A Stream Clustering Algorithm for Classifying Network IDS Alerts

Risto Vaarandi

A Stream Clustering Algorithm for Classifying Network IDS Alerts

Risto Vaarandi
Centre for Digital Forensics and Cyber Security
Tallinn University of Technology
Tallinn, Estonia
firstname.lastname@ttu.ee

***Abstract*—Network IDS is a widely used security monitoring technology for detecting cyber attacks, malware activity, and other unwanted network traffic. Unfortunately, network IDSs are known to generate a large number of alerts which overwhelm the human analyst, with many alerts having low importance or being false positives. This paper addresses this issue and proposes a lightweight stream clustering algorithm for classifying IDS alerts and discovering frequent attack scenarios.**

***Index Terms*—network IDS alert classification, discovery of attack scenarios, stream clustering, network security monitoring**

I. Introduction

Over the last two decades, signature based network IDS has become a predominant security monitoring technology used by many organizations for detecting malicious network traffic [1]–[3]. However, network IDSs are known to generate many alerts, with a significant part of them having low importance or being false positives [4]–[6]. Although many alert processing methods have been suggested for reducing the workload of the human analyst [4]–[20], they suffer from several drawbacks.

First, many approaches involve supervised machine learning, semi-automated data mining, and rule based methods [4], [7], [8], [13]–[15], [17], [19], [20]. However, the creation of training data sets for supervised machine learning algorithms, interpretation of knowledge discovered with data mining methods, and rule development are time consuming activities and require human experts. Since the threat landscape is rapidly evolving and new IDS signatures are introduced at a fast pace, these activities have to be repeated periodically that further increases their cost. Second, several existing unsupervised methods [5], [6], [9] are computationally expensive and thus difficult to deploy on nodes with limited computing resources. Third, most previous works have assumed that IDS alerts have few basic attributes such as timestamp, transport protocol, source and destination IP addresses and ports, and signature ID. However, modern network IDS platforms often include additional contextual information in the alert. For example, if the alert has been raised for HTTP protocol, Suricata IDS can augment it with attributes like HTTP method and URL [1]. This contextual information is highly useful for human security analysts and should also be considered during automated alert processing. Finally, the implementations of most previously suggested methods are not publicly available.

This work was supported by research project RITA1/02-96-05, funded by the Estonian Government.

This paper addresses the above shortcomings and proposes a lightweight unsupervised stream clustering algorithm for classifying IDS alerts and discovering frequent attack scenarios. Also, we have developed an open source implementation of the algorithm for Suricata IDS and have evaluated it during 3 months in a network of a large academic institution. The rest of this paper is organized as follows – section II presents related work, section III describes the stream clustering algorithm, section IV discusses its implementation and performance evaluation, and section V outlines future work.

II. Related Work

Kidmose, Stevanovic, Brandbyge and Pedersen [4] have developed a supervised method which uses recurrent neural networks and LSA for learning a mapping function that converts textual alerts to vectors, so that alerts representing the same incident will produce similar vectors. After generating vectors for labeled IDS alerts, they are clustered with DBSCAN algorithm and each cluster is assigned a label of the majority of core points in the cluster. Detected clusters are then used for classifying incoming alerts, measuring their distance from labeled core points. Tjhai, Furnell, Papadaki and Clarke [9] have suggested the use of SOM neural networks for generating two-dimensional map from IDS alerts that is clustered with k-means algorithm. Clusters are then turned into data points, and SOM neural networks and k-means clustering are used again for dividing the points into false and true positives. During preliminary experiments on smaller data sets, the method identified most false positive alerts.

Shittu et al. [5] have proposed an unsupervised method which arranges incoming IDS alerts into graphs by similarity. A priority value is then assigned to each graph, so that graphs dissimilar to other graphs would get the highest priority. Although the method reduced the number of false positives during the experiments, it also misclassified significant part of true positives. Spathoulas and Katsikas [10] have suggested an unsupervised clustering framework which merges alerts into clusters based on similarity of timestamps, signature IDs, source and destination IP addresses. The framework is also able to detect missing security events and visualize clusters which helps the human to spot critical alert groups. Our

previous paper [6] describes an unsupervised classification method that employs frequent itemset mining and data clustering algorithms for daily mining of IDS alert logs in order to discover patterns for filtering alerts of low importance.

Long, Schwartz and Stoecklin [20] have suggested a supervised clustering algorithm that creates data points from IDS alerts of the same network session and divides the data points into clusters to distinguish normal network sessions from malicious ones. Giacinto, Perdisci and Roli [8] have proposed a supervised stream clustering algorithm for IDS alert compression, where the number of clusters and their labels (attack classes) are learned during a training procedure. Attack classes are then used by stream clustering algorithm for merging alerts of the same attack class. Long, Shen, Li and Ge [11] have described another alert compression method which is derived from the ISODATA clustering algorithm.

Julisch and Dacier [13] have suggested a conceptual clustering technique for discovering knowledge for writing alert filtering and correlation rules. Al-Mamory and Zhang [19] have proposed a clustering algorithm for identifying false positive alerts with the same root cause, so that human analyst can write filters for such alerts. Treinen and Thurimella [17] have applied an association rule mining algorithm for discovering alert patterns that represent novel attacks in order to manually update a rule base for detecting these attacks. Ma, Li and Li [7] have employed a frequent sequential pattern mining method for identifying signature ID patterns of frequent attacks and used this knowledge for developing attack detection rules.

Viinikka et al. have studied the use of EWMA control charts [16] and non-stationary autoregressive models [18] for monitoring alert flows from verbose IDS signatures to discover unexpected changes in the number of such alerts. Other IDS alert processing methods include semi-automated analysis with visualization tools [12], rule-based event correlation [14], and supervised rule-based classification [15].

As discussed in section I, the algorithm proposed in this paper is unsupervised and does not require assistance from human experts like supervised, semi-automated, and rule based methods. Also, unlike several other unsupervised methods, the algorithm is lightweight and has a low computational cost. Finally, unlike previous methods, the algorithm employs an advanced alert model that harnesses application layer data.

III. Stream Clustering Algorithm for Network IDS Alerts

The stream clustering algorithm relies on the properties of IDS alert data (described in subsection III-A) and processes incoming IDS alerts in two stages. Alerts are first divided into groups by the alert group detection module (*AlertGroupModule*, described in subsection III-B), with each group representing malicious activity for the same external host. Detected alert groups are then sent to the clustering module (*ClusteringModule*, described in subsection III-C) which either assigns the group to a cluster or regards it as an outlier. For each cluster, a cluster centroid is maintained which corresponds to some frequent alert pattern and has a human readable representation. Since the set of clusters is changing over time as new frequent alert patterns emerge and existing patterns become infrequent, detected centroids provide valuable information to human analysts about current frequent attack scenarios and emerging threats.

In addition, *ClusteringModule* calculates alert group's similarity with its centroid (henceforth called the *similarity score*). If the alert group is assigned to a cluster, the similarity score ranges from 0 to 1, with values close to 1 indicating a strong similarity with the cluster centroid. If the alert group is an outlier, it receives the similarity score of -1. Therefore, when the human analyst filters alert groups by similarity score (e.g., score < 0.5), unusual alert groups can be identified that are either outliers or dissimilar to their cluster centroid.

A. Properties of Network IDS Alert Data

For studying the properties of IDS alert data, we have used a data set that was collected on the external network perimeter of a large academic institution during 1 year from October 2019 to October 2020 (366 days). The external network perimeter was monitored by Suricata IDS with more than 40,000 signatures that generated 105,063,324 alerts. When analyzing the data, we found that 10 most frequently matching signatures produced 98,681,092 (93.93%) alerts. Also, 4 of these signatures generated alerts on all 366 days, while all 10 signatures were triggering alerts during at least 174 days (almost every second day). Although the IDS had more than 40,000 signatures, only 1,186 signatures triggered alerts during the year, while 694 signatures produced less than 50 alerts and 728 signatures produced alerts during less than 10 days. According to above findings, most alerts are triggered by a small fraction of frequently matching signatures, while most signatures are seldom generating any alerts.

When investigating alerts from verbose signatures more closely, we found that they manifest network scanning, attempts to exploit old vulnerabilities, and other events of low importance that are well known to security personnel and do not pose any threat to organizational network. Similar observations about frequently matching signatures have been reported in other papers [6], [16], [18]. These findings suggest that there are frequent patterns in IDS alert logs which match a large part of the alerts and mining these patterns will help to identify well known alerts of low importance.

We also made another observation – the same malicious activity from an external host can often trigger many alerts. For example, if a botnet member scans the external network perimeter, hundreds of alerts can appear in a short time frame. Also, many common attacks are matched by several signatures which increases the volume of alerts even further. For example, Fig. 1 displays alerts that are raised for the Plesk Apache zero-day vulnerability attack. Thus, if several alerts are triggered for the same external host in a short time frame, it is worthwhile to consider these alerts together, since they are likely to represent the same malicious activity. For the sake of brevity, any malicious activity is called *attack* in the rest of the paper.

```
# Suricata IDS alert pattern for Plesk Apache zeroday
# vulnerability attack from June 2013 (CVE-2013-4878)

ET WEB_SERVER PHP tags in HTTP POST
ET WEB_SERVER allow_url_include PHP config option in uri
ET WEB_SERVER safe_mode PHP config option in uri
ET WEB_SERVER suhosin.simulation PHP config option in uri
ET WEB_SERVER disable_functions PHP config option in uri
ET WEB_SERVER open_basedir PHP config option in uri
ET WEB_SERVER auto_prepend_file PHP config option in uri
```

Fig. 1. Example alert pattern for a frequent attack.

B. Alert Group Detection Module

The algorithm presented in this paper processes a stream of incoming IDS alerts, where *alert stream* A is defined as an infinite alert sequence $A = (a^1, a^2, ...)$. Each alert has the following attributes – occurrence time *t*, signature ID *id*, transport protocol *proto*, external IP address *extip*, external port *extport*, internal IP address *intip*, and internal port *intport*. For portless transport protocols like ICMP, *extport* and *intport* attributes are set to 0. In addition, the alert can have application layer specific attributes. For example, if the alert is triggered by the signature which matches HTTP traffic, it has additional HTTP-specific attributes such as *HttpMethod* and *HttpUrl*. This alert model reflects the behavior of modern IDS platforms which are able to add contextual data to IDS alerts for common application layer protocols. Note that attributes are not created for source and destination IP addresses and ports of alerts, since although attackers can be often identified by source IP address field, some signatures match victim responses to attacks and report the attacker in the destination IP address field. For distinguishing attackers from victims, home network address has to be provided as an input parameter for *AlertGroupModule* which is used for creating *extip*, *extport*, *intip*, and *intport* attributes from source and destination information in the alert. If a is alert and $attr$ is its attribute, a_{attr} denotes the value of attribute $attr$ for a. If A is a stream of alerts, $a^i, a^j \in A$, and $i < j$, we assume that $a_t^i \leq a_t^j$.

For grouping alerts for the same attack together, *AlertGroupModule* divides the stream of incoming alerts into *sessions*, so that all alerts in one session are associated with the same external IP address. If an incoming alert is observed for which there is no session, it is considered the first alert in the session. If more than *SessionTimeout* seconds have elapsed since observing the most recent alert in the session, the session is considered ended. Also, for preventing very long sessions and ensuring their timely processing, maximum allowed length of the session (i.e., time since observing the first alert) is *SessionLength* seconds. More formally, if $S = (a^1, ..., a^k)$ is a session of k alerts which are ordered by time of occurrence, then $a^i_{extip} = a^j_{extip}$ for $1 \leq i, j \leq k$. In addition, $a_t^i - a_t^{i-1} \leq$ *SessionTimeout* for $1 < i \leq k$, and $a_t^k - a_t^1 \leq$ *SessionLength*.

When the session S ends, all alerts in the session are merged into *alert group* G_S. Let I_S be the set of all signature IDs that triggered alerts in session S: $I_S = \{a_{id} \mid a \in S\}$. Also, let L_S be the sequence obtained by sorting the elements of set I_S. Then the set $V_S^{sig,attr}$ denotes all values of attribute $attr$ of signature sig from session S:

$V_S^{sig,attr} = \{a_{attr} \mid a \in S, a_{id} = sig\}$

Also, the set K_S^{sig} denotes all alert attributes except signature ID *id* and occurrence time *t* created by signature sig for alerts in session S. For example, if signature 23 does not create application layer attributes and triggers two alerts for internal hosts 192.168.1.1 and 192.168.1.2 during session S, then $K_S^{23} = \{proto, extip, extport, intip, intport\}$ and $V_S^{23,intip} = \{192.168.1.1, 192.168.1.2\}$.

For session S, alert group G_S is a data structure that holds tuples $(sig, attr, V_S^{sig,attr})$ for every attribute $attr$ of every signature sig from I_S:

$G_S = \{(sig, attr, V_S^{sig,attr}) \mid sig \in I_S, attr \in K_S^{sig}\}$

In other words, the alert group holds all evidence extracted from IDS alerts that describe the attack for some external host. Note that the alert group does not contain information about the order of IDS alerts in the session, since modern IDS platforms employ multiple threads or processes for network traffic analysis [1], [3] and alerts manifesting the same attack are thus not always appearing in the same order. After *AlertGroupModule* has created the alert group for the session, it will be passed to *ClusteringModule* for further handling.

C. Clustering Module

Fig. 2 presents the algorithm implemented for clustering incoming alert groups, with lines 17-32 in Fig. 2 describing the main processing loop. *ClusteringModule* maintains centroids for clusters and cluster candidates in memory, with each centroid (and relevant cluster or candidate) being identified by a sequence of sorted signature IDs. Therefore, each centroid represents a specific attack type that manifests itself by alerts triggered by the given combination of signatures.

```
1  procedure ClusteringModule(α, MaintTime,
2            MaxCandAge, CandTimeout, ClusterTimeout,
3            MinKeyValue, KeyInitValue,
4            MaxTableSize, EntropyThreshold)
5
6  after each MaintTime seconds do
7    foreach C in {set of clusters} do
8      if current_time - C.update_time > ClusterTimeout
9        drop cluster C
10   foreach X in {set of candidates} do
11     if current_time - X.update_time > CandTimeout
12       drop candidate X
13     elsif current_time - X.create_time > MaxCandAge
14       promote candidate X to cluster
15   PruneAttributeTables(MinKeyValue)
16
17 foreach G_S received from AlertGroupModule do
18   if cluster C with ID L_S exists
19     sim := FindSimilarity(G_S, C,
20                MaxTableSize, EntropyThreshold)
21     Merge(G_S, C, α, KeyInitValue)
22     C.update_time := current_time
23   elsif candidate X with ID L_S exists
24     sim := -1
25     Merge(G_S, X, α, KeyInitValue)
26     X.update_time := current_time
27   else
28     sim := -1
29     X := CreateCandidate(G_S, L_S)
30     X.create_time := current_time
31     X.update_time := current_time
32   Report(G_S, sim)
```

Fig. 2. Clustering module.

For each incoming alert group G_S for session S, the cluster with ID L_S is first looked up. If the cluster exists, alert group is assigned to that cluster, a similarity with the cluster centroid is calculated, and alert group is merged with the cluster centroid (lines 19-22 in Fig. 2). If the cluster candidate with ID L_S is found, alert group is merged with the candidate centroid and similarity score is set to -1 (lines 24-26 in Fig. 2), otherwise a new candidate is created with setting similarity score to -1 (lines 28-31 in Fig. 2). Finally, the alert group is reported with its similarity score to the end user (line 32 in Fig. 2).

After each *MaintTime* seconds (by default, *MaintTime*=10), a maintenance procedure is executed (lines 6-15 in Fig. 2) which drops clusters and cluster candidates that have not been recently updated (during the last *ClusterTimeout* and *CandTimeout* seconds respectively). If a cluster candidate has stayed in memory for more than *MaxCandAge* seconds without being dropped, it will be promoted to cluster. Therefore, the maintenance procedure ensures that the stream clustering algorithm is able to adjust to environment changes, creating clusters for new frequent alert patterns and dropping clusters after corresponding patterns have become infrequent.

For the cluster and candidate centroid, *ClusteringModule* employs the following data structure – for each attribute of each signature from the centroid ID, there is a hash table called *attribute table* which stores recently seen attribute values as keys. If X is a cluster or cluster candidate, then $X.sig.attr$ denotes the attribute table of attribute $attr$ of signature sig for the centroid of X. Also, $X.sig.attr[key]$ denotes the value of key key in attribute table $X.sig.attr$. For each attribute table key, the corresponding value ranges from 0 to 1 and represents the frequency estimate the given key (attribute value) has been seen in past alert groups. For example, if the centroid ID is (23, 98) and the table for the *intip* attribute of signature 23 holds the key-value pair *192.168.1.1=0.75*, then about 75% of previously seen alert groups have contained the *intip* attribute value 192.168.1.1 for signature 23.

For implementing frequency tracking in a memory efficient way, each value μ in the attribute table is maintained as an exponentially weighted moving average (EWMA):

$$\begin{cases} \mu_1 = x_1 \\ \mu_i = \alpha * x_i + (1-\alpha) * \mu_{i-1} \ , \ i > 1 \end{cases} \quad (1)$$

EWMA is known to estimate the average of last $(2/\alpha) - 1$ observations from time series $x_1, x_2, ...$ (e.g., see [16]). When a new candidate centroid is created for alert group (line 29 in Fig. 2), attribute tables are created for each signature in the alert group. Values for each attribute are then extracted from the alert group and keys are created from values in relevant attribute tables, initializing them to 1 (see *CreateCandidate* procedure in Fig. 3). When an alert group is merged with a candidate or cluster centroid (lines 21 and 25 in Fig. 2), keys in attribute tables are updated with either 1 or 0 according to (1), depending on whether a given attribute value is present in the alert group (see *Merge* procedure in Fig. 3). If the alert group has an attribute value that is not present as a key in corresponding attribute table, new key is created and initialized to *KeyInitValue*. *ClusteringModule* uses the default value $1/((2/\alpha)-1)$ for the *KeyInitValue* parameter to indicate that a new attribute value was seen for the first time for the last $(2/\alpha) - 1$ alert groups.

```
1  procedure CreateCandidate(G_S, L_S)
2
3  X := initialize candidate with ID L_S
4
5  foreach sig in I_S do
6    foreach attr in K_S^sig do
7      foreach value in V_S^{sig,attr} do
8        X.sig.attr[value] := 1
9
10 return X
11
12 procedure Merge(G_S, X, α, KeyInitValue)
13
14 foreach sig in I_S do
15   foreach attr in K_S^sig do
16     foreach key in X.sig.attr do
17       if key ∈ V_S^{sig,attr} x := 1 else x := 0
18       X.sig.attr[key] := α*x + (1-α)*X.sig.attr[key]
19
20 foreach sig in I_S do
21   foreach attr in K_S^sig do
22     foreach value in V_S^{sig,attr} do
23       if not exists X.sig.attr[value]
24         X.sig.attr[value] := KeyInitValue
```

Fig. 3. Centroid creation and update procedures.

For measuring the similarity between the alert group G_S and the centroid of its cluster C, we define the attribute table lookup function for the centroid of C, signature sig, attribute $attr$, and attribute value v as follows – if key v exists in table $C.sig.attr$, then $Lookup(C, sig, attr, v) = C.sig.attr[v]$, otherwise $Lookup(C, sig, attr, v) = 0$. The similarity for the centroid of C, alert group G_S, and attribute $attr$ of signature sig is defined as follows:

$$sim_{sig,attr}(G_S, C) = \frac{\sum_{v \in V_S^{sig,attr}} Lookup(C, sig, attr, v)}{|V_S^{sig,attr}|} \quad (2)$$

Also, the value of the *FindSimilarity* function (lines 19-20 in Fig. 2) is calculated as follows:

$$\frac{\sum_{sig \in I_S} \frac{\sum_{attr \in K_S^{sig}} sim_{sig,attr}(G_S, C)}{|K_S^{sig}|}}{|I_S|} \quad (3)$$

According to (3), similarity between the alert group and the centroid of its cluster ranges from 0 to 1, with values close to 1 indicating that the attribute values of the alert group have been frequently seen in the past, while lower similarity values reveal the presence of unusual attribute values. Since the similarity score -1 is assigned to outlier alert groups (lines 24 and 28 in Fig. 2), alert groups with lower similarity scores under some user-defined threshold (e.g., 0.5) represent unusual attacks which deserve closer attention from human analysts.

When calculating similarity for an attribute (see (2)), the following issue will arise – if the attribute table contains many keys with small values, lookup function will always yield a

small positive value, or 0 if key is not present. For example, this scenario is common for *extport* attribute, since many attacks are launched from a randomly selected port. In general, the presence of many keys with similar values indicates that the attribute can assume any value with an equal probability. Since no value can be regarded as unusual, a high similarity score should be returned for all values.

For detecting such cases, normalized information entropy is calculated for each attribute table. If the attribute table has k keys with values $(v_1, ..., v_k)$, the values are turned into a vector with k components $(x_1, ..., x_k)$, where $x_i = v_i/(\sum_{j=1}^{k} v_j)$, and $x_1 + ... + x_k = 1$. Normalized information entropy for attribute table is calculated as follows:

$$\begin{cases} -\sum_{i=1}^{k} x_i * log(x_i)/log(k) \ , \ k > 1 \\ -1 \ , \ k = 1 \end{cases} \quad (4)$$

According to (4), normalized information entropy ranges from 0 to 1 if $k > 1$ and is -1 for $k = 1$. Also, values close to 1 indicate that most values in the attribute table are similar. Therefore, if the attribute table for attribute $attr$ of signature sig contains *MaxTableSize* or more keys and its entropy is at least *EntropyThreshold*, $sim_{sig,attr}$ is set to 1 when calculating the value of *FindSimilarity* function (see (3)).

Since *ClusteringModule* keeps cluster and candidate centroids in memory, one important design consideration is the memory consumption of the algorithm. Fortunately, as discussed in subsection III-A, a small fraction of signatures are triggering IDS alerts, and therefore the number of centroids remains modest (it is also illustrated by experiment data in section IV). However, the attribute tables can nevertheless grow very large over time, and *ClusteringModule* employs the following technique for addressing this issue – if an attribute table has a key with a value smaller than *MinKeyValue*, the key is dropped from the table (line 15 in Fig. 2). In other words, if the attribute value has a very low frequency, frequency value of 0 is assumed. *ClusteringModule* assumes the default value *KeyInitValue* / 5 for the *MinKeyValue* parameter.

IV. Algorithm Implementation and Performance

In order to evaluate the performance of the algorithm, we have created its publicly available implementation in Perl (https://ristov.github.io/scas) that provides *AlertGroupModule* and *ClusteringModule* as UNIX command line tools. *AlertGroupModule* receives data from Suricata IDS in EVE-log format and outputs alert groups in JSON format. Currently, the module creates 25 application layer specific attributes that are supported by Suricata for HTTP, TLS, DNS, SMTP, and SSH protocols. *ClusteringModule* outputs processed alert groups in JSON format, and in our environment, alert groups are sent to ElasticStack based monitoring and visualization solution.

We have tested the algorithm during 3 months (92 days) for clustering alerts from Suricata IDS in a network of a large academic institution. The algorithm was executing on a 7 year old server with Intel Xeon E5-2620v2 CPU and Linux as an OS. During 3 months, 22,327,086 alerts were triggered by IDS (242,685.72 alerts per day). *AlertGroupModule* was running with settings *SessionTimeout*=60 and *SessionLength*=300, and divided incoming alerts into 2,236,198 groups (24,306.5 per day). On average, each group was generated for 9.98 alerts that were triggered by 1.08 signatures (the largest number of signatures per group was 64). Therefore, *AlertGroupModule* achieves a high compression rate for original alert data.

For evaluating *ClusteringModule* with different settings, we executed three instances of it by setting *CandTimeout=H**3600 and *MaxCandAge=D**24*3600 (i.e., *H* and *D* reflect the number of hours and days). Both *H* and *D* were set to 1, 3, and 10 for the first, second, and third instance respectively. In other words, smaller values of *H* and *D* create clusters for frequent alert patterns in shorter time frames, while larger values identify as clusters less frequent patterns that appear over longer periods of time. *ClusterTimeout* was set to 7*24*3600 (i.e., 1 week), *MaxTableSize* to 50, *EntropyThreshold* to 0.8, and α to 0.01 for all instances, while other parameters assumed their default values. Note that setting α to 0.01 means that attribute tables represent the attribute value frequencies for the last 199 alert groups. Table I provides detailed performance data for all instances.

TABLE I
Algorithm Performance

	Instance1	**Instance2**	**Instance3**
Avg # of clusters[a]	39.36	48.98	56.07
Max # of clusters	46	57	64
Avg # of candidates[a]	19.16	32.40	79.17
Max # of candidates	59	91	159
Max consumed memory	95,856 KB	97,584 KB	108,276 KB
CPU utilization[b]	1.44%	1.67%	1.98%
Alert groups with:			
similarity = 1	44.36%	43.90%	43.82%
0.9 $\leq$ similarity $<$ 1	18.57%	19.04%	19.39%
0.8 $\leq$ similarity $<$ 0.9	20.06%	20.63%	21.10%
0.7 $\leq$ similarity $<$ 0.8	6.30%	6.40%	6.29%
0.6 $\leq$ similarity $<$ 0.7	3.33%	3.32%	3.29%
0.5 $\leq$ similarity $<$ 0.6	0.47%	0.47%	0.46%
0 $\leq$ similarity $<$ 0.5	0.03%	0.03%	0.03%
similarity = -1	6.88%	6.21%	5.62%

[a]averages are given per day.
[b]since the implementation is single-threaded, given per one CPU core.

According to Table I, each *ClusteringModule* instance created a moderate number of clusters and cluster candidates with memory footprint of 95,856–108,276 KB, utilizing one CPU core by 1.44–1.98%. Since *AlertGroupModule* consumed 17,824 KB of memory and utilized one CPU core by 0.05%, the stream clustering algorithm has modest resource requirements and can thus be deployed on commodity hardware or directly on IDS appliances.

Also, only 5.62–6.88% of alert groups were reported as outliers, while 82.99–84.31% alert groups had the similarity score 0.8 or higher. Considering that there is one alert group for 9.98 IDS alerts and a small part of alert groups require closer inspection, the algorithm can significantly reduce the workload of security analysts with low computational costs.

The algorithm's performance is better or similar to other methods that detect irrelevant alerts and have achieved alert reduction rates of 42.6–98.7% [5], [6], [9], [13], [15], [19].

When alert groups are divided into classes by similarity score (see Table I), three instances produce classes of fairly similar size, since most alert groups are assigned to the same clusters detected by all three instances. However, larger values of H and D allow for the detection of more attack types. Our implementation supports on-demand reporting of cluster centroids in human readable format which provides security analysts with useful insights into common attack scenarios. In our environment, detected cluster centroids have helped to identify the activity of new botnets and other novel threats (see Fig. 4 for few examples of detected centroids).

```
# A centroid for one signature representing Zyxel NAS device
# remote command injection vulnerability (CVE-2020-9054)
# attack. Note that past attacks have always involved the
# same malicious URL (its attribute table value is 1).

ET EXPLOIT Zyxel NAS RCE Attempt Inbound (CVE-2020-9054) M1
...
  Attribute HttpUrl:
    /adv,/cgi-bin/weblogin.cgi?
      username=admin%27%3Bls%20%23&password=asdf = 1

# A centroid for three signatures that represent DrayTek
# Vigor router vulnerability (CVE-2020-8515) attack.
# Note that over 99% of past attacks have originated from
# Hoaxcalls botnet that employs user agent string "XTC".

ET WEB_SERVER WebShell Generic - wget http - POST
ET WEB_SERVER 401TRG Generic Webshell Request -
  POST with wget in body
ET EXPLOIT Multiple DrayTek Products Pre-authentication
  Remote RCE Inbound (CVE-2020-8515) M2
...
  Attribute HttpUserAgent:
    XTC = 0.999999999999994
```

Fig. 4. Examples of cluster centroids that represent common attack scenarios.

In order to evaluate the precision and recall of the algorithm for identifying unusual attacks, we measured its performance during external pen-testing and during common network scanning activity generated with several scanning tools. We expected alert groups from pen-testing being classified as outliers, and alert groups from scanning being assigned to clusters with high similarity scores. Pen-testing and scanning activities triggered 115,770 and 53,671 alerts which were divided into 76 and 28 alert groups respectively. 75 alert groups of unusual attacks were classified as outliers by all three instances, while 1 alert group received a high similarity score of 0.8. Assuming that alert groups with similarity scores 0.8 or higher are of low importance and represent routine scanning, recall would be 75/76 $\approx$ 98.68%. Also, 16 alert groups of scanning received the similarity score of 1, while for 12 alert groups the similarity scores ranged from 0.88 to 0.9, yielding the precision of 100%. If precision is estimated conservatively and 12 alert groups of scanning with similarity scores below 1 are regarded as false positives, precision would still remain relatively high: 75/(75+12) $\approx$ 86.21%. These results are comparable to the performance of other recent methods that have achieved the precision of 86.5–99.6% and recall of 71.9–99.9% [4]–[6], [9].

V. Future Work

Currently, the stream clustering algorithm is not able to detect a sudden increase in the volume of alert groups with high similarity scores. However, such changes might indicate a DDoS attack or increased botnet activity. For addressing this issue, we plan to augment the algorithm with time series analysis methods. We are also considering the development of other similarity functions for the clustering module.

References

[1] https://suricata-ids.org
[2] https://www.snort.org
[3] https://www.cisco.com/c/en/us/products/security/ngips/index.html
[4] Egon Kidmose, Matija Stevanovic, Søren Brandbyge, and Jens M. Pedersen, "Featureless Discovery of Correlated and False Intrusion Alerts," IEEE Access, vol. 8, 2020, pp. 108748–108765.
[5] Riyanat Shittu, Alex Healing, Robert Ghanea-Hercock, Robin Bloomfield, and Rajarajan Muttukrishnan, "OutMet: A New Metric for Prioritising Intrusion Alerts using Correlation and Outlier Analysis," 2014 IEEE Conference on Local Computer Networks, pp. 322–330.
[6] Risto Vaarandi and Kārlis Podiņš, "Network IDS Alert Classification with Frequent Itemset Mining and Data Clustering," 2010 International Conference on Network and Service Management, pp. 451–456.
[7] Jie Ma, Zhi-tang Li, and Wei-ming Li, "Real-Time Alert Stream Clustering and Correlation for Discovering Attack Strategies," 2008 International Conference on Fuzzy Systems and Knowledge Discovery, pp. 379–384.
[8] Giorgio Giacinto, Roberto Perdisci, and Fabio Roli, "Alarm Clustering for Intrusion Detection Systems in Computer Networks," 2005 International Workshop on Machine Learning and Data Mining in Pattern Recognition, pp. 184–193.
[9] Gina C. Tjhai, Steven M. Furnell, Maria Papadaki, and Nathan L. Clarke, "A preliminary two-stage alarm correlation and filtering system using SOM neural network and K-means algorithm," Computers and Security, vol. 29, 2010, pp. 712–723.
[10] Georgios P. Spathoulas and Sokratis K. Katsikas, "Enhancing IDS performance through comprehensive alert post-processing," Computers and Security, vol. 37, 2013, pp. 176–196.
[11] Chun Long, Hanji Shen, Jun Li, and Jingguo Ge, "An SR-ISODATA Algorithm for IDS Alerts Aggregation," 2014 IEEE International Conference on Information and Automation, pp. 92–97.
[12] Damien Crémilleux, Christophe Bidan, Frédéric Majorczyk, and Nicolas Prigent, "VEGAS: Visualizing, Exploring and Grouping AlertS," 2016 IEEE/IFIP Network Operations and Management Symposium, pp. 1097–1100.
[13] Klaus Julisch and Marc Dacier, "Mining Intrusion Detection Alarms for Actionable Knowledge," 2002 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 366–375.
[14] Benjamin Morin and Hervé Debar, "Correlation of Intrusion Symptoms: an Application of Chronicles," 2003 International Symposium on Recent Advances in Intrusion Detection, pp. 94–112.
[15] Tadeusz Pietraszek, "Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection," 2004 International Symposium on Recent Advances in Intrusion Detection, pp. 102–124.
[16] Jouni Viinikka and Hervé Debar, "Monitoring IDS Background Noise Using EWMA Control Charts and Alert Information," 2004 International Symposium on Recent Advances in Intrusion Detection, pp. 166–187.
[17] James J. Treinen and Ramakrishna Thurimella, "A Framework for the Application of Association Rule Mining in Large Intrusion Detection Infrastructures," 2006 International Symposium on Recent Advances in Intrusion Detection, pp. 1–18.
[18] Jouni Viinikka, Hervé Debar, Ludovic Mé, Anssi Lehikoinen, and Mika Tarvainen, "Processing intrusion detection alert aggregates with time series modeling," Information Fusion Journal, vol. 10(4), 2009, pp. 312–324.
[19] Safaa Al-Mamory and Hongli Zhang, "Intrusion detection alarms reduction using root cause analysis and clustering," Computer Communications, vol. 32, 2009, pp. 419–430.
[20] Jidong Long, Daniel Schwartz, and Sara Stoecklin, "Distinguishing False from True Alerts in Snort by Data Mining Patterns of Alerts," 2006 SPIE Defense and Security Symposium, pp. 62410B-1–62410B-10.