the quark particle that constitutes protons and neutrons), whose implementation is contained in file `webserv.c`, has been developed with the purpose of:

- Providing an intuitive and simple interface to the user in order to explore and analyse alert clusters and their correlations, and also the ability to interactively change the visualization;
- Providing the capability to analyze the contents of single alerts inside of alert clusters, by offering the possibility to select an alert inside of a cluster and save the traffic content associated to it, if available, in PCAP format;
- Providing the possibility to modify on runtime wrong correlations or specify correlations not found;
- Providing the possibility to easily view the alerts and clusters of alerts marked as "clustered" after performing the k-means clusterization over the output layer of the *SOM* neural network;
- Providing the possibility to "locate" on a geographical map an IP that raised an alert, if the geographical information for that IP is available.

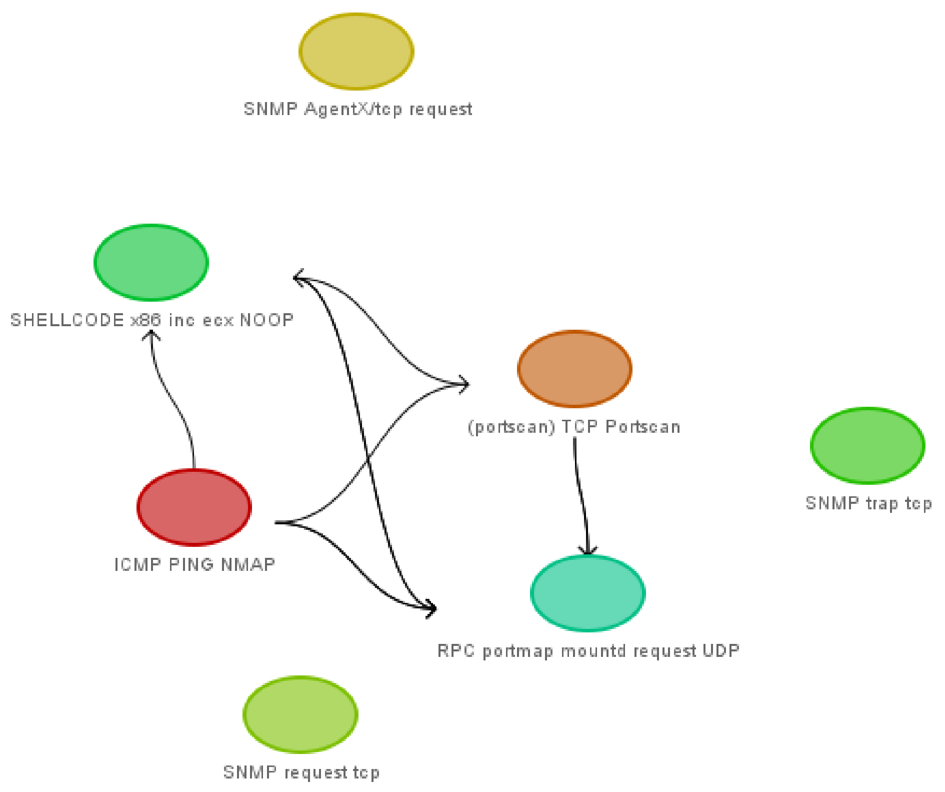

Figure 4.13: Representation of the alert correlation graph through the web interface

This web server supports most of the common *HTTP/1.1* methods (GET, POST and HEAD), it is backward compatible with *HTTP/1.0* requests as well, it supports cookies and most of the common HTTP headers sent by the client or explicitely set by a *CGI* script or application (*Content-Length, Content-Type, Accept, Connection, Reason, User-Agent, Referrer, ...*) setting them as environment variables for the downstream *CGIs*, and it supports most of the common mime types, defaulting them to *application/extension* when the extension type is unknown or *text/plain* when the file has no extension. It should support any *CGI* [33] script or application complaint to the specification 1.1, provided that it has extension `.cgi`, it is placed in the directory specified by the configuration option `webserv_dir` (all the files that we want to be publicly visible in the web server should be placed here) and it is an executable file (and, of course, in the case of CGI script the associated interpreter exists on that machine). Each client request generates a new *server* thread. The web server is modular and it can be reused other projects or as stand-alone web server.

The web server starts listening on the port specified by `webserv_port` configuration option (default if not specified: 7654) and can be disabled by setting this configuration option to 0. In this case the files contained in `webserv_dir` need to be copied to the directory of another web server if we want to use (strongly suggested) the web interface, and the configuration option `webserv_dir` needs to be changed as well in order to point to the location of the `htdocs` directory of this server.

After the module is started, the web interface can be simply accessed by opening through a web browser the link `http://snort-machine-name:webserv_port`, for example `http://localhost:7654`. This makes things easier when the module for Snort is executed on a machine but we want the results to be accessible also from other machines. A snapshot of the web page showing alert correlations is shown in fig. 4.13.

The representation of the alert graph through the web interface is based on the JavaScript library *Dracula* [43], a library for managing interactive graphs through JavaScript, based itself on the library *Raphaël* [44], a complex JavaScript library that simplifies the management of vectorial SVG images in a web page. Both the libraries have been modified in order to assign to each node a unique ID, make it "clickable" and capture the click events so that the information over a cluster of alerts will be expanded when clicked.

It is possible through the web interface to:

- Custome the visualization of the graph by using the "Redraw" button or performing drag'n'drop operations for moving nodes of the graph around the page area;
- Analyze the details of a single cluster of alerts, by clicking on it and viewing details over the single alerts included in it;
- Download the network traffic associated to a single alert, if it exists, in PCAP format, as a file to be analyzed by network analysis tools like *tcpdump, Wireshark* or *Snort* itself;
- Set a manual correlation between two clusters of alerts marked as not correlated, or stating that two alerts marked are correlated actually are not so, by clicking on the buttons "Manually correlate" or "Remove correlations" and then on the two alerts to explicitely correlate or mark as not correlated;
- Possibly view on *Google Maps* service the geographic location of an IP address that triggered an IDS alert, clicking, if available, on the "locate" link next to a certain source IP address.

Alert correlations are saved from the module itself through the function `__AI_correlated_alerts_to_json()` in `correlation.c`. This function saves the current alert correlation graph on file `webserv_dir/correlation_graph.json`, a *JSON* [29] file having, for example, the following format:

```
[
{
    "id": 409,
    "snortSID": "469",
    "snortGID": "1",
    "snortREV": "4",
    "label": "ICMP PING NMAP ",
    "date": "Tue Nov 16 21:45:08 2010",
    "clusteredAlertsCount": 2,
    "from": "10.8.0.1:0",
    "to": "10.8.0.6:0",
    "latitude": "0.000000",
    "longitude": "0.000000",
    "clusteredAlerts": [
        {
            "id": 414,
            "label": "ICMP PING NMAP ",
            "date": "Tue Nov 16 21:45:21 2010",
            "from": "10.8.0.1:0",
            "to": "10.8.0.6:0",
  "latitude": "0.000000",
  "longitude": "0.000000"
        }
    ],
    "connectedTo": [
        { "id": 410 },
        { "id": 418 }
    ]
},
{
    "id": 410,
    "snortSID": "1",
    "snortGID": "122",
    "snortREV": "0",
    "label": "(portscan) TCP Portscan ",
    "date": "Tue Nov 16 21:45:08 2010",
    "clusteredAlertsCount": 1,
    "from": "10.8.0.1:0",
    "to": "10.8.0.6:0",
    "latitude": "0.000000",
    "longitude": "0.000000",
    "connectedTo": [
        { "id": 418 }
    ]
},
{
    "id": 418,
    "snortSID": "1394",
    "snortGID": "1",
    "snortREV": "12",
    "label": "SHELLCODE x86 inc ecx NOOP ",
    "date": "Tue Nov 16 21:45:40 2010",
    "clusteredAlertsCount": 1,
    "from": "10.8.0.1:58213",
    "to": "10.8.0.6:4444",
    "latitude": "0.000000",
    "longitude": "0.000000",
    "packets": [
        "RQAALDOnAAAOBjWPCggAAQoIAAaZYhHNobznV".
            "QAAAABgAgQATCoAAAIEBVgAAAAAAAAAAA==",
        "RQAAKAAAQABABia6CggABgoIAAEBOZliAAAAA".
            "KG851ZQFAAAeAsAAAIEBVgAAAAAAAAAAA==",
        "RQAAPKz8QABABnmpCggAAQoIAAbjZRFcx9hmj".
            "AAAAACgAhbQIRQAAAIEBVgEAggKDszKcA==",
        "RQAANAAAQABABiauCggAAQoIAAbjZRFcx9hqd".
            "gWNhpqAEAC3dt0AAAEBCAoOzM+OAHRZCw==",
        ...
    ]
```

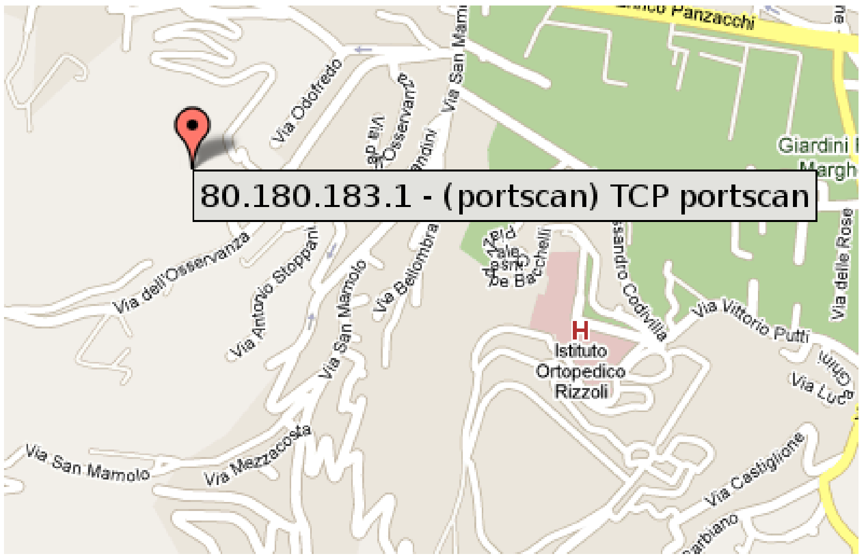

Figure 4.14: Integration between the software and *Google Maps* for identifying, using `hostip.info`, latitude and longitude, if available, of an IP that triggered an alert

```
}
]
```

Every node represents a cluster of alerts, each one identified by a unique ID number. The web interface is usable only if the module is configured with the output database, since the ID number is only assigned to an alert by the database storage process. The three numbers that Snort uses for identifying the alert type, a description label (usually the same used by Snort for identifying that type of alert), the date, and the source and destination ports and addresses of that alert. A connection between two clusters of alerts is modelled through the `connectedTo` array, that contains the identificator numbers of the cluster of alerts derived from the current cluster. If available, the sequence of packets associated to a certain alert is stored as well, in *base64* [30] in order to avoid that special characters in the binary sequences may corrupt the file. The base64 encoding is performed by the module through the functions included in the library provided in directory `base64/`, and the packets are then decoded from base64 when they are stored in a PCAP file.

Moreover, the module can attempt to locate the geographical location of an IP address that raised an IDS alert by interacting with the web site `http://hostip.info` in order to fetch the geographical coordinates (latitude and longitude), if available, associated to a given IP address through HTTP POST requests (see file `geo.c`). This information, if available, is placed in the JSON file as `latitude` and `longitude`, then used to interact with Google Maps in order to show the location. It is just needed to click on a node in the graph, and for a single alert contained in the cluster click on the "locate" link placed next to the source IP address.

The JSON format has been chosen for storing the alerts since it can be easily acquired by the AJAX web interface through the method `XMLHttpRequest()` [34] and then parsed out by the method `JSON.parse()` [45] in a structure easily manageable by JavaScript itself. The click on a node lets JavaScript dynamically load the information over that cluster in the `div` placed at the bottom of the page through `document.getElementById()` and `innerHTML` JavaScript mechanisms.

It is possible to select a single alert in a cluster from the bottom page div and, if available, to store its associated traffic information as packet sequence in PCAP format. This is done by calling the Perl CGI script `pcap.cgi`, that takes as arguments the base64 encoded representations of the packets associated to a certain alert (fetched from the JSON correlations file), decodes them and prints them on output according to the PCAP format specifications and setting the content type to `application/pcap`. The PCAP specifications provides a global header for describing the sequence of packets:

```
typedef struct pcap_hdr_s {
    guint32 magic_number;   /* magic number */
    guint16 version_major;  /* major version number */
    guint16 version_minor;  /* minor version number */
    gint32  thiszone;       /* GMT to local correction */
    guint32 sigfigs;        /* accuracy of timestamps */
    guint32 snaplen;        /* max length of captured packets */
    guint32 network;        /* data link type */
} pcap_hdr_t;
```

and before the content of each packet a local header:

```
typedef struct pcaprec_hdr_s {
    guint32 ts_sec;         /* timestamp seconds */
    guint32 ts_usec;        /* timestamp microseconds */
    guint32 incl_len;       /* number of octets of packet saved in file */
    guint32 orig_len;       /* actual length of packet */
} pcaprec_hdr_t;
```

so the output will consist in a global PCAP header followed by a sequence of binary packets, each preceded by its local header. The data link type in the global header is automatically set to *ethernet*, and each packet is padded by a pseudo-ethernet header having `00:00:00:00:00:00` both as source and destination MAC address, since the packet captured by Snort's PCAP interface usually do not contain the data link layer.

The downloaded file can then be analyzed through most of the common network sniffers and analyzers.

Another opportunity offered by the web interface is to manually set correlations and *un*-correlations between alerts. This is done by using the keys "Manually correlate" or "Remove correlations" available in the web interface, then clicking on the two alerts to correlate or *un*-correlate. These keys invoke the *correlate.cgi* Perl CGI script, that acts on the XML files discussed in section 4.6.5 adding or removing correlations, and these files will be parsed back by the correlation module with a priority higher than the normal correlation indexes (two alerts having a high correlation index but manually specified as uncorrelated will not be correlated).

Moreover, the clusters of alerts (or the cluster of clusters of alerts) deriving from the clustering process of the *SOM* neural network's output layer described in section 4.6.4 are saved on the XML file `webserv_dir/neural_clusters.xml`, that can be directly open and explored through the browser (e.g. `http://snort_host:7654/neural_clusters.xml`) since it provides a HTML interface dynamically built by an XSLT [35] XML stylesheet interface contained in the file `webserv_dir/default.xsl`.

4.10 Configuration and installation

The latest source files of the module can be fetched from its *GitHub* page, `https://github.com/BlackLight/Snort_AIPreproc`, or explicitely performing a `git clone` operation:

```
$ git clone git://github.com/BlackLight/Snort_AIPreproc.git
```

After this, we can move in directory `Snort_AIPreproc` and run the configure script by `./configure <options>`, where options can be any of the following:

- `--prefix=<path>`: by default the module's files are installed in `/usr`. Use this option if you have Snort installed in any other location;
- `--with-mysql`: enables the support for MySQL database, both for reading and storing alert information (it requires `libmysqlclient`);
- `--with-postgresql`: enables the support for PostgreSQL database, both for reading and storing alert information (it requires `libpq`);
- `--without-graphviz`: by default the module uses `libgraphviz` for rendering correlation graphs expressed in *DOT* language into PostScript or PNG files. Use this option for disabling its support, if this library is not installed and cannot be installed on your system. In this case, in order to render the *DOT* graphs an external software would be required.

The required dependencies for building the module are:

- `libpthread` (*REQUIRED*), used for managing multiple threads in this software;
- `libxml2` (*REQUIRED*), used for parsing hyperalert knowledge base's XML files and manual correlations;
- `libgraphviz` (*RECOMMENDED*), used for rendering *DOT* graphs as PostScript or PNG;
- `libmysqlclient` (*OPTIONAL*), for interfacing the module with a MySQL database;
- `libpq` (*OPTIONAL*), for interfacing the module with a PostgreSQL database;
- A DBMS installed (*RECOMMENDED*), for storing information about clusters and correlations, both for making the analysis easier, and for letting the web interface and the neural network correlation indexes work properly. So far MySQL and PostgreSQL are supported, and according to the DBMS installed on the system the right option (`--with-mysql` or `--with-postgresql`) must be set at the configure script;
- *Perl* interpreter (*RECOMMENDED*), used for the CGI scripts invoked by the web interface that manage manual correlations and PCAP output format;
- `XML::Simple` Perl module (*RECOMMENDED*), used by the Perl CGI script that manages manual correlations.

After the dependencies have been installed and the configure script has been run as well, it is possible to compile and install the module through the commands:

```
$ make
$ make install
```

The last step consists in the configuration of the module's options. This step has already been described in section 4.1, and for further information over the options' default values the `README` file in the sources directory contains all the details.

Chapter 5

Experimental results

In order to test the software we considered several possible attack scenarios. In particular we take in exam the results of the module in the following conditions:

1. Some simple multi-step attacks, for example a sequence ICMP ping → port scan → shellcode execution → remote NFS mount;
2. A more complex network scenario with some network traffic from DARPA 99 [46] without any explicit attack, in order to train the module with some "background" traffic;
3. Some network traffic from DARPA 99 containing explicit attacks.

5.1 A simple multi-step attack sequence

We are going to trigger Snort's alert engine through a quite common multi-step attack pattern launched from a machine on the external network:

1. An ICMP ping sweep through the network scanner `nmap` [47] in order to find running machines in the network;
2. A TCP port scan launched on one of them through `nmap`;
3. A shellcode launched on a port running a vulnerable service;
4. A scanning of RPC resources on the host, once compromised, in order to find external filesystems that we can mount there.

We will launch the attack from an external machine (10.8.0.1) directly to the machine running Snort (10.8.0.6), without generating background traffic, supposing that the ping and the shellcode against this machines are run twice. The hyperalert models for these alerts are provided as XML files in the directory `corr_rules`, in particular the preconditions and consequences will be defined as it follows:

```
<!-- ICMP ping NMAP -->
<post>HostExists(+DST_ADDR+)</post>

<!-- TCP portscan -->
<pre>HostExists(+DST_ADDR+)</pre>
<post>HasService(+DST_ADDR+, +ANY_PORT+)</post>
```

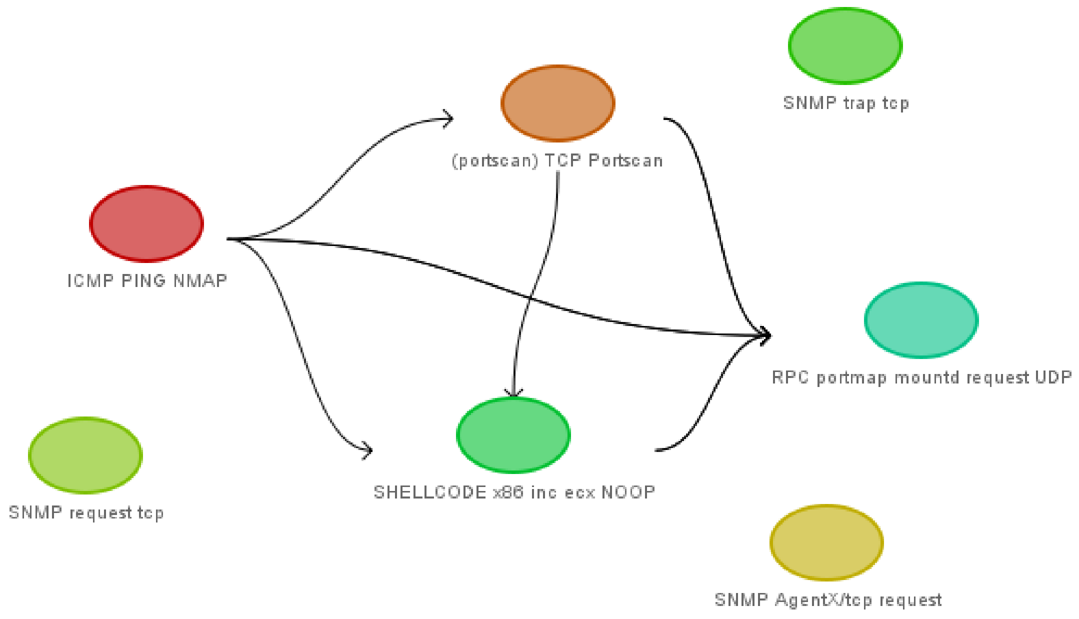

Figure 5.1: Representation of the alert correlation graph associated to the attack scenario (1) through the web interface

```
<!-- SHELLCODE x86 -->
<pre>HostExists(+DST_ADDR+)</pre>
<pre>HasService(+DST_ADDR+, +DST_PORT+)</pre>
<post>HasRemoteAccess(+SRC_ADDR+, +DST_ADDR+)</post>

<!-- RPC mountd request UDP -->
<pre>HostExists(+DST_ADDR+)</pre>
<pre>HasService(+DST_ADDR+, +DST_PORT+)</pre>
<pre>HasRemoteAccess(+SRC_ADDR+, +DST_ADDR+)</pre>
<post>HasNfsAccess(+SRC_ADDR+, +DST_ADDR+)</post>
```

As shown in fig. 5.1 a correlation between the main steps in the attack scenario is correctly found, since the hyperalert model, with its preconditions and consequences, was well-defined. A consistent hyperalert description model is indispensable when the system has not acquired enough traffic packets in order to perform a good heuristic alert correlation through the bayesian and the neural networks, since these correlation modules cannot be adequately trained yet to provide meaningful correlation indexes. Moreover, when few packets and alerts have been acquired by the system, the weights of these correlation indexes will be quite low or null as well, leading not to any major changes in the overall correlation indexes. So, on a system that has not acquired enough traffic packets and alerts, a good hyperalert model is the only way for having a good alert correlation graph.

We can notice three more alerts in the correlation graph, raised by the *SNMP* Snort's rules and triggered when a SYN packet comes on these specific ports. These alerts are raised since the TCP port scan also scans these ports, but as we see no correlation is found between them and the port scan since we do not have any hyperalert model in our knowledge base for modelling these alerts. Anyway, when more port scans are performed and more traffic is acquired by the system, and also the bayesian and neural networks correlation indexes gain a higher weight, a correlation may be found between the port scan alert and the *SNMP*-related alerts, since they often happen in the same time window.

We can also notice, through the web interface, that in the case when two ICMP ping or two shellcode alerts are raised in the same time window and/or from the same IP subnet, these alerts will be automatically grouped together by the alert clustering module, and their contents can be explored by the web interface itself.

Another interesting result comes from the file `webserv_dir/neural_clusters.xml`. Even if we just acquired few traffic packets and a pair of alerts as only training set for the neural network, a meaningful k-means clusterization is anyway available on the output layer from the *SOM*. We obtained a file like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="default.xsl" type="text/xsl"?>
<!DOCTYPE neural-clusters PUBLIC "-//blacklight//DTD NEURAL CLUSTERS//EN"
    "http://0x00.ath.cx/neural_clusters.dtd">

<clusters>
   <cluster id="0">
      <alert desc="SHELLCODE x86 inc ecx NOOP " gid="1" sid="1394" rev="12"
          src_ip="10.8.0.1" src_port="54714" dst_ip="10.8.0.6" dst_port="4444"
          timestamp="Wed Nov 24 22:26:00 2010" xcoord="18" ycoord="2"/>
      <alert desc="RPC mountd UDP export request " gid="1" sid="1924" rev="8"
          src_ip="10.8.0.1" src_port="39941" dst_ip="10.8.0.6" dst_port="44819"
          timestamp="Wed Nov 24 22:26:36 2010" xcoord="18" ycoord="2"/>
      <alert desc="RPC portmap mountd request UDP " gid="1" sid="579" rev="10"
          src_ip="10.8.0.1" src_port="56304" dst_ip="10.8.0.6" dst_port="111"
          timestamp="Wed Nov 24 22:26:35 2010" xcoord="18" ycoord="0"/>
   </cluster>

   <cluster id="1">
      <alert desc="(portscan) TCP Portscan " gid="122" sid="1" rev="0"
          src_ip="10.8.0.1" src_port="0" dst_ip="10.8.0.6" dst_port="0"
          timestamp="Wed Nov 24 22:25:09 2010" xcoord="0" ycoord="19"/>
   </cluster>

   <cluster id="2">
      <alert desc="SNMP AgentX/tcp request " gid="1" sid="1421" rev="13"
          src_ip="10.8.0.1" src_port="56050" dst_ip="10.8.0.6" dst_port="705"
          timestamp="Wed Nov 24 22:25:09 2010" xcoord="19" ycoord="0"/>
      <alert desc="SNMP request tcp " gid="1" sid="1418" rev="13"
          src_ip="10.8.0.1" src_port="56050" dst_ip="10.8.0.6" dst_port="161"
          timestamp="Wed Nov 24 22:25:14 2010" xcoord="19" ycoord="1"/>
      <alert desc="SNMP trap tcp " gid="1" sid="1420" rev="13"
          src_ip="10.8.0.1" src_port="56050" dst_ip="10.8.0.6" dst_port="162"
          timestamp="Wed Nov 24 22:25:14 2010" xcoord="19" ycoord="1"/>
      <alert desc="SNMP AgentX/tcp request " gid="1" sid="1421" rev="13"
          src_ip="10.8.0.1" src_port="56050" dst_ip="10.8.0.6" dst_port="705"
          timestamp="Wed Nov 24 22:25:09 2010" xcoord="18" ycoord="0"/>
      <alert desc="SNMP request tcp " gid="1" sid="1418" rev="13"
          src_ip="10.8.0.1" src_port="56050" dst_ip="10.8.0.6" dst_port="161"
          timestamp="Wed Nov 24 22:25:14 2010" xcoord="18" ycoord="0"/>
      <alert desc="SNMP trap tcp " gid="1" sid="1420" rev="13"
```

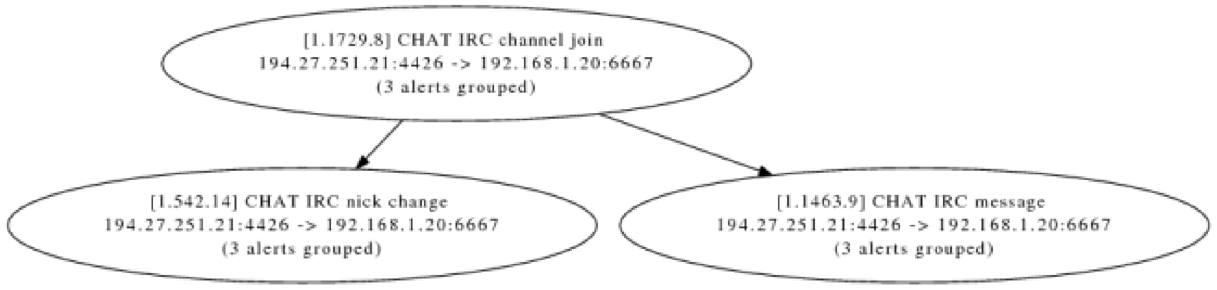

Figure 5.2: IRC "false" alert sequence correctly identified by the module on the first week of DARPA 99 traffic

```
         src_ip="10.8.0.1" src_port="56050" dst_ip="10.8.0.6" dst_port="162"
         timestamp="Wed Nov 24 22:25:14 2010" xcoord="18" ycoord="0"/>
   </cluster>

   <cluster id="3">
      <alert desc="ICMP PING NMAP " gid="1" sid="469" rev="4"
         src_ip="10.8.0.1" src_port="0" dst_ip="10.8.0.6" dst_port="0"
         timestamp="Wed Nov 24 22:25:09 2010" xcoord="0" ycoord="1"/>
      <alert desc="ICMP PING NMAP " gid="1" sid="469" rev="4"
         src_ip="10.8.0.1" src_port="0" dst_ip="10.8.0.6" dst_port="0"
         timestamp="Wed Nov 24 22:25:09 2010" xcoord="0" ycoord="0"/>
   </cluster>
</clusters>
```

In this representation `xcoord` and `ycoord` denote the coordinates on the Cartesian plan where a certain alert, identified by its type, description, source and destination addresses and ports and timestamp, is *mapped*.

We notice that, through our "smart" weight initialization algorithm, even a relatively small data set can give interesting results. In particular here we find the shellcode alert and the two RPC-related alerts grouped together, the SNMP-related alerts grouped together as well, the two ICMP ping alerts also forming a cluster, and the TCP port scan in a separated cluster. This result can be analyzed in a more comfortable way through the web interface, since the XSLT file included in this XML provides a dynamic conversion of the XML itself into a nicely formatted HTML content.

5.2 Big traffic dumps without attacks

The second step is to test the software through big dumps of traffic in .pcap format without notable threats. In order to do this we select the traffic dump from the first day of the first week of DARPA 99 [46] network traces.

DARPA traces are big dumps of network traffic collected in three years (98, 99 and 2000), some of which containing attacks and some not. These have been traditionally used in several papers and projects over intrusion detection as *benchmarks* for testing the software or some of its components.

Some alerts are anyway triggered by Snort using the "community rules", since IRC actions are logged as alerts as well (in particular channel joins, nick changes and IRC messages) and the same happens for some usually harmless alerts, like "consecutive TCP small segments exceeding threshold" or "attempted ARP cache overwrite attack" alerts sometimes triggered on a network with a large amount of ARP replies. The trace from DARPA's first week contains many of these alerts, successfully recognized and grouped in homogeneous clusters by the module: indeed, the alert clusters' correlation graph successfully identifies a node associated to an "attempted ARP cache overwrite attack" cluster containing 505 alerts of this type. The same happens with "consecutive TCP small segments exceeding threshold", as these alerts are successfully grouped in clusters depending on their source and destination IP subnets/ranges of ports. Moreover, the module, as shown in fig. 5.2, successfully recognizes the common sequence of three IRC "pseudo" alerts. Indeed, the module succeeds in identifying a cause-effect relationship in the sequence "IRC channel join" $\rightarrow$ {"IRC nick change", "IRC message"}. Actually we know that a nick change operation or a message can only happen when a certain IP address is connected to a channel. In order to have such a plausible result we need to:

- Choose the right value for the parameter `correlation_thread_coefficient`, the k parameter in eq. 3.10, depending on the network topology, on the traffic to be analyzed (or the usual traffic on the network we analyze), on the correlation threshold we need to have for the purposes of our application and analysis, and other solution-specific parameters. For the specific solution of DARPA 99 first week we found out that 1.0 is a reasonable value for this parameter. This value could change in other contexts or applications with different alert correlation distributions, but it was noticed that a value $\simeq 1$ is quite good for most of the network traffic types whose correlation coefficients are *adequately* approximated by a normal gaussian distribution. This behaviour is usually recognizable in most of the network scenarios with long-term stored information over the history of associated network traffic and alerts, since the average correlation value does not change much after a certain quantity of acquired knowledge over network traffic and alerts (supposing the kind of traffic on that network does not change much), and even new alerts and types of traffic, if occasional, would not sensibly change this numerical distribution. Anyway, for other cases and scenarios a different value should be empirically chosen for best modelling the usual correlation index. The analyst should also be careful not to choose a too low (close to zero or even less than zero) or too high value for this parameter. A too low value would result in correlation relations found from most of the alerts to most of the alerts. On the contrary, a too high value would result in a nearly empty correlation graph, since the correlation threshold is too high. Testing the software with the first week of DARPA 99 it was noticed that even a value of 0.6 for this parameter dramaticly increases the number of correlations found between actually uncorrelated or loosely correlated alerts;

- Provide a good hyperalert model for as many alerts as possible, or at least for the most common in a certain network or traffic scenario. The knowledge base correlation index has in fact always the highest weight among the correlation indexes when the overall correlation is computed, so it is crucial sometimes for identifying actually correlated alerts, especially when the acquired history of traffic and alerts does not contain enough elements for identifying such relations using indexes from the neural or bayesian network;

- Provide a good clusterization model that best fits the topology of the network, its kind of traffic and the analysis purposes. A wrong specification of IP subnets or the port ranges usually receiving most of the traffic, in fact, may result in a poor alert clusterization, in too large groups (for example having 0.0.0.0/0 as associated "clustered" IP subnet), if few and too large clusters were specified, or in an excessive fragmentation of clusters, probably consisting in many clusters containing a single alert, if too many clusters actually "groupable" together were specified.

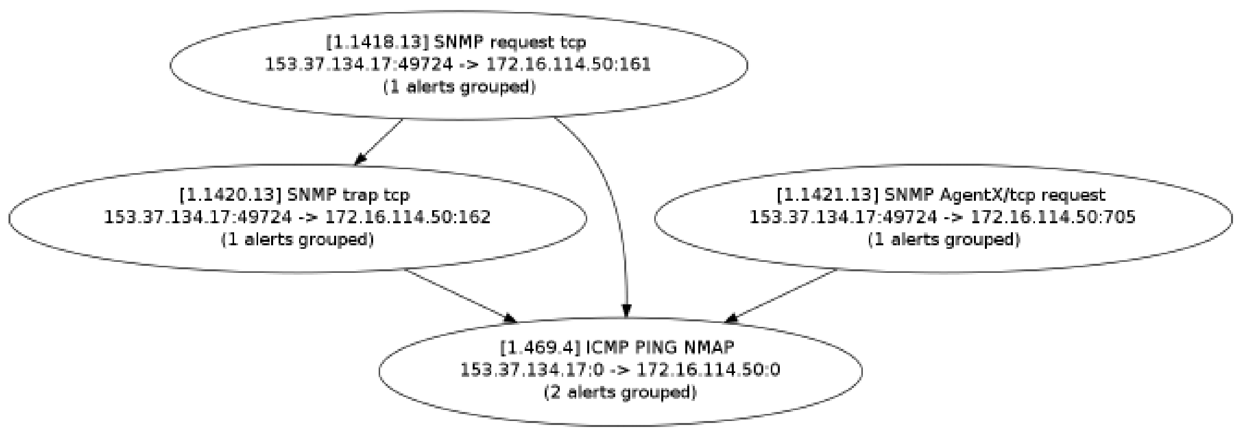

Figure 5.3: Correlations between nmap port scan and SNMP-related alerts found by the module

5.3 Big traffic dumps with attacks

The next step while testing the software has been to test it using traffic dumps from the second week of DARPA 99 data set, about 400 MB of traffic dump. Several attack scenarios are correctly identified by the module, even without having a complete hyperalert description for all the alerts. For example, in the first test we discussed over the SNMP alerts triggered by Snort during an nmap port scan, since some ports that raise an SNMP alert are "knocked". Our first example scenario could not find a correlation among the scan and the SNMP alerts, since no hyperalert model was available for the SNMP alerts and the IDS did not sniff a reasonable amount of traffic for empirically finding correlations. With such a large "training set" the *pseudo*-bayesian module finds a strong temporal correlation between the SNMP alerts and the port scan, i.e. they very often happen in the same time window. This successfully identified correlation is shown in the graph in fig. 5.3.

We also defined a hyperalert model for expressing preconditions and consequences of port scans, shellcode alerts and directory listings. In particular, we defined:

```
<!-- (portscan) UDP Portsweep -->
<pre>HostExists(+DST_ADDR+)</pre>
<post>HasService(+DST_ADDR+, +ANY_PORT+)</post>
```

```
<!-- SHELLCODE x86 NOOP -->
<pre>HostExists(+DST_ADDR+)</pre>
```

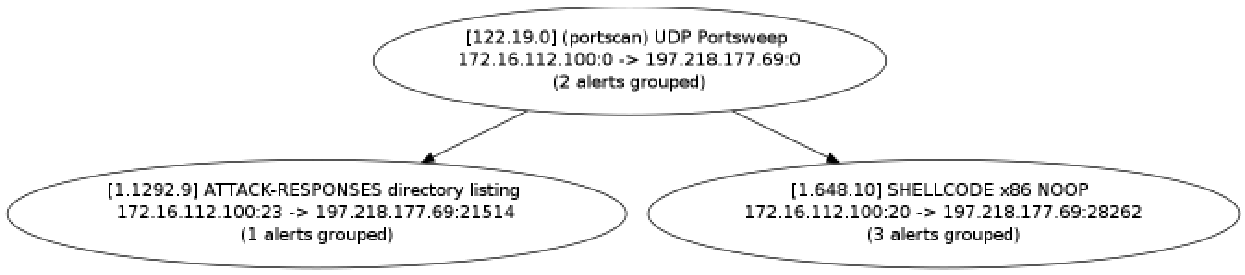

Figure 5.4: Correlations between port scan, shellcode execution and directory listings found by the module

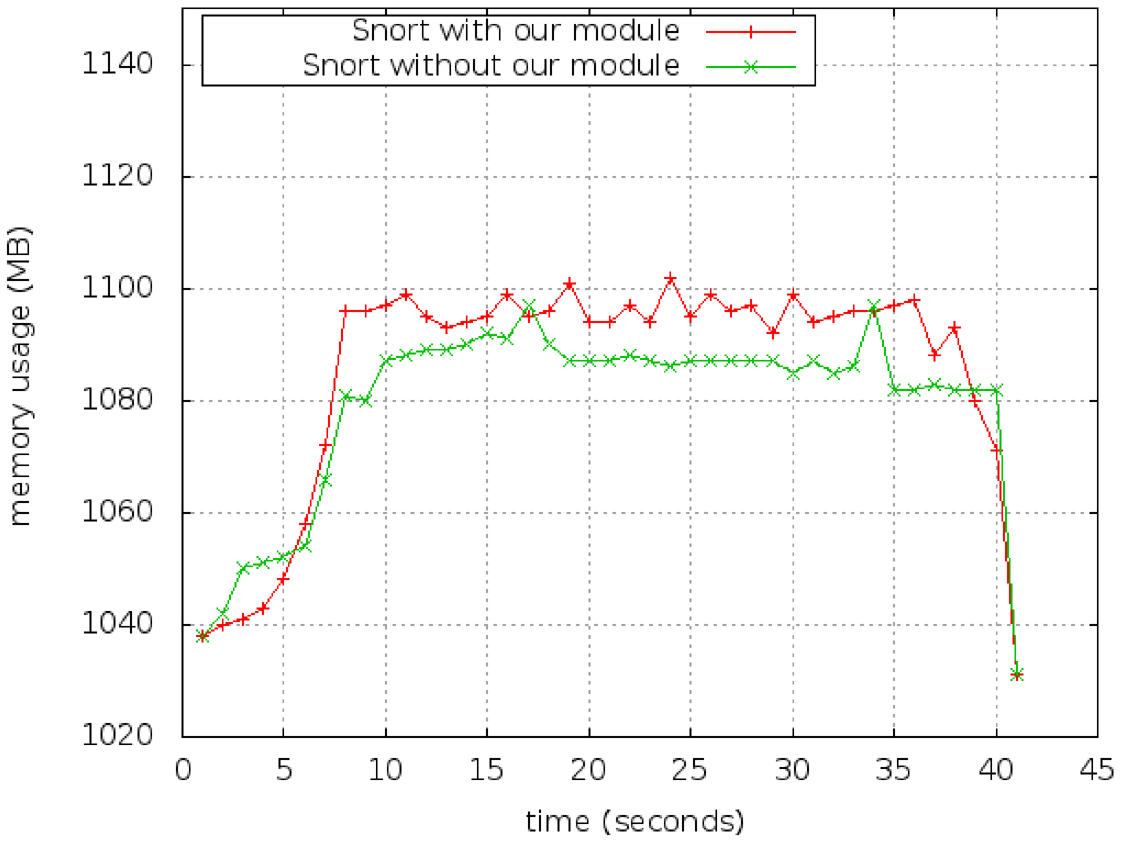

Figure 5.5: Snort's memory usage graph shown with and without our module

```
<pre>HasService(+DST_ADDR+, +DST_PORT+)</pre>
<post>HasRemoteAccess(+SRC_ADDR+, +DST_ADDR+)</post>

<!-- ATTACK-RESPONSES directory listing -->
<pre>HostExists(+DST_ADDR+)</pre>
<pre>HasService(+DST_ADDR+, +DST_PORT+)</pre>
<post>HasFileInfo(+SRC_ADDR+, +DST_ADDR+)</post>
```

Using this model, the software successfully recognizes the correlation graph shown in figure 5.4.

5.4 Software performance

As shown in fig. 5.5, representing the Snort's memory usage graph when elaborating the first week of DARPA 99 network dump (about 300 MB of raw traffic), the memory usage does not change drastically when Snort is executed with (red line in the graph) or without (green line) our module. The initial "slope" in memory usage (period 0-7 seconds) identifies Snort's startup, followed by a relatively stable memory usage and the deallocation of Snort's memory resources at the end (after about 40 seconds). The "spikes" in the red graph identify the dynamic allocation and deallocation of the module's structures while analyzing the alerts, and, as we notice, the overall memory usage is approximately the same with or without the module and no visible leak is detected. Also the analysis time is the same, about 40 seconds in both the cases for analyzing 300 MB of DARPA 99 traffic on an *Intel Centrino* platform.

Anyway, when analyzing large amounts of traffic it is strongly suggested, unless it is really needed, to disable the memory storage of the currently acquired traffic flows into the hash table. In fact, each TCP flow is stored in the hash table as long as its associated alert remains there, this usually means until the software is running. Also adopting some strategies for optimizing the usage of the memory, like implementing the single linked lists representing a TCP flow as a *circular buffer* (when a list reaches its maximum size the oldest packets are removed), the problem of memory consumption on a system where many alerts are raised still remains. A port scan suffices for generating a large number of TCP flows in memory, stored in the traffic hash table, since each SYN packet sent to a port generates a TCP flow associated to an IDS triggered alert that therefore is not deallocated until its associated alert stays there. The solution of setting a low number for the configuration parameter `max_hash_pkt_number` provides the chance to reduce the number of packets contained in each flow stored in the hash table, and therefore the overall memory consumption due to the packet hash table itself. However, when this solution should not be enough to minimize the memory consumption, for example on a system where many different alerts are triggered by many different sources, a better solution may be to set the configuration option `use_stream_hash_table` to 0 (default: 1), so that the packet hash table is not used at all. The most evident drawback is that the traffic associated to a raised alert cannot be store as PCAP file and analyzed anymore, but this could be the only adoptable solution when the number of triggered alerts is too high for storing all the traffic information.

Apart from this, the software should not have any known bugs with memory. An analysis executed by running Snort and this module into the open source memory analyzer *valgrind* [48] showed no memory management error or leaks in the module's circa 15000 lines of code.

Chapter 6

Conclusions and future work

Many goals have been achieved in the design and development of this software compared to the current state of art. These improvements will be briefly analyzed in this section.

6.1 Algorithmic improvements

The alert clustering algorithm proposed by Klaus Julisch [6] has been significantly improved through:

- A *time-window* oriented mechanism that avoids to cluster together similar alerts raised in distant moments;
- A heuristic strategy for computing the best size for the cluster of alerts based on the alert type heterogeneity, therefore avoiding to statically specify this parameter.

The correlation indexes between pairs of alerts are computed as weighted mean among several single correlation indexes, both semi-deterministic and completely heuristic, therefore succeeding in the aim to combine together indexes associated to completely different correlation strategies. Moreover, the heuristic correlations are multiplied by a dynamic weight, that is computed in function of the acquired traffic and alert knowledge.

A completely new *pseudo*-bayesian algorithm (it "resembles" a bayesian correlation since it computes the conditioned probability of an alert type given another alert type, but the strategy used for computing this probability does not use Bayes' theorem) has also been proposed and successfully tested for computing alert correlations based on temporal constraints and gaussian correlation decay in function of the time elapsed between two alerts.

A Self-Organizing Map (*SOM*) has been used as correlation index, as discussed in [8] and other documents. Anyway the performance and accuracy of the network's results has been significantly improved by using a smart initialization algorithm for the weights, and for the first time a clustering strategy over the neural network's output layer (with a heuristic estimation over the number of clusters) is used in order to identify groups of alerts belonging to the same attack scenario.

A modular and customizable XML-based language has also been developed and successfully tested for defining hyperalert types used as knowledge base for the correlation indexes. This language inherits the positive sides of languages like *CAML* (fully modular, expressive, extensible, supporting macros and user-defined variables) and succeeds in the aim to be clean, easily human-readable and analyzable by any XML parser.

6.2 Software improvements

For the first time an alert clustering and correlation engine is built directly on top of Snort, the most popular Intrusion Detection System, and directly integrated as preprocessor module.

The developed module is highly configurable through `snort.conf` and it groups together several techniques for clustering and correlating alerts, also providing an easy web-based interface for navigating, modifying and analyzing clusters and correlation graphs of alerts. Moreover, the developed module is itself modular, since new correlation indexes can be easily added to the engine through an intuitive programming interface. It can be therefore considered as a *framework* for alert analysis.

The system also supports manual correlation, specifiable though an easy "drag'n'drop" mechanism in the web interface. A XML-based language has also been proposed for modelling these correlation.

The software is stable, even when analyzing huge amounts of traffic, and it does not have a strong impact over the performance of the IDS.

The obtained results, both for alert clustering and alert correlation, have been very satisfactory. Anyway a good hyperalert module, especially when the other correlation modules are not enough "trained" yet, would be needed for having a more accurate alert correlation, as well as a good value for the correlation threshold parameter in the configuration depending on the network topology and the traffic to be analyzed. Moreover, when the module lacks of knowledge over some hyperalert models, and the temporal information over these pairs of alerts is incomplete or not fully adherent to the expected values, some wrong correlations may be found (false positives) or some expected correlations may not be found (false positives). This problem can be partly fixed by using the web interface in order to specify the actual correlation between pairs of alerts.

6.3 Future work

Some features have been identified as plausible to be developed in future for improving even more the software. Among these:

- Improving the hyperalert knowledge base. So far the directory `corr_rules` contains approximately 20 hyperalert models. This number can be increased in future by specifying more and more models, since it was tested that a more accurate knowledge over the contexts of alerts provides more accurate results in alert correlations, especially in lack of satisfactory traffic information and history. We should moreover remember that the accuracy of the correlation process also depends on the accuracy of the *IDS* itself to properly detect some attacks, minimizing the amount of false positives and false negatives, since the module relies on Snort in order to cluster and correlate alerts;

- Providing more correlation indexes, in order to correlate a set of alerts using more points of view and therefore reducing the possibility of correlation errors. An interesting approach is the one discussed in [2], using a naive bayesian network with sliding temporal window where no knowledge but some basic information over the usual goals of an attacker is needed in order to correlate alerts. This algorithm can be implemented as correlation module in this software;

- Finding an algorithm that computes the best value for k parameter in equation 3.10. This parameter now needs to be manually set by the user in function of the traffic type, of the triggered alerts and their heterogeneity. A solution for the future may be to find an algorithm that automatically computes the best value for this parameter in function of the network traffic and the types of triggered alerts;

- Integrating the module with logs from more sources. So far the module analyzes alerts from Snort logs. In future it could be useful to fetch traffic information from other sources as well, for example other Intrusion Detection Systems (like *Prelude*) or logs from *Apache* or the Unix *syslog* itself;

Bibliography

[1] Barbarà, Jajodia. *Detecting Novel Network Intrusions Using Bayes Estimators.* 1st SIAM International Conference on Data Mining

[2] Salem Benferhat, Tayeb Kenaza. *Causality and intervention for alarm correlation: A Naive Bayes approach for detecting coordinated alerts.*

[3] François Chapeau-Blondeau. *Numerical Evaluation of the Lambert W Function and Application to Generation of Generalized Gaussian Noise with Exponent 1/2.* IEEE Transactions on Signal Processing, Vol. 50, No. 9, September 2002

[4] Cheung, Lindqvist, Fong. *Modelling Multistep Cyber attacks for Scenario Recognition.* Proceedings of the Third DARPA Information Survivability Conference and Exposition, 2003, Vol. 1, pages 284-292

[5] Mahmoud Jazzar and Aman Jantan. *A Novel Soft Computing Inference Engine Model for Intrusion Detection.* IJCSNS International Journal of Computer Science and Network Security, Vol. 8, No. 4, April 2008

[6] Klaus Julisch. *Clustering Intrusion Detection Alarms to Support Root Cause Analysis.* ACM, Vol. 2., No. 3, 09 2002, pages 111-138

[7] Teuvo Kohonen. *Self Organizing Maps.* Published by Springer, 1997

[8] Kumar, Siddique, Noor. *Feature-Based Alert Correlation in Security Systems Using Self Organizing Maps.*

[9] Manganaris, Christensen, Zerkle, Hermiz. *A Data Mining Analysis of RTID Alarms.* Computer networks 34(4), pages 571-577

[10] Matti Manninen *Using Artificial Intelligence in Intrusion Detection Systems.* Helsinki University of Technology

[11] Pablo Navarrete and Javier Ruiz-del-Solar. *Interactive Face Retrieval using Self-Organizing Maps.*

[12] Ning, Cui, Reeves, Xu. *Techniques and Tools for Analyzing Intrusion Alerts.* ACM, Vol. 5, May 2004, pages 1-44

[13] Nowicka, Zawada. *Modelling Temporal Properties of Multi-Event Attack Signatures in Interval Temporal Logic.*

[14] Sheyner, Haines, Jha, Lippmann, Wing. *Automated Generation and Analysis of Attack Graphs.* Proceedings of the 2002 IEEE Symposium on Security and Privacy

[15] Smith, Japkowicz, Dondo, Mason. *Using Unsupervised Learning for Network Alert Correlation.*

[16] Mu-Chun Su, Ta-Kang Liu and Hsiao-Te Chang. *Improving the Self-Organizing Feature Map Algorithm Using an Efficient Initialization Scheme.* Tamkang Journal of Science and Engineering, Vol. 5, No. 1, pp. 35-48 (2002)

[17] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel. *Advances in Information Security, Intrusion Detection and Correlation Challenges and Solutions.* Springer 2005

[18] Wang, Ghorbani, Li. *Automatic Multi-Step Attack Pattern Discovering.* International Journal of Network Security, Vol. 10, No. 2, pages 142-152

[19] Yun Xiao and Chongzhao Han. *Correlating Intrusion Alerts into Attack Scenarios based on Improved Evolving Self-Organizing Maps.* IJCSNS - International Journal of Computer Science and Network Security, Vol. 6 No. 6, June 2006

[20] Jang-Hee Yoo, Byoung-Ho Kang and Jae-Woo Kim. *A Clustering Analysis and Learning Rate for Self-Organizing Feature Map.*

[21] `http://www.snort.org` *The open source Intrusion Detection System used as platform in this document*

[22] `http://en.wikipedia.org/wiki/Euclidean_distance` *Definition of Euclidean distance*

[23] `http://en.wikipedia.org/wiki/Manhattan_distance` *Definition of Manhattan distance*

[24] `http://en.wikipedia.org/wiki/Lambert_W_function` *Definition of Lambert W function, used for solving a logarithmic equation in this document*

[25] `http://en.wikipedia.org/wiki/Self-organizing_map` *Self-Organizing Maps*

[26] `http://en.wikipedia.org/wiki/Boyce-Codd_normal_form` *Boyce-Codd normal form for database schemas*

[27] `http://en.wikipedia.org/wiki/DOT_language` *DOT language, used for modelling graphs in this document*

[28] `http://en.wikipedia.org/wiki/Pcap` *PCAP library and file format, used both for sniffing traffic and for exporting flows of traffic into a uniformed file format*

[29] `http://en.wikipedia.org/wiki/JSON` *JSON format*

[30] `http://en.wikipedia.org/wiki/Base64` *base64 encoding*

[31] `http://en.wikipedia.org/wiki/Inotify` *inotify, a Linux mechanism for monitoring the actions on a file*

[32] `http://en.wikipedia.org/wiki/K-means_clustering` *k-means clustering algorithm*

[33] `http://www.w3.org/CGI/` *CGI format and protocol*

[34] `http://www.w3.org/TR/XMLHttpRequest/` *XMLHttpRequest JavaScript method*

[35] `www.w3schools.com/xsl/` *XSL/XSLT format*

[36] `http://www.modelselection.org/bic/` *Schwarz criterion used for linear regression and for heuristically choose the best number of clusters in a data set*

[37] `http://goo.gl/Ozgc` *Equation solution using the engine Wolfram-Alpha*

[38] `http://goo.gl/9ht4` *Equation solution using the engine Wolfram-Alpha*

[39] `http://www.graphviz.org/` *Graphviz library, used for managing graphs in this software*

[40] `http://uthash.sourceforge.net/` *uthash, library used for managing hash tables in this software*

[41] `https://github.com/BlackLight/fsom` *fsom, library developed by the author of this document and used in this software for managing Self-Organizing Maps*

[42] `https://github.com/BlackLight/fkmeans` *fkmeans, library developed by the author of this document and used in this software for managing k-means clustering*

[43] `http://www.graphdracula.net/` *Dracula, JavaScript library for dynamically managing graphs on a web interface using SVG graphics and JavaScript*

[44] `http://raphaeljs.com/` *RaphaelJS, JavaScript library on top of which Dracula was built, developed for managing any kind of SVG graphics in JavaScript*

[45] `http://www.json.org/js.html` *JavaScript method JSON.parse()*

[46] `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/docs/index.html` *DARPA 98, 99 and 2000 data sets used as traffic dumps for testing the software*

[47] `http://nmap.org` *nmap, probably the most used open source port scanner and network analyzer*

[48] `http://valgrind.org/` *valgrind, an open source tool used for debugging memory leaks and memory errors*

List of Figures