Using an Intrusion Detection Alert Similarity Operator to Aggregate and Fuse Alerts

Fabien Autrel et Frédéric Cuppens

GET-ENST-Bretagne, 35576 Cesson Sévigné (France)

An important problem in the field of intrusion detection is the management of alerts. Intrusion Detection Systems tend to produce a high number of alerts, most of them being false positives. But producing a high number of alerts does not mean that the attack detection rate is high. In order to increase the detection rate, the use of multiple IDSs based on heterogeneous detection techniques is a solution but in return it increases the number of alerts to process. Aggregating the alerts coming from multiple heterogeneous IDSs and fusing them is a necessary step before processing the content and the meaning of the alerts. We propose in this paper to define a similarity operator that takes two IDMEF alerts and outputs a similarity value between 0 and 1. We then propose some algorithms to process the alerts in a on-line or off-line approach using this operator. The article ends up with experimentations made with the *Nmap* tool and the *Snort* IDS.

Mots-clés: Intrusion Detection, Alert, Intrusion Detection Message Exchange Format, aggregation, fusion, Similarity

1 Introduction

Every IDS has its avantages and drawbacks respectively to the technique it makes use of in order to detect attacks. Actually using only one IDS over a computer network will restrict the range of detected attacks. Therefore using multiple IDSs is a solution to increase intrusion detection rate, each IDS compensating the other IDSs'weaknesses. But making use of multiple IDSs raises some problems. First, the amount of alerts generated is too important to be handled by a system administrator. Second, since some IDSs may detect the same attack at the same time, information carried by the alerts are redundant. Hence it would be convenient to be able to assess the similarity of two alerts, a similarity of 0 meaning "those two alerts are not related to the same event" and a similarity of 1 meaning "those two alerts have been generated upon the same event". Once we are able to evaluate the similarity of two alerts, we can apply clustering algorithms to a set of alerts in order to do on-line or off-line processing.

The remainder of this paper is organized as follows: section 2 presents previous work related to alert agregation. Section 3 presents the problem of measuring the similarity of two intrusion detection alerts. We present how we model the alerts and how we compare their attribute. This section ends by presenting the way we aggregate the similarity values computed between the attributes of two alerts to produce a similarity value. Section 4 presents the aggregation tool we have implemented and some experimental results. Finally, section 5 conclude this paper and presents some ongoing work.

2 Related work

Some authors have investigated the alert aggregation problem and some solutions have been suggested. In [VS01] Valdes and Skinner use a probabilistic approach where a similarity function is defined for each attribute. They obtain an overall similarity value by combining similarity functions using an expectation of similarity.

In [Jul03] Julisch proposes to build clusters of alarms minimizing the cluster dissipation given that its size must be kept above a minimum number of alarms given by the user. In this case the analysis is done

off-line to determine the so called "root causes", i.e. the most basic causes that explain the alarm clusters. Julisch argues that these root causes are mostly configuration problems.

In [DW01], Debar and Wespi present the aggregation and correlation of intrusion detection alerts. Similar alerts are aggregated through the use of *duplicate definitions*. A duplicate definition specifies the alerts that should be observed when an event is detected. Namely, a definition specifies the alert A_1 that should be observed after another alert A_2 and which attributes should match between A_1 and A_2. This approach allows the authors to provide a fast implementation but cannot aggregate alerts that have no associated duplicate definition.

In [Cup01], the author defines a logic predicate $sim\_alert(Alertid_1, Alertid_2)$, that is true when two alerts are similar. To evaluate this predicate, a set of expert rules must be defined to determine when two alerts attributes are similar. The comparison of two attributes depends on the alerts classification, i.e the events associated with the alerts. As in [DW01], this approach lacks the ability to aggregate alerts with an unknown classification. We can add that for the two last approaches, it is not possible to quantify the similarity of two alerts.

3 Intrusion detection alerts similarity

Intrusion detection alerts have multiple formats depending on the IDS generating it. The set of information included in the alert also differs depending the on intrusion detection technique used. The set of information associated with an alert is called its attributes.

The process of alert comparison consists in comparing the sets of attributes of two alerts. We need to formally define this set of attributes in order to define the set of functions needed to compare two alerts generated by two arbitrary IDSs.

It is important to note that this article does not deal with alert correlation as it is defined in [CM02] or in [NCR02]. The aggregation process tries to group alerts by causes [Cup01], whereas the correlation process tries to chain alerts in order to find scenarios of alerts (see also [MMDD02]).

3.1 Intrusion detection alerts modelling

Very few articles try to define a model for intrusion detection alerts. In [VS01] the authors represent an alert as a set of features. In [Jul03] Julisch models an alert as a tuple over the Cartesian product $X_{1<i<n}\text{dom}(A_i)$, $A_1, A_2, .., A_n$ being the set of attributes and $\text{dom}(A_i)$ the values A_i can take.

The later model is the one chosen in this article. Note that the set of attributes that can be included into an alert (an alert may include only a subset of all available attributes) is determined by the alert data format we have chosen. We choosed the IDMEF alert data format [DC04]. Our motivation for choosing this format is exposed in section 3.3.1.

We define a set E of events types that can generate alerts. The idea is to be able to associate an event and a set of properties linked to this event to an alert. An alert can then be viewed as an instance of an element of E.

3.2 Alert attributes comparison

In this section we expose the set of functions we need in order to be able to compare the attributes of two alerts. We distinguish 4 types of attributes:

- Categorical attributes: discrete values without order (IP address, port number...)
- Numerical attributes: counters, packets size.
- Temporal attributes: different from numerical attributes because of some properties such as periodicity.
- Character strings: arbitrary text data if not formatted.

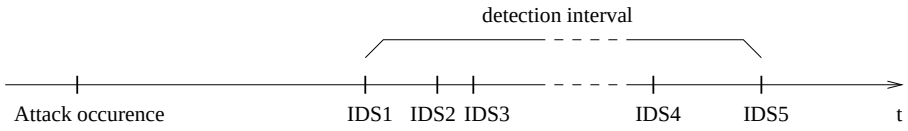

Fig. 1: Detection time interval for the occurrence of an attack

Each of the presented function returns a similarity value given two attributes, i.e they return a real in the interval $[0,1]$.

The existing alert formats have a subset of attributes in common. For example the source IP address, the date of the alert, the number of the accessed port, etc... However some intrusion detection technique may be able to add some extra information to a report. The IDMEF alert data format allows the IDSs to include a wide range of information into the alert. We use this format because of its capacity to include data output by various intrusion detection techniques. The attributes presented in this section are only pertaining to the IDMEF format.

3.2.1 Time comparison

An intrusion detection alert can contain two different dates. It can either tell us when the event at the origin of the alert has been detected or it can give the date at which the alert has been created. We make no distinction between those dates and consider the alert creation time as the associated event detection time.

For each event type of E we define a time interval specifying the temporal window outside of which executing another instance of the attack does not allow to say that those two events are related.

Computing the temporal window

To determine the similarity function parameters, we launch an instance of each event type in E on a network monitored by IDSs employing as many different detection techniques as possible. Since our goal is to aggregate alerts created by IDSs using different detection approaches, we must take into account the fact that some detection techniques implementation might require more processing power than other techniques. Hence such slower IDSs may generate their alert after other IDSs upon the detection of the same event. After playing an event instance we obtain a set of alerts not necessarily raised at the same time, depending on the processing power required by each IDS. We therefore consider the most recent and the oldest DetectTime values in this set of alerts to compute a temporal interval (figure 1).

This temporal interval is useful for aggregation since we want to be able to aggregate alerts that have been sent upon the occurrence of the same event instance. The idea is to separate alerts associated with the same event type but not the same event instance. Such alerts should have time values outside of the detection temporal window of the previous event instance. However two event instances may be in the same temporal window. In this case some of the alerts associated with two event instances may be aggregated. This means that an alert resulting from the fusion of a group of aggregated alerts may have greater granularity than elementary alerts.

Actually we compute two temporal intervals. The first value is obtained by playing the event instance on a network with no other traffic (T_{min}) and the second value is obtained by playing the event instance on a network with neutral traffic (T_{max}). Generally (T_{max}) is greater than (T_{min}) since the neutral traffic analysis slows down the IDSs and slows down the alert transmission on the network.

Computing the similarity value

Given two dates t_1 and t_2 and the temporal window corresponding to the alert event (see figure 2), the similarity function f_{s_t} is defined as follows:

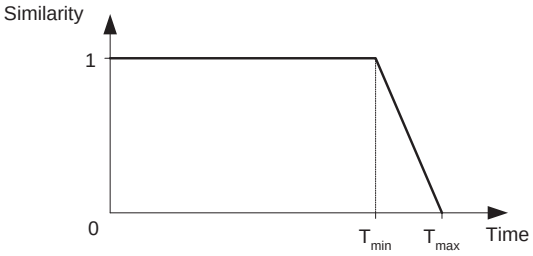

Fig. 2: Similarity function for the DetectTime attribute

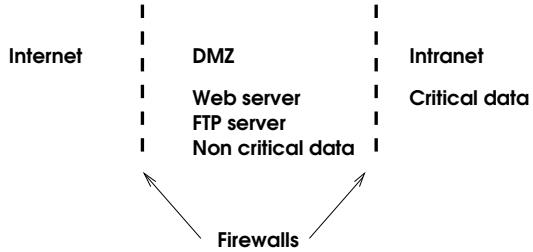

Fig. 3: Generic network configuration

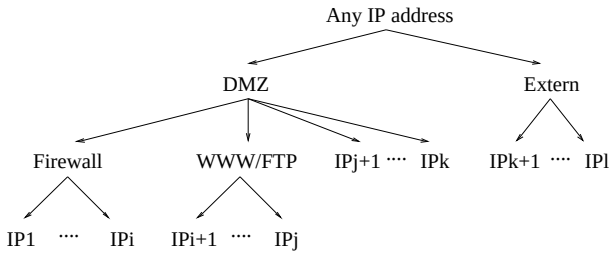

Fig. 4: Example of IP address taxonomy

$$f_{s_t}(t_1,t_2) = \begin{cases} 1 & \text{if } |t_1 - t_2| < T_{min} \\ \frac{T_{max} - |t_1 - t_2|}{T_{max} - T_{min}} & \text{if } T_{min} < |t_1 - t_2| < T_{max} \\ 0 & \text{otherwise} \end{cases}$$

Comparing the dates of two alerts makes sense only if the two analyzers that generated the alerts are synchronized, otherwise the similarity value might be lower or higher than expected. In the case of an architecture where IDS alerts are collected by a specific module and then forwarded to other alert processing modules, we expect this concentration module to correct the clock shifts.

3.2.2 IP address comparison

To be able to compare two IP addresses, we must think about the topology of a local network connected to the internet. Generally the network has a protected LAN with a private server, a DMZ (DeMilitarized Zone) with a public server and firewalls that add a layer of security between critical data and the internet (see figure 3).

The comparison of two IP addresses must take into account the two machines locations in the network topology. For example two addresses referring to two different sub-nets should be considered as less similar than two addresses referring to the same subnet. It is also interesting to take into account the type of machine associated with an IP address. This machine might be a web server, a DNS or a firewall. To achieve similarity comparison of those information, we choose to use a taxonomy over the IP addresses. This choice has been motivated by the work exposed in [Jul03]. The IP address taxonomy depends on the network topology (figure 4).

Computing the similarity value

To compute the similarity value between two IP addresses, we compute the distance between the two IP addresses in the taxonomy tree. The similarity value is then equal to the inverse of the maximal distance minus the calculated distance.

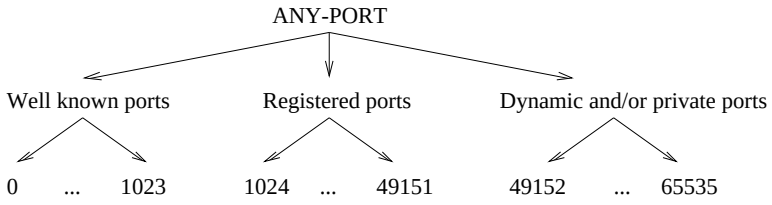

Fig. 5: Port numbers taxonomy

3.2.3 User data comparison

An alert may contain data about the user at the origin of an alert and/or about the user targeted by the action at the origin of the alert. The data we have to compare is specified by the IDMEF UserId class.

The User class contains one or more instances of the UserId class, which encapsulates specific information about a user or process. The UserId class contains interesting information about a user or a group, namely its user or group name and number. We said in the introduction that string attributes are hard to use since they often contains arbitrary formatted data. But in the UserId class, the user or group name is stored as string attribute that can be used to compare two groups or users. However we should not compare a group name with a user name since they have different semantics. To avoid such comparisons we just have to check the type of data encapsulated by the UserId class Instance by looking at the UserId's type attribute.

When comparing two UserId class instances, we first determine if the two "name" attributes represent two groups or two users. If both of the two instances carry user information, we give a similarity value of 1 if the same user is represented, a similarity value of 0.5 if they belong to the same group, and a similarity value of 0 if they do not belong to the same group. If both of the two instances carry group information, we give a similarity value of 1 if the groups are identical and a value of 0 if they are different.

3.2.4 Process data comparison

The Process class aims at providing information about the process causing the alert (for the source) and the one aimed by the event (for the target). In order to generate a similarity value, the most interesting information in this class are the "name" and "pid" aggregate classes. We simply compare the values of two instances of IDMEF alerts for those classes to give a similarity value of 1 if they are equal and 0 if they are different. The arguments of a process are ordered so we calculate the similarity values between arguments as defined in section 3 for ordered sets.

3.2.5 Service data comparison

The Service class provides information about network services running on sources and targets. The four aggregate classes that make up Service are name, port, portlist and protocol. The name class raises a problem since it is a string and that no obligation is given regarding its format. Moreover after having looked at the output of some IDSs we could see that the "name" aggregate class is not used and even sometimes the name of the service is stored in the "ident" attribute. It seems preferable to rely on the port or portlist classes since they are just a number or a formatted list of numbers. The "protocol" aggregate class is often used in the IDSs alerts we have looked at and seems to have no particular format. Hence we can use a string comparison to produce a similarity value of 0 or 1.

To deal with service ports, we use a taxonomy over the ports numbers. We distinguish the three ranges of port numbers: the Well Known Ports, the Registered Ports, and the Dynamic and/or Private Ports (figure 5). A similar taxonomy is defined in [Jul03]. A set of ports is considered an unordered set.

Two other aggregate classes are part of the Service class: the WebService and SNMPService classes. They carry additional information related to Web traffic and SNMP traffic.

- The WebService Class: Four aggregate classes make up this class (url, cgi, http-method and arg). They are all string of characters. Here string comparison is applied, hence similarity values of 1 or 0 are obtained. Since the arguments passed to the CGI scripts are separated in multiple strings, we can

get a similarity value between 0 and 1 for the list of arguments if multiple arguments are used in the request. As for the process class, the arguments of two CGIs are ordered.

- The SNMPService Class: Three aggregate classes are part of this class. They are all string values and represent the object identifier in the SNMP request (oid), the object's community string (community) and the command sent to the SNMP server (command). String comparison is applied here, hence we get 1 if strings are equal and 0 otherwise.

3.3 Similarity values aggregation

By applying the set of similarity functions on two intrusion detection alerts, we obtain a set of similarity values. In order to compute a final similarity value between the two alerts, we have to define how to aggregate those values. The first natural idea is to use the mean operator. This aggregation mode is however not desirable since somme attribute, for a given event type, may be more important than another one. For example, in the case of a spoofed source address, the similarity between this spoofed address and a real address is not significant.

The IDMEF data format is not just a set of alert attributes but also provides an object-oriented representation of the set of alert attributes. Attributes are organized in classes and form a class hierarchy. We propose to aggregate the similarity values calculated between the alert attributes by respecting this class hierarchy and we propose to associate a weight to each class and class attribute.

3.3.1 IDMEF alert structure

IDMEF alerts are represented by XML documents. Since XML cannot represent class subclassing, only aggregation relationships are defined between classes in the IDMEF model. An IDMEF alert can be represented by a graph, more precisely a tree, where nodes are IDMEF class instances or class attributes. For a more in-depth explanation of this alert format, refer to [DC04].

Note that some aggregation relationship, like the one between the *Alert* class and the *Source* class, allow to have an arbitrary number of class instances for the aggregated class. Hence we must handle the case where, for example, the two compared alerts do not have the same number of instances for the Target class. More generally we have to compare two sets of attributes not having the same number of elements.

3.3.2 Weighting the similarity values

As we said in section 3.3, some alert attributes are more significant than others given the event type associated to the alert. In order to take in account this, we associate a weight to each class type in the IDMEF data model.

3.3.3 The similarity operator

As we mentionned in the previous sections, in order to obtain a similarity value between two alerts, we must calculate the similarity values between the alerts'attributes and aggregate them to obtain a final similarity value. In our approach the two processes are done in parallel. Let us start with some definitions.

- Let E be the set of event types that can create alerts (an alert is associated to the instance of one of those types). If we consider the snort IDS for example, E will be the set of events detected by the set of snort rules. We add to this set the *unknown* event type mapped to the alerts that cannot be mapped to a known event.
- Let T be the set of attribute and class types.
- Let F_s be the set of similarity functions, each function being associated to one attribute or class type.
- Let $type(V) \mapsto T$ be the function returning an attribute type given an IDMEF alert vertex.
- Let $p : T \times T \times E \times E \mapsto \mathbb{R}$ be the function returning the weight associated to the similarity value resulting from the comparison of two vertexes. Note that this function returns the default weight associated to an attribute type if the two alerts'associated events are not of the same type.

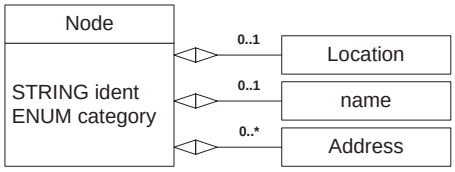

Fig. 6: IDMEF *Node* class structure

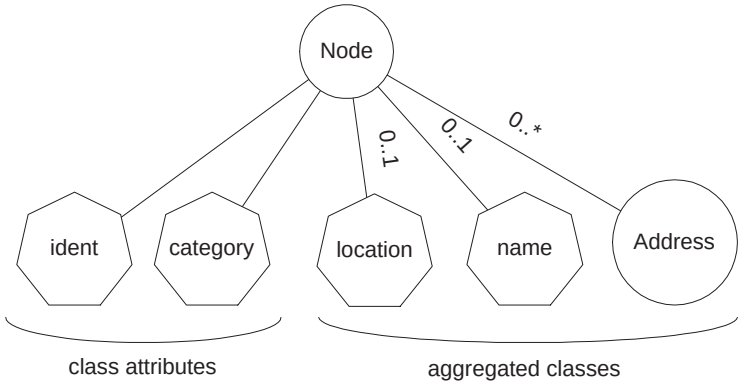

Fig. 7: IDMEF *Node* class represented as a tree

Comparing two IDMEF class instances

As we said above, we compute and aggregate the similarity values between the attributes of two alerts at the same time. In section 3.2 we presented some functions to compare the most important attributes in the IDMEF data model. Before exposing how we browse the alert structure when comparing two alerts, we explicit how to compare two IDMEF class instances. Figure 6 shows the structure of the *Node* class, we take this class as an example to illustrate the comparison of two arbitrary IDMEF classes.

Generally, a class instance of the IDMEF data model has a set of attributes (*ident* and *category* for the *Node* class), and a set of aggregate class instances (*location*, *name* and *Address* for the *Node* class). The set of aggregate class instances can have a variable number of instances since the cardinality of some aggregation relationships are not fixed. This set can be splitted up in subsets of class instances grouped by class types. Hence, comparing two instances of the same class consists in:

- comparing their attributes as ordered sets (attribute number n in the first set will be compared with attribute number n of the second set, which implies that both of them are instanciated).
- comparing the two sets of aggregate class instances by comparing their subsets by pairs of the same type.

When comparing two subsets of aggregate class instances of the same type, two situations can arise:

- the two subsets are not ordered, like for example a list of IP addresses. In this case each element of the first subset will be compared with all the elements of the second subset. Then obtaining a similarity value in $[0,1]$ is done by dividing the result by the number of comparisons made between the two sets. However, we want the comparison of two sets of instances to have some properties. Namely, when we compare a set S_C of instances of a class C with itself, we want to have $Sim(S_C,S_C) = 1$. If we compare those two equal sets by comparing each element of the first set with all the elements of the second set, and then dividing the sum of those similarity values by the number of comparisons done, we might have $Sim(S_C,S_C) < 1$. This will occur if some elements of S_C are not similar. To avoid this problem, when we are computing the similarity value between two sets S_{C1} and S_{C2}, we keep for each instance of S_{C1} compared with an instance of S_{C2} the highest computed similarity value. The same is done when comparing the instances of S_{C2} with those of S_{C1}. The following equation explicits what we said above:

$$Sim_{unordered}(S_{C1},S_{C2}) = \frac{1}{|S_{C1}|+|S_{C2}|}\{\sum_{i=0}^{|S_{C1}|} max_{j\in\{1,|S_{C2}|\}}(f_{s_m}(S_{C1}[i],S_{C2}[j])) + \sum_{i=0}^{|S_{C2}|} max_{j\in\{1,|S_{C1}|\}}(f_{s_m}(S_{C2}[i],S_{C1}[j]))\}$$

where $S_{C1}[i]$ is the i^{th} instance of class C of set S_{C1} and f_{s_m} is the similarity function to use to compare instances of class C. This way of comparing two sets of unordered instances of the same class ensures that we get a similarity of 1 when comparing two equal sets.

- the two subsets are ordered, like for instance the arguments of a process in the *Process* class. In this case when comparing two ordered subsets with the same cardinality, we compare the n^{th} element of the first subset with the n^{th} element of the second:

  $$Sim_{ordered}(S_{C1}, S_{C2}) = \frac{1}{|S_{C1}|} \sum_{i=0}^{|S_{C1}|} f_{s_m}(S_{C1}[i], S_{C2}[i])$$

  When the two subsets do not have the same cardinality, we consider the two subsets as unordered and apply the previous definition.

Aggregating the similarity values

Computing the similarity value between two instances of the same IDMEF class consists in aggregating the similarity values computed between the instances of its attributes (basic types and abstract types). The similarity values are ponderated using the weights associated with the event attached to the compared alerts. Those weights are selected thanks to the p function. We aggregate the similarity value recursively, by starting from the top node in the XML tree of two alerts (the *Alert* class). The formula for computing the similarity value between two instances C_1 and C_2 of the same class is given below:

$$Sim(C_1, C_2) = \frac{1}{\sum_k p_k} \{ \overbrace{\sum_{i=0}^{n} p_i f_{s_i}(attribute_{1_i}, attribute_{2_i})}^{S_1} + \overbrace{\sum_{a \in NO(C_1), b \in NO(C_2)} p_{NO_{ab}} \times Sim_{unordered}(a,b)}^{S_2} + \overbrace{\sum_{a \in O(C_1), b \in O(C_2)} p_{O_{ab}} \times Sim_{ordered}(a,b)}^{S_3} \}$$

S_1 represents the comparison of class attributes (basic types) instanciated in C_1 and C_2, ponderated by their weight p_i. S_2 represents the comparison of the subsets of instances of unordered attributes of the same type (aggregate classes). $NO(C)$ is the set of unordered subsets of attributes instances of an instance of the C class. $p_{NO_{ab}}$ is the weight associated to the type of compared attributes. S_3 represents the comparison of the ordered sets of aggregate class instances of C_1 and C_2, $O(C)$ being the set of subsets of ordered attributes instances of the C class. As for the unordered case, $p_{O_{ab}}$ represents the weight associated to the type of compared class instances. In S_2 and S_3, a and b represents subsets of attribute instances. $\sum_k p_k$ is the set of weight used in the calculus. It allows us to obtain a similarity value in $[0,1]$.

4 Experimentation

Our experimentation on our implementation of the aggregation tool consisted in playing some *Nmap* commands on a computer monitored by a *Snort* IDS. We did not define any event in the E set, so default weights have been used to ponderate the similarity values between the alerts attributes. T_{min} and T_{max} defined in 3.2.1 are set respectively to two and ten seconds. The aggregation tool is written in C++ and we are currently writting a Java version.

Nmap [Fyo] is a free tool which can be used to explore networks. It allows to efficiently scan a large number of IP addresses and to get various information. Actually *Nmap* can determine, among other information, the operating system of a computer, the names and the versions of running services and the installed firewalls. This tool can be used by a system administrator to quickly gather some information from the system he manages, but it can also be used by an attacker. Actually an attacker can gather information about potential targets to identify some vulnerabilities that could be exploited to start an intrusion scenario.

We used two target computers monitored by a *Snort* probe and executed the same *Nmap* command at the same time on both computers. We made this choice to see if the aggregation tool is capable of separating the alerts concerning the two computers.

We used the following *Nmap* commands:

- *Nmap -sO*: This command allows to identify the protocols used by a machine. Nmap sends IP packets without protocol header to the target machine on all possible protocols. When executing this command, Snort generated 3776 alerts in 120 seconds. After aggregating and fusioning the information inside each cluster to obtain one alert per cluster, we have 64 clusters. The first 32 clusters concerns the first target, and the 32 other the second target. The two sets of 32 clusters have the same composition, so we only analyse the 32 clusters concerning the first target. This confirms that the aggregation tool could separate the alerts depending on the target.

  Among the 32 clusters, around ten clusters have a size ranging from 30 to 400 alerts. Inside those clusters, the maximum time gap between two alerts is 10 seconds. For those clusters, the alert classification is the same and correspond to the *BAD-TRAFFIC Unassigned/Reserved IP protocol* event. Those alerts are generated because some IP packets are sent with an invalid protocol number. This number is expressed with 8 bits, so *Nmap* generates packets using the 256 possible values.

  The other clusters have a smaller size, ranging from 1 to 6 alerts. All the alerts inside a cluster have the same classification and the time gap does not exceed 2 seconds. For each of those clusters, the associated event corresponds to IP packets which have been sent with a reserved protocol number. The following classifications appear in those clusters: *BAD-TRAFFIC IP Proto 55 IP Mobility*, *BAD-TRAFFIC IP Proto 103 PIM*, *BAD-TRAFFIC IP Proto 77 Sun ND* and *BAD-TRAFFIC IP Proto 53 SWIPE*.

  Hence, we have been able to isolate the punctual events drowned into a set of alerts corresponding to the use of invalid protocols. Moreover after fusing the informations inside each cluster, we get 64 alerts, one alert per cluster, which take 104Kb. The 3776 original alerts take 3Mb.

- *Nmap -sS*: this command executes a port scan by sending *SYN* packets on each port to know if it is open. If the port is open, a *SYN-ACK* is received. Otherwise a *RST* packet is recieved. Executing this command generates 3 alerts. The gap between the first and the last alert is one second, the Snort IDMEF plugin doesn't seem to produce alert timestamps more precise than one second. The 3 alerts are aggregated together given the fact that the time gap is low, the source and target IP are the same and the source and target port numbers are identical. Moreover the target ports are part of the "well known" ports (see 3.2.5 for the port numbers taxonomy). After fusing the cluster, we get one alert with one source address, one target address, one source port and a list of 3 target ports.

Those two examples shows that the aggregation tool we have developped has an acceptable behaviour for the two tests we conducted. This is satisfactory since we did not have to configure it. We are currently modifying the tool to connect it to an alert database to enhance the cooperation capabilities of this module.

5 Conclusion

Based on the fact that using multiple IDSs in a computer network environment is a good solution to increase intrusion detection rate, we proposed a framework to reduce the number of alerts to create more synthetic alerts. The approach adopted consists in calculating a similarity value for each attribute belonging to two alerts. However our approach is generic concerning the alert data format, provided that alerts can be represented as trees. We choose the IDMEF data format because it is supported by several IDSs. Whereas in [Jul03] the alert aggregation is done off-line, we propose to process alerts in an on-line or off-line approach. The on-line alert processing can be integrated into a more sophisticated alert processing flow, the fusion alert can be forwarded to a correlation module for instance. In [Cup01] similarity relations are defined as logic predicates. However using similarity relations and no real similarity value raises a problem: an alert

may be similar to two other alerts, that are not similar. In our approach each alert is associated with only one event instance. We have presentes some experimentation with the *Aggregator* tool which implements our approach. The tests conducted with the *Nmap* tool show the benefit of alert aggregation but confirms the need to process the fusion alerts after the aggregation module. The experimentation also shows that defining weights for some event types is necessary. This needed work can be used when installing the fusion module in another network. We have implemented a correlation module which follows the principle exposed in [BAC03]. Experimental results showing the two chained modules in action will be presented in a forthcoming article. We think that our approach can also be interesting to aggregate clusters of alerts generated by our tool instead of only aggregating alerts sent by IDSs. As exposed in [Jul03], some alert clusters may be related to some configuration problems that produce bursts of alerts. Some of those problems are occuring at fixed intervals and thus have a property of periodicity. We think our approach can be interesting in this case by processing the alert clusters, created by aggregating the IDSs alerts, using different similarity functions. Namelly, the temporal similarity function can be modified to compute similarity values based on some periodicity property. This approach will be exposed in a forthcomming paper.

References

[AFV02] D. Andersson, M. Fong, and A. Valdes. Heterogeneous sensor correlation: A case study of live traffic analysis, 2002.

[BAC03] S. Benferhat, F. Autrel, and F. Cuppens. Enhanced correlation in a intrusion detection process. In *In* Mathematical Methods, Models and Architecture for Computer Network Security (MMM-ACNS 2003)*, St Petersburg, Russia, September 2003*, 2003.

[CM02] F. Cuppens and A. Miège. Alert correlation in a cooperative intrusion detection framework. In *In Proceedings of the IEEE Symposium on Research in Security and Privacy, pp. 202-215, Oakland, USA*, May 2002.

[Cup01] F. Cuppens. Managing alerts in a multi-intrusion detection environment. In *17th Annual Computer Security Applications Conference (ACSAC'01)*, 2001.

[DC04] H. Debar and D. Curry. The IDMEF format. Available at: http://www.ietf.org/html.charters/idwg-charter.html, 2004.

[DW01] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Recent Advances in Intrusion Detection, Proc. 4th Int'l Symp., RAID*, 2001.

[Fyo] Fyodor. Nmap free security scanner. http://www.insecure.org/nmap/.

[GHH+01] Robert P. Goldman, W. Heimerdinger, Steven A. Harp, Christopher W. Geib, V. Thomas, and Robert L. Carter. Information modeling for intrusion report aggregation. In *DISCEX*, 2001.

[Jul03] K. Julisch. Using root cause analysis to handle intrusion detection alarms. Phd Thesis, 2003.

[MMDD02] B. Morin, L. Mé, H. Debar, and M. Ducassé. M2d2: A formal data model for ids alert correlation. In *RAID*, pages 115–127, 2002.

[NCR02] Peng Ning, Yun Cui, and Douglas S. Reeves. Analyzing intensive intrusion alerts via correlation. In *RAID*, pages 74–94, 2002.

[VS01] A. Valdes and K. Skinner. Probabilistic alert correlation. *Lecture Notes in Computer Science*, 2212:54–68, 2001.