Go With the Flow: Clustering Dynamically-Defined NetFlow Features for Network Intrusion Detection with DynIDS

Luis Dias[1,2] Simão Valente[1,2] Miguel Correia[2]
[1]CINAMIL, Academia Militar, Instituto Universitário Militar – Portugal
[2]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa – Portugal
dias.lfxcm@exercito.pt valente.spa@exercito.pt miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—The paper presents DynIDS, a network intrusion detection approach that flags malicious activity without previous knowledge about attacks or training data. DynIDS dynamically defines and extracts features from network data, and uses clustering algorithms to aggregate hosts with similar behavior. All previous clustering-based network intrusion detection approaches use a static set of features, restricting their ability to detect certain attacks. Instead, we use a set of features defined dynamically, at runtime, avoiding that restriction without falling into the curse of dimensionality, something that we believe is essential for the adoption of this kind of approaches. We evaluated DynIDS experimentally with an evaluation and a real-world dataset, obtaining better F-Score than alternative solutions.**

***Index Terms*—network intrusion detection, clustering, feature engineering, security analytics**

I. Introduction

The unstoppable growth of cyberattacks [1], raises the need for research in new methods for intrusion detection. Interestingly companies take many days to detect some attacks, e.g., roughly 58 days [2]. This number shows that the large variety of *real-time* prevention (e.g., packet filters of different sorts) and detection (e.g., malware detectors) mechanisms deployed do not provide enough protection. Hence, organizations have to dig into traffic and logs to search for anomalous patterns in *larger windows of time*.

Most approaches for configuring intrusion detection systems (IDS), more specifically *network intrusion detection systems* (NIDS) that are the focus of this work, require either knowledge about attacks (to define signatures/rules) or clean training data (to configure anomaly detectors) [3]. The first tends to be incomplete, whereas the second is hard to obtain in systems in production. Moreover, the constant evolution of attacks and the inherent dynamism of computer networks create severe difficulties for traditional NIDSs, letting them unable to detect novel attacks, or generating a high number of false positives.

A more recent approach to intrusion detection uses machine learning (ML) techniques, *clustering or outlier detection*, to identify entities – typically users or hosts – that have an anomalous behavior in a period of time, unobservable in real-time [4]–[14]. This approach is interesting because it does not require knowledge about attacks (signatures/rules) or clean training data. However, most of these approaches suffer from *two serious flaws not yet investigated:* (1) they consider a static group of features; and (2) they consider few features when, in practice, many relevant features can be derived from network traffic, e.g., those related to service/ports. In the following two paragraphs, we consider each of these aspects in turn.

In relation to (1), in the related work that uses clustering techniques for network intrusion detection – summarized in Table I –, the feature engineering process defines a set of static features, e.g., the sum of packets sent to port 22-SSH or to port 194-IRC. Then, in every clustering iteration, at runtime, the predefined set of features is used. The choice of the features is based on knowledge of the domain, such as knowledge of TCP/UDP ports commonly associated with security problems (e.g., port 22-SSH is often brute-forced). However, this feature pre-selection clearly limits the system's ability to detect attacks that are not related to those features (e.g., brute forcing an SSH server listening on a non-standard port).

Concerning (2), none of the related work uses more than 52 features (see table). Some mention that it is possible to increase the number of features, but none explores further that possibility and assesses the impact on performance. Moreover, selecting a broader range of port-based features is problematic as there are around 1000 system ports (also known as well-known ports) plus 10000 user ports assigned (also known as registered ports) [15], [16]. In fact, simply increasing the number of features is far from innocent as it may lead to a phenomenon called the *curse of dimensionality* [17]–[20]: with many features, typically more than 1000, clustering no longer works as expected as relevant features are masked by others and geometry behaves nonintuitively in high dimensions [18], [19], [21]. This issue

TABLE I
Comparison of related approaches

Reference	#features	#ports	Definition	Algorithms
BotMiner [4]	52	0	static	X-Means
Yen et al. [7]	9	0	static	PCA, K-Means
NADO [5]	41	0	static	K-Means
Bhuyan et al. [6]	50	0	static	TreeCLUS
UNIDS [8]	9	0	static	SCC with DBSCAN
FIRMA [10]	11	0	static	FIRMA
Beehive [9]	8	0	static	K-Means
Gonçalves et al. [11]	34	0	static	EM
Bhuyan et al. [12]	25	0	static	TCLUS
FlowHacker [13]	17	5	static	K-Means
OutGene [14]	26	4	static	K-Means
DynIDS	*400+*	100	*dynamic*	*K-Means, Agglomerative, DBSCAN*

prevents, e.g., having features for all ports, as they are many more than 1000. Hence, most works use generic features, e.g., overall sum of bytes sent, disregarding valuable port-based features, e.g., bytes sent on a specific port (see table).

We propose a method to *define features at runtime*, dynamically, according to data analyzed in each time window. That is, our approach defines which features should be used in the clustering process, by analyzing the network data corresponding to a specific period time (e.g., 60 minutes). The idea is novel and appealing: *dynamically defining traffic features* based on network flows (that we will designate *netflow* after the original Netflow [22], although there are now several others [23], [24]), within a specific time period.

We have made an initial experimental analysis in order to understand which number of features is desirable and if there are advantages in increasing the number of features (e.g., at the limit having one feature generated for each port used in a time period). For this purpose, we studied the theoretical and experimental complexity of several clustering algorithms in order to select those that could support the analysis of data with higher dimensions (i.e., high number of features) and more volume (characteristic of computer network data). The algorithms we have chosen later proved to perform well in terms of detection capability and complexity.

We present DYNIDS, a *network intrusion detection approach* that can dynamically *define* and extract features from network data, and uses a clustering ensemble to aggregate hosts with similar behavior, analyzed in different time windows. Similarly to Marchetti et al. [25], our approach searches for anomalous behaviors submerged on the behavior of thousands of hosts, e.g., among the hosts of a large organization. However, our approach derives features based on port/service communications during the analyzed time windows. According to insights of attacker techniques from MITRE's Adversarial Tactics, Techniques & Common Knowledge (ATT&CK) framework [26], [27], we choose to define features based on the *top-used* ports and *less-used* ports. More specifically, at runtime for each analyzed period, we select not only ports/services that are more often used (e.g., for detection of top talkers, vulnerability scans or brute-force attacks) but also ports/services that are less used (e.g., for detection of network recognition or vulnerability scans) or used by few machines (e.g., for detection of command&control communications, Trojans). Regarding the cluster ensemble, DYNIDS uses three different algorithms based on different strategies: partition-based (K-Means), hierarchical (Agglomerative), and density-based (DBSCAN). To improve the performance of the proposed approach, we made an ensemble of these algorithms calculating an outlier score according to the interception results obtained.

We concluded that defining port-based features at runtime and for each time-period analyzed provides a significant improvement in the detection of attacks that generate traffic in different ports (e.g., scanning, brute-force or DoS). This approach translates into an ability to detect unknown attacks, i.e., attacks on which there is no signature, without training the system, and without having to know in advance which features are associated with that attack. Furthermore, our experiments suggest that by increasing the number of features, we better characterize the data, i.e., machines with similar behavior are more precisely grouped (e.g., web servers, print servers, department X, Y, Z machines). However, in some cases, it becomes more challenging to obtain outliers using clustering because the features that contribute to highlight the outlier lose weight. We can conclude that if we want an outlier detector based on clustering, the features used shall be related to the anomalous behavior we want to find. Our dynamic feature definition feature handles this.

We evaluated DYNIDS experimentally with a netflow dataset publicly available (CIC-IDS-2018 [28]) and real traffic data obtained at a large military infrastructure. The source code is freely available for download[1]. Our approach achieved an overall F-Score of 0.97 for the public dataset, which is an excellent performance and outperforms related approaches from the literature and alternative approaches. The evaluation with the real-world dataset detected not only the emulated attacks with high recall, but also unexpected anomalies that required further investigation. We also compared DYNIDS with two recent schemes: FlowHacker [13] and OutGene [14]. DYNIDS's F-Score was always better than the other two.

II. BACKGROUND

This section provides an overview of the clustering approach for intrusion detection and explains how we have chosen the algorithms to apply in DYNIDS.

A. Clustering approach

The amount of digital data is growing fast, mostly due to the Internet of Things. Having so much data creates a great problem for security systems and analysts since they must search through a lot more data. Security data processing is often regarded as a big data problem [29]. Managing such an amount of data is beyond human capabilities, and the usage of machine learning methods is becoming more useful to extract information from vast and multi-dimensional data.

Machine learning techniques are commonly divided into two main categories, although there are others [30]: supervised and unsupervised learning. *Supervised learning* requires training data, typically manually labeled by humans. Supervised learning has been used in some *misuse-based NIDSs* to classify traffic in two classes: malicious or not. As humans label data, this approach looks for previously known attacks.

On the other hand, *unsupervised learning* algorithms do not require labeled data. Instead, they may be used to infer unknown classes based on data similarity, a problem called *clustering*. Typically, unsupervised learning methods used in the security domain can be considered to be a sub-category of *anomaly-based* intrusion detection [31], [32]. However, unlike classical anomaly-based NIDSs, NIDSs based on clustering, e.g., those in Table I, do not require clean training data. Clustering algorithms are applied over feature vectors, each vector representing, e.g., a machine or a user, and cluster

[1] https://github.com/a3ceProject/DynIDS

the entities (machines, users) with similar behavior, i.e., with similar feature values. This is particularly useful when we are trying to create a system to detect unknown attacks or anomalous behavior. The key idea is that big clusters represent normal behavior and the outliers (i.e., small clusters of entities or noise) can correspond to anomalous behavior. However, different clustering algorithms have different initializations and produce different data partitions [33] according to the shape and structure of data. One option to overcome the limitations of a single clustering technique is to combine different clustering techniques.

B. Clustering algorithms

The clustering algorithms we considered can be classified as partition-based, hierarchical, and density-based.

K-Means [34] is a landmark in clustering [33], [35] and the most popular *partition-based* clustering algorithm. It is known to produce good results in the context of intrusion detection [9], [13], [14], as in many other areas [33], [35]. K-Means clusters are represented by a central vector (cluster mean). Consider n the number of d-dimensional vectors (to be clustered), k the number of clusters and i the number of iterations needed until convergence. To find the global optimum of K-Means is considered an NP-Hard problem. Hence, the practical approach is to find a local optimum, which takes linear time $O(nkdi)$. Hence, in practice, K-Means has linear complexity.

For *hierarchical* algorithms we use an algorithm that we designate Agglomerative clustering, although the term is somewhat generic [36]. This class of algorithms is not as far as popular as K-Means but has been shown to provide good results with large numbers of data items (our case) and clusters (not our case) [37], [38]. The algorithm starts with each object being considered a cluster. Then, it computes pairwise distances of n data-points and links those together according to a linkage function (e.g., minimum distance). The results can be represented by a dendrogram where the root node represents the whole dataset (single cluster) and each leaf node is a data object. The complexity of the algorithm is $O(n^2)$, mainly due to the cost of computing all pairs of distances.

Regarding *density-based* clustering, both Density-Based Spatial Clustering of Applications with Noise (DBSCAN [39]) and Ordering Points To Identify the Clustering Structure (OPTICS [40]) are well-known. We selected DBSCAN for reasons explained below. DBSCAN has been shown to be efficient [39], then criticized [41]; currently it is known to be able to perform well but to be somewhat sensitive to proper configuration [42]. For each point of the dataset, DBSCAN groups together points with many nearby neighbors and marks as outlier points that lie alone (i.e., appear in low-density regions). It takes as inputs the epsilon-neighborhood (the radius) and MinPts (the minimum quantity of points within radius) parameters. In short, DBSCAN generates a new cluster from a data point by absorbing its neighborhood. OPTICS is heavily inspired in DBSCAN , but does not explicitly segment the data into clusters. Instead, it produces a visualization of reachability distances and uses this visualization to cluster the data. OPTICS overcomes the problem of DBSCAN's poor performance when clusters have varying density. DBSCAN and OPTICS have time complexity of $O(n \times log(N))$ and $O(n^2)$ respectively.

TABLE II
CLUSTERING ALGORITHMS COMPLEXITY

Algorithm	Method	Complexity	Ref.
K-Means	Partition	$O(nkdi)$	[39]
Agglomerative	Hierarchical	$O(n^2)$	[36]
DBSCAN	Density	$O(n \times log(N))$	[39]
OPTICS	Density	$O(n^2)$	[40]

In Table II we summarize the studied algorithms. For further details, Xu et al. [43], [44] provide a survey of clustering algorithms describing the theoretical time complexity of each clustering algorithm. They refer that K-Means and DBSCAN perform well on large-scale data. Our experimental evaluation of clustering algorithms is according to these theoretical results.

a) Experimental analysis: To decide which algorithms to use, we tested the most commonly used algorithms in each category (partition-based, hierarchical and density-based). The criteria used to choose the most appropriate clustering algorithms were:

- *Performance* with data with a high number of features (i.e., high dimensional data);
- *Ability* to identify an attacker in a (labeled) dataset as an outlier (i.e., isolating the attacker IP as a single cluster).

These initial experiments allowed us to evaluate the performance of several algorithms with high-dimensional data and their efficiency in identifying attacks. For the first criteria, we used a benchmark from the hdbscan clustering Python library [45], adapted in order to test the algorithms in terms of performance against large datasets. For the second criteria, we used a private labeled dataset, from a real administrative network, known to produce good results with K-Means in a previous work [14]. The labeled dataset contains network traffic flows from one day. This day contains flows from 4616 entities, one of which is an attacker that performed a port scan and a dictionary attack.

Most of the algorithms used were those provided by the scikit-learn Python library [46]: K-Means, DBSCAN, Agglomerative, and OPTICS. We tested the algorithms, with the default configuration and using the data and features from [14], in order to understand the algorithms ability to detect attacks. From the algorithms evaluated, we shortlisted K-Means, Agglomerative, and DBSCAN, as they performed much better than the rest. Given these results we decided to use an ensemble of K-Means, Agglomerative, and DBSCAN in DYNIDS.

III. DYNIDS OVERVIEW

Inspired in previous works, DYNIDS does not rely on knowledge about what is bad behavior, as in signature-based methods, or what is good behavior, as in typical anomaly detection. As in previous works, DYNIDS uses clustering to group entities (e.g., hosts) with similar behavior. That behavior is characterized by features extracted from netflow [22]). In DYNIDS the entities are hosts, identified by an IP address.

As explained in the introduction, there is a set of literature that follows generically our approach, summarized in Table I.

Most works aim to provide *general-purpose NIDSs*, but they have all the limitation of defining the features (some related to ports) in advance, i.e., before runtime, and in small numbers. For example, in [14], a set of 16 predefined port-based features was used, specifically for ports: 80-HTTP, 194-IRC, 25-SMTP and 22-SSH. For each port, four features were created: count of packets sent and received in each port, as the source or destination host. Notice that used ports are very interesting features, as they are used in various ways, notably: as endpoint process identifiers and as application protocol identifiers [15].

Instead of having predefined features, we explore the idea of having different port-based features according to the traffic in each analysed time window. Our key ideas that go beyond related work are: (1) by choosing specific port features we are limiting the system's ability to detect attacks related only to that port; (2) the use of certain ports/services can vary over time and this information can be extracted from the traffic itself; (3) often the services or attacks may be running in different ports than standard or known ones; (4) besides observing and deriving features from frequently used ports, it should also be interesting to derive features from less frequently used ports. As an example, adversaries may conduct command&control (C2) communications over a non-standard port [47] or may attempt to get a listing of services running on remote hosts [26].

Hence, our intuition is that we should define which features to extract dynamically, at runtime, both to consider all relevant ports and to avoid considering too many ports, which would lead to the curse of dimensionality. In the case of netflow events, our insight is that for each time window analyzed there are at least three types of different port-based features, derived from: (1) much used ports, or that are used a lot (e.g., brute force, DoS); (2) uncommon ports, i.e., ports that are used by few hosts (e.g., can reveal worm propagation, reconnaissance activities or botnet communication); (3) ports that appear in very few flows (e.g., detecting probes to non-existent services). This same idea can be applied to other types of data sources, such as Windows OS events. In this case, we could use features derived from eventID (i.e., less frequent eventID in addition to the most frequent). However, in the paper we focus in network flows.

Another key challenge that DYNIDS addresses, is that different clustering algorithms produce different partitions of data [8]; even different initialization or parameters can give different results for the same algorithm. To avoid the limitations of a single algorithm, we propose combining a set of clustering algorithms. The partial results of each algorithm are translated into a scoring scheme that we detail in the next section.

IV. DYNIDS DESIGN

This section presents the details of DYNIDS design and some implementation aspects. As already mentioned, DYNIDS extracts features from netflow data to group hosts based on their traffic characteristics. Hence, the approach is divided into feature engineering, dynamic feature definition, normalization & parameter inference and clustering ensemble and outlier scoring. Each of these aspects will be described next.

TABLE III

THE *fixed features* WITH SOURCE IP AS AGGREGATION KEY

Feature	Description
SrcIPContacted	# of different IPs contacted by an entity
SrcConnMade	# of flows where the entity is the source
SrcPortUsed	# of different src ports used by an entity
SrcPortContacted	# of different dst ports contacted by an entity
SrcTotLenRcv	Sum of total packets length received by an entity
SrcTotLenSent	Sum of total packets length sent by an entity

A. Feature engineering

Each flow is analyzed using as an aggregation key either the source IP (SrcIP) or the destination IP (DstIP). The idea is to capture 1-to-1 (e.g., authentication brute-forcing), 1-to-N (e.g., probe, worm), and N-to-1 (e.g., DDoS, botnet C2) anomalies. As an example, consider a feature that counts the number of different ports contacted. This feature would highlight an attacker executing a port scan [48] as SrcIP (i.e., contacted many ports). On the other hand, the victim would be highlighted by other features as DstIP (i.e., received contacts to many ports).

DYNIDS extracts a set of 12 *fixed features* and a set with a variable number of dynamically defined *port-based features* proportional to the number (x) of selected ports. The first half of the 12 fixed features, with source IP as the aggregation key, is shown in Table III. These fixed features describe general network activity of an entity (i.e., IP address). The other 6 are similar but for the destination IPs, thus beginning with Dst.

In addition to the fixed features, DYNIDS dynamically defines 4 features for each selected port. To improve explainability, each port-based feature is tagged with a: T (Top) for most used ports; M (Min) for the least used ports; and U (Uncommon) for ports used by few hosts. For example, consider that port 80 is the port with the highest packet count (i.e., number of packets sent to, or received from, port 80) from all flows for a given time window being analyzed. Hence, port 80 is a T-top port and would be selected to define 4 features: (1) T80SrcFrom, # of packets sent from port 80; (2) T80SrcTo, # of packets sent to port 80; (3) T80DstTo, # of packets received on port 80; (4) T80DstFrom, # of packets received from port 80. This variable set of features (4 for each selected port) is obtained with different port selection algorithms that we define next.

B. Dynamic feature definition

DYNIDS extract features from *netflow* data in multiple time windows, following OutGene [14]. The idea is to analyze the stream of events in different time windows, at different time scales, so that we can detect attacks independently of the pace at which they are executed (e.g., avoiding evasion techniques such as a slow network scan). For example, an attack may be detected if we analyze traffic at the scale of one hour, but not at the scale of one day or one minute. Hence, the approach can be executed on a *base time window* of duration $\mathcal{B}$ (see Figure 1).

DYNIDS dynamically defines which port-based features (four to each selected port) to extract at runtime. The algorithm, which we name DYN3_x, serves as the basis for this dynamic

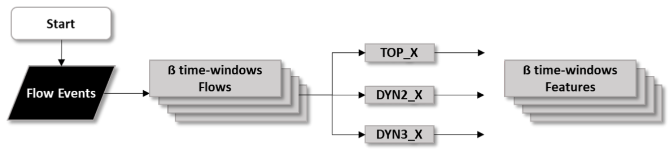

Fig. 1. Flowchart of the dynamic feature definition process

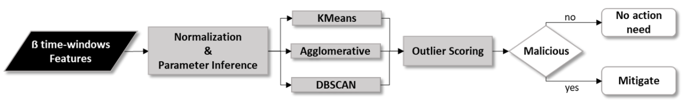

Fig. 2. Flowchart of the clustering process

definition of port-based features. This algorithm derives features from the most and least used ports and the ports used by fewer machines. To compare with other approaches and show the benefits of the chosen one, we define three variants:

- TOP_x: features based on the x ports that appear in more flows;
- DYN2_x: features based on the $x/2$ ports that appear in more flows and the $x/2$ ports that appear in fewer flows;
- DYN3_x (the DYNIDS algorithm): features based on the $x/3$ ports that appear in more flows, the $x/3$ ports that appear in fewer flows, and the $x/3$ ports used by fewer machines.

The idea of having different algorithms is to explore different strategies to generate features in order to understand the advantages and limitations of each one through the experimental analysis (see Section V). Also, the variable x, allow exploring the effects of decreasing/increasing the number of features. However, in runtime, only one of these algorithms should be used with a fixed x.

Next, we define the search space for selecting the port-based features. According to RFC 6335 [15], port numbers are assigned in various ways, based on three ranges: System Ports (0-1023), User Ports (1024-49151), and Dynamic and/or Private Ports (49152-65535). The first two groups are available for service identifier and assignment through IANA, although many are not currently assigned [16], while the later must not be used as a service identifier. Having this in mind, we limit our search space for the most used ports and uncommon ports, within the range of System and User Ports (0-49151). The search space for less frequently used ports was limited to System Ports (0-1023) only, the range more prone to probes and scans. It is worth to refer that we tried other alternatives (e.g., using the entire port range for all types of port-based features), although with less success.

C. Normalization and parameter inference

The extracted features for each $\mathcal{B}$ time window must be normalized before being given to the clustering algorithms, as their values can vary significantly (see Figure 2). For example, if we chose Euclidean distance as a distance measure for clustering, normalization can assure that every feature will contribute proportionally to the final distance. In order to perform normalization, min-max scaling has to be used: $x' = (x - min(x))/(max(x) - min(x))$, where $min(x)$ and $max(x)$ represent range values. This method returns feature values within range [0,1]. The most obvious alternative would be to use logarithmic scaling, but it would mitigate the differences between values, making detection harder (we observed it experimentally).

After normalization, a critical decision is to select the parameters of the clustering algorithms correctly, e.g., the K for K-Means, a non-trivial task [49]. Since each time window can have a different number of entities and features, data can vary significantly. Thus, fixing the number of clusters (for K-Means and Agglomerative) or *epsilon* (for DBSCAN) would not be a good choice since it could be unfit to that specific data. To solve this problem, we propose applying the *elbow method* to each time window. The idea of this method is to test various numbers of clusters in order to achieve the optimal number of clusters, i.e., to choose a number of clusters K such that adding another cluster does not improve much better the total within-clusters sum-of-squares (WCSS). In the case of the DBSCAN epsilon parameter, the distances between each entity and its neighbors are calculated and sorted. A suitable value for epsilon is where the change is most pronounced [50]. All clustering algorithms were set to use Euclidean distance.

D. Clustering ensemble and outlier scoring

The goal of using clustering is to group machines with similar behavior. The behavior is defined by the 12 fixed features and the port-based features, which are defined dynamically from network traffic in each time window, by inspecting the flows observed in that window. The assumption that is made is that machines that behave differently from the majority are anomalous. This anomaly can indicate the machine is suffering or performing an attack. Hence, we use clustering to detect anomalies in an unsupervised way. However, besides the possibility of producing different results, the various clustering algorithms deal differently with different shapes of data [33]. To avoid the lack of robustness of a single clustering algorithm, we propose combining the results of different algorithms (K-Means, Agglomerative and DBSCAN) based on multiple clustering strategies (partition, hierarchical and density-based). Several classification methods can be used and, if needed, manual inspection can be performed by a security analyst, starting with the smallest clusters. However, to automate the identification of anomalies, we consider an outlier as an entity that is isolated in a cluster itself. The disadvantage of this approach is that this method does not work when there are several machines with the same anomalous behavior (i.e., they are isolated in a cluster with more than one entity).

Finally, a score is assigned to every outlier. The score can have 3 weights: (1) very high confidence, when the same outlier is given by all the three algorithms; (2) high confidence, if the outlier is given by two algorithms; and (3) low confidence, when the outlier is given by only one algorithm. The human analyst may intervene or not according to the priority given to outliers. Trivially, if no algorithm produces outliers, no action is required.

TABLE IV
METRICS USED IN THE EVALUATION

Metric	Meaning/Formula
True Positives (TP)	entities correctly classified as outliers
False Positives (FP)	entities wrongly classified as outliers
True Negatives (TN)	entities correctly classified as inliers
False Negatives (FN)	entities wrongly classified as inliers
Precision (PRE)	TP / (TP+FP)
Recall (REC)	TP / (TP+FN)
F-Score	2 $\times$ PRE $\times$ REC / (PRE+REC)

V. EXPERIMENTAL EVALUATION

To develop and implement DYNIDS for evaluation, we used Python (v3) [51]. Additionally, we used well-known libraries such as Pandas [52] for data manipulation, scikit-learn [46] for data processing and clustering algorithms, and matplotlib [53] to get heatmaps to aid visualization of features that are relevant in identifying outliers. All the experiments, were done in commodity hardware (Intel(R) Core™ i7-8750H CPU @ 2.2GHz with 16GB RAM).

The focus of the experiments is: (1) the analysis of results when increasing number of features; (2) the comparison of different approaches for the dynamic feature definition; (3) the improvements obtained by the cluster ensemble; (4) and performance evaluation.

a) Evaluation Metrics: We consider an outlier to be a host, identified by an IP address, isolated in a cluster (one entity cluster). The expressions in Table IV can be translated into: (1) *Precision* the fraction of outliers that are real (i.e., true positives); (2) *Recall* the fraction of outliers that are correctly classified as such by the detector; and (3) *F-Score* a global detection score. Another metric, accuracy, is frequently used in this context, but it is misleading with unbalanced datasets, which are essentially all realistic cases. Therefore, we avoid using accuracy, and we privilege F-Score, which summarizes the overall performance.

The results presented in the following sections, consider the outliers with very high confidence, i.e., those flagged by all the three clustering algorithms (see Section IV-D). Although we did experiments with other time windows, we present the results only for 10 and 60 minutes for lack of space. Moreover, regarding the CIC-IDS-2018 dataset, we have made feature extraction and the clustering process with both internal and external entities as aggregation keys. However, for simplicity, we considered for evaluation only the results for the internal machines, which are the ones we are interested in protecting.

A. Dataset characterization

We used two datasets containing netflow events for the experimental evaluation: a public synthetic dataset provided by the Canadian Institute for Cybersecurity (CIC-IDS-2018 [28]) and real traffic flows (private and confidential) obtained at a large military infrastructure. The information about the datasets is summarized in Table V. The public dataset was used for a comprehensive evaluation that we describe next, while the real dataset was used to validate the approach in a real-world scenario.

TABLE V
SUMMARY OF THE DATASET CHARACTERISTICS

Dataset	Size	Num. events	Num. hosts
CIC-IDS-2018	5.7GB	82,108,448	450 (internal)
Military	160 GB		5,500 (internal)

TABLE VI
SUMMARY OF THE ATTACKS FOR THE CIC-IDS-2018 DATASET

Day	Attacks (duration)	Pattern
Day1	Brute force to FTP & SSH (90min each)	1-to-1
Day2	DoS GoldenEye & Slowloris (40min each)	1-to-1
Day3	Brute Force to FTP & DoS Hulk (60min + 35min)	1-to-1
Day4	DDoS LOIC-HTTP (60min)	N-to-1
Day5	DDoS LOIC-UDP & HOIC (30+60min)	N-to-1
Day6	Brute force Web/XSS & SQL inj. (60min+40min)	1-to-1
Day7	Brute force Web/XSS & SQL inj. (60min+70min)	1-to-1
Day8	Infiltration & port scan (70+60min)	1-to-1
Day9	Infiltration & port scan (60+90min)	1-to-1
Day10	Botnet (80+90min)	1-to-N

a) CIC-IDS-2018: This dataset was developed to provide data to analyse, test and evaluate NIDSs. To generate such a dataset, its authors developed a systematic approach in order to produce a diverse and comprehensive benchmark dataset. In their approach, they created user profiles with abstract representations of activity seen on typical networks. The benign behavior of each machine was generated using CIC-BenignGenerator [28], which is a tool to generate B-Profiles, i.e., realistic benign behaviors of a network. The tool uses machine learning and statistical analysis techniques to generate network events as if users in a typical network produced them. The network topology represents a typical medium company, with six subnets, deployed on the AWS computing platform.

This dataset includes seven different attack scenarios: Brute-force, Heartbleed, Botnet, DoS, DDoS, Web attacks, and infiltration of the network from inside. The ten days of normal activity and attacks performed are shown in Table VI. In the table, it is shown which attacks were conducted each day and what was the duration. In all days (except day 4) the attacks occurred in two distinct periods (one attack at a time). The rightmost column indicates the relation between the number of attackers and victims. The attacks were performed from one or more machines, using Kali Linux, in a specific network (within public IPs range) created only to attacker machines. Some of the tools used were *Patator* for brute force, *Ares* botnet, *Selenium* and *Heartleech* for web testing, *Hulk*, *GoldenEye*, *Slowloris*, *Slowhttptest* for DoS, and Low Orbit Ion Canon (LOIC) for DDoS.

b) Military dataset: The dataset of the military infrastructure was obtained from the Security Information and Event Management system (SIEM) [54] in production in that network, which collects netflow events from internal routers. These flows can give insights into misbehavior of internal hosts, undetected by deployed security systems. The dataset corresponds to a full month, with approximately 5,500 computers and 160 GB of size.

We emulated 4 attacks in that network to serve as ground truth when evaluating DYNIDS. The attacks were stealth dictionary attacks (against SSH and RDP) preceded by port scans

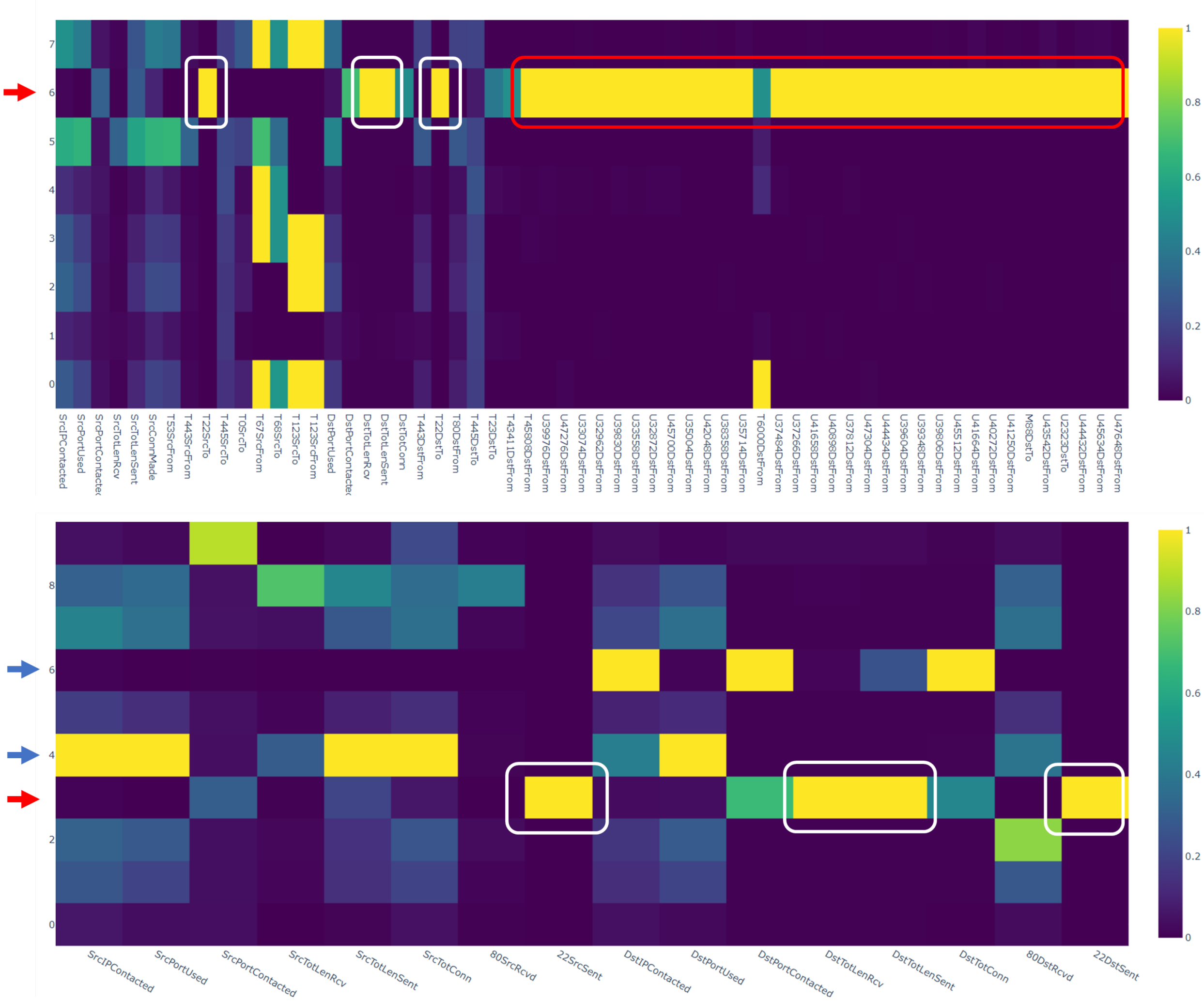

Fig. 3. HeatMap of DYN3_100 (top) and OutGene (bottom) approaches for SSH brute force attack of day1 (CIC-IDS-2018 dataset). Red and blue arrows on the left represent TP and FP, respectively. White surrounded features are equivalent between heatmaps. The red surrounded features are features corresponding to source ports used by the attacker.

(1-to-N and 1-to-1) at a slow pace (1 and 5-second interval). The main reasons for choosing these attacks were: (1) to have attacks that go unnoticed by traditional protection systems; (2) to capture internal reconnaissance activities (e.g., port scans) and slow dictionary attacks used by attackers with privileged information.

B. Increasing the number of features

This section shows the impact of increasing the number of features. For that purpose, we tested DYN2_x varying x from 10 to 100.

Recall that DYN2_x consists in dynamically defining port-based features by selecting the $x/2$ ports in more flows and the $x/2$ ports in less flows (Section IV). Also, notice that for each selected port, four different features are derived from the traffic analyzed in each time window, so we used at most 412 features (12 fixed, 400 for 100 ports).

We also compare with OutGene [14] and FlowHacker [13], both recent related work that only use fixed features and a single clustering algorithm, K-Means. Note that both OutGene

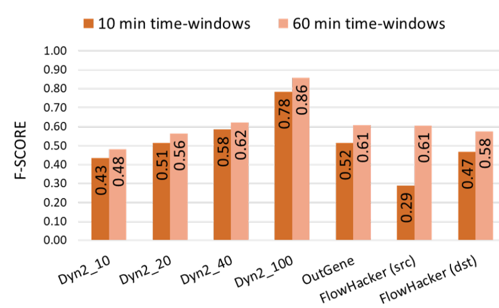

Fig. 4. Effect of increasing the number of features (w/CIC-IDS-2018 dataset).

and FlowHacker also consider IP addresses as the aggregation key. OutGene builds a single vector calculating features (count-based) depending on whether the IP address is source or destination. FlowHacker constructs two feature vectors (statistic and count-based) using IP addresses as a source or destination aggregation key and processes both keys independently, so we present results for both, which we denominate *FlowHacker* *(src)* and *FlowHacker (dst)*. We used Python to implement the feature extraction and clustering process of both approaches, using the same libraries we used to implement DYNIDS. We selected OutGene and Flowhacker because they are recent; we do not compare with more solutions as they would be older and no implementations are available.

In Figure 4, F-Score results are represented for all the attacks of the CIC-IDS-2018 dataset, only for the 10-minute and 60-minute time windows for lack of space. We can observe that by increasing the number of features, we get better F-Score values, e.g., 0.48, 0.56, 0.64, and 0.86 respectively for DYN2_10, DYN2_20, DYN2_40, and DYN2_100, with $\mathcal{B} = 60$min. The explanation for this is that with the increase in the number of features, we have more information to discriminate behaviors, namely anomalous behaviors. For example, excellent results are achieved in the detection of brute-force attacks.

To illustrate this example and to help understanding the meaning of outliers, we show heatmaps with the most relevant features, i.e., with the features with highest variance between clusters. Figure 3 shows two heatmaps (both DYN3_100 and OutGene) for the clustering for day one, when there was an SSH brute-force attack (see Table VI). Features are at the bottom (x-axis), clusters on the left (y-axis), the color represents the value of each feature for each cluster (the lighter, the higher). The comparison allows us to see how more features can reduce FPs and give more information about which attack is being performed. In this case, OutGene produces 1 TP along with 2 FPs, whereas DYN3_100 only produces the expected TP.

Something unexpected is that the approaches with less dynamic features (DYN2_10 and DYN2_20) performed worse than OutGene that uses a fixed set of features. What happens is that DYN2_x not always selected the features necessary to detect some of the attacks. OutGene, on the contrary, was configured with the necessary features by coincidence.

We also tested values of $x > 100$ but we found that the performance does not increase, as no further insight about attacks is gained by inspecting the activity on more ports.

C. Comparison of different approaches

This section compares the different approaches to dynamic feature extraction (see Section IV). Contrary to what could be intuitive at first, defining dynamic features solely based on the most used ports (i.e., TOP_x) is not the approach that guarantees the best results. This is observable on Figure 5, e.g., on the graph of the left, where TOP_100 has worse F-Score (0.72) than DYN3_100 (0.8/0.97 for $\mathcal{B} = 10$ and $\mathcal{B} = 60$min.) and DYN2_100 (0.78/0.86 for the same windows) that use exactly the same number of features (412). The main reason is that the least used ports are also important for detecting the type of attacks. Consider an example of network reconnaissance directed to well-known ports (i.e., 0 to 1023); features would be generated from these connection attempts, including less used ports with low traffic volume. A set of port-based features would be generated, where only the entity that made the port scan and the victim, the only with traffic in these ports, would have features with maximum values; all the other entities would have those same features at zero. Thus, we end up having a sparse matrix, where those features will only be relevant for attackers/victims related to network recognition. However, for all IPs that have those features at zero, the Euclidean distance to all the other entities does not change, making it irrelevant to have those features when comparing those entities with all the others (besides attacker and victim). Another example is the detection of unauthorized software, easily unveiled by features based on ports used by a few machines, regardless of traffic volume. However, it is not trivial to see that this type of feature allows the detection of attacks, such as brute force, that generate many requests sequentially with several different source ports (e.g., see features surrounded in red in Figure 3).

A second conclusion is that increasing the size of the analyzed time window, TOP_x performs worse. That is, in a larger time window, there is a broader set of used ports and the probability of the TOP_x approach select ports relative to the attack decreases. On the other hand, this factor is beneficial for the other two approaches because the less used ports end up being more easily highlighted. This can be seen in Figure 5 by observing the F-Score values for each time window in the different approaches, as previously mentioned.

In comparison, OutGene and FlowHacker performed worse than all the approaches that use dynamic port-based features in terms of F-Score, especially than Dyn3_100 that uses a little more than 400 features (Figure 5). A few results for FlowHacker aggregated by destination IP address provided very good results, but the approach is prone to higher values of FP.

In summary, both DYN2_x and DYN3_x (DYNIDS) achieve the best results, the latter being the best because that it aggregates three different types of port-based features, thus obtaining a better characterization of the data according to the factors mentioned above. DYN3_x can detect all the dataset attacks, except for the attack on day 10, when there are 10 victims infected with a botnet (Zeus and Ares). The non-detection is due to the classification method used – an outlier is an entity isolated in a cluster– does not allow detecting groups of victims with the same behaviors (10 in this case). This should be addressed by an analyst, doing manual inspection of small clusters. For this reason, this attack was not taken in account for the evaluation metrics.

D. Performance of cluster ensemble

The evaluation of the cluster ensemble is presented in Figure 6. As can be observed, the K-Means and Agglomerative algorithms have similar performances (F-Score on the left), whereas DBSCAN has the highest Recall (i.e., is the most sensitive) at the cost of generating the highest number of false positives (thus, the lowest Precision). The ensemble significantly improves the individual results of each algorithm, in particular those of DBSCAN. Notice also that Precision is what improves the most with the use of the ensemble, because there is a high reduction of the number of FPs.

Regarding the execution times, to calculate the total time for the whole process, we have to add the time needed for extracting the features from flow data, to the time needed for the clustering process. The highest values we obtained were around 1% of the time window size itself. Overall, the cluster ensemble

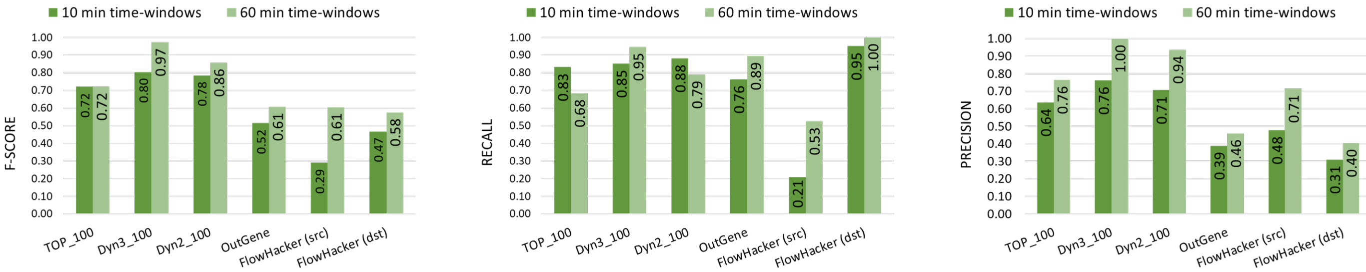

Fig. 5. Comparison of the performance DYNIDS (DYN3_100 in the graphs), other approaches, OutGene and FlowHacker.

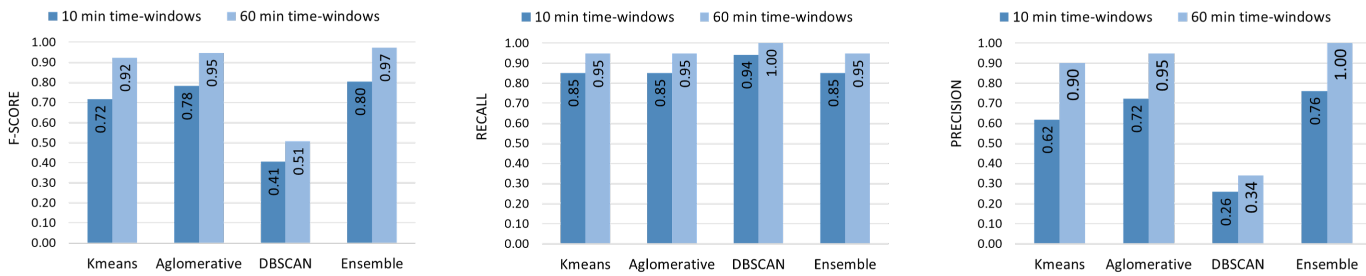

Fig. 6. Comparison of the overall performance of the different clustering algorithms and with the cluster ensemble (i.e., DYNIDS).

complexity is $O(n^2)$ due to the use of Agglomerative clustering algorithm. The cost to extract features grows linearly with the size of the input data, which depends on the size of the network, the duration of the time window analyzed and the volume of traffic and connections in that period. Considering we did not implement parallel processing and have used commodity hardware in the evaluation, we can say that the complexity allows a practical implementation in real-world scenarios and that our dynamic feature selection adds no significant delay to the analysis.

E. Evaluation with a real-world dataset

We made a less detailed evaluation using the DYN3_x approach and the military dataset, intending to show that DYNIDS works with real-world data. DYNIDS was able to reliably isolate both the attackers and the victims (in both cases internal hosts), leading to no FPs. The port scan, even at a very slow pace, generates port-based features based on the least used ports. This allowed the detection of the attack. Table VII summarizes the results for both days when the emulated attacks occurred. The first day includes a 1-to-1 slow port scan (5sec. pace) plus an SSH dictionary attack (2min. pace). The second day includes a 1–to-N slow port scan (1sec. pace) plus an RDP dictionary attack (30sec. pace).

TABLE VII
SUMMARY OF RESULTS WITH MILITARY NETWORK DATASET

Day	F-Score	Best window	Comments
day1	1	10min	no FP among 2332 entities
day2	1	10min	no FP among 2112 entities

We also processed the days of the dataset with regular traffic (i.e., with no attacks injected) and unexpectedly found some anomalies indicating misconfigured devices or unauthorized software, that we reported to the security operations team, which in turn provided excellent feedback on DYNIDS.

In summary, the alerts raised by DYNIDS corresponded to a real threat or anomaly. The fact that DYNIDS have not obtained false positives, as it did in the public dataset, has to do with the existence of more machines in the real-world data and, consequently, larger groups with the same pattern, allowing to isolate outliers in single clusters better. All in all, DYNIDS proved to be useful in a practical setting without significant effort to deploy since it just needs to be fed with netflow events.

Acknowledgements This research was supported by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID), by the Portuguese Army (CINAMIL), and by the European Commission under grant agreement number 830892 (SPARTA). We warmly thank prof. Ibéria Medeiros for feedback on a previous version of this work.

REFERENCES

[1] Check Point Research, “Check point 2020 cyber security report.” 2020.
[2] Fireye and Mandiant, “Special report, M-TRENDS 2020.” 2020.
[3] H. Debar, M. Dacier, and A. Wespi, “Towards a taxonomy of intrusion detection systems,” *Computer Networks*, vol. 31, pp. 805–822, Apr. 1999.
[4] G. Gu, R. Perdisci, J. Zhang, and W. Lee, “Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection,” in *17th USENIX Security Symposium*, pp. 139–154, 2008.
[5] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, “Nado: Network anomaly detection using outlier approach,” in *Proceedings of the International Conference on Communication, Computing & Security*, pp. 531–536, 2011.
[6] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, “An effective unsupervised network anomaly detection method,” in *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, pp. 533–539, 2012.
[7] T.-F. Yen, *Detecting stealthy malware using behavioral features in network traffic*. PhD thesis, Carnegie Mellon University Department of Electrical and Computer Engineering, 2011.

[8] P. Casas, J. Mazel, and P. Owezarski, "Unsupervised network intrusion detection systems: Detecting the unknown without knowledge," *Computer Communications*, vol. 35, no. 7, pp. 772–783, 2012.
[9] T.-F. Yen, A. Oprea, K. Onarlioglu, T. Leetham, W. Robertson, A. Juels, and E. Kirda, "Beehive: large-scale log analysis for detecting suspicious activity in enterprise networks," in *Proceedings of the 29th ACM Annual Computer Security Applications Conference*, 2013.
[10] M. Z. Rafique and J. Caballero, "Firma: Malware clustering and network signature generation with mixed network behaviors," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 144–163, Springer, 2013.
[11] D. Gonçalves, J. Bota, and M. Correia, "Big data analytics for detecting host misbehavior in large logs," in *Proceedings of the 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2015.
[12] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "A multi-step outlier-based anomaly detection approach to network-wide traffic," *Information Sciences*, vol. 348, pp. 243–271, 2016.
[13] L. Sacramento, I. Medeiros, J. Bota, and M. Correia, "Flowhacker: Detecting unknown network attacks in big traffic data using network flows," in *17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 567–572, 2018.
[14] L. Dias, H. Reia, R. Neves, and M. Correia, "Outgene: Detecting undefined network attacks with time stretching and genetic zooms," in *International Conference on Network and System Security*, pp. 199–220, Springer, 2019.
[15] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire, "Internet assigned numbers authority (iana) procedures for the management of the service name and transport protocol port number registry." IETF Request For Comments 6335, 2011.
[16] IANA, "Service name and transport protocol port number registry." Available at https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml, accessed: 2020-02-06.
[17] R. Bellman, *Adaptive Control Processes*. New Jersey, USA: Princeton University Press, 1961.
[18] M. Verleysen and D. François, "The curse of dimensionality in data mining and time series prediction," in *International Work-Conference on Artificial Neural Networks*, pp. 758–770, 2005.
[19] T. Ronan, Z. Qi, and K. M. Naegle, "Avoiding common pitfalls when clustering biological data," *Science signaling*, vol. 9, no. 432, pp. re6–re6, 2016.
[20] L. M. Weber and M. D. Robinson, "Comparison of clustering methods for high-dimensional single-cell flow and mass cytometry data," *Cytometry Part A*, vol. 89, no. 12, pp. 1084–1096, 2016.
[21] S. Adachi, "Rigid geometry solves "curse of dimensionality" effects in clustering methods: An application to omics data," *PloS one*, vol. 12, no. 6, 2017.
[22] B. Claise, "Cisco systems netflow services export version 9." IETF Request For Comments 3954, 2004.
[23] B. Claise, B. Trammell, and P. Aitken, "Specification of the IP flow information export (IPFIX) protocol for the exchange of flow information." IETF Request For Comments 7011, September 2013.
[24] M. Wang, B. Li, and Z. Li, "sFlow: Towards resource-efficient and agile service federation in service overlay networks," in *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*, pp. 628–635, 2004.
[25] M. Marchetti, F. Pierazzi, M. Colajanni, and A. Guido, "Analysis of high volumes of network traffic for advanced persistent threat detection," *Computer Networks*, vol. 109, pp. 127–141, 2016.
[26] "MITRE ATT&CK, network service scanning." Available at https://attack.mitre.org/techniques/T1046/, accessed: 2020-06-28.
[27] "MITRE ATT&CK, brute force." Available at https://attack.mitre.org/techniques/T1110/, accessed: 2020-06-27.
[28] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *4th International Conference on Information Systems Security and Privacy*, pp. 108–116, 2018.
[29] R. Zuech, T. M. Khoshgoftaar, and R. Wald, "Intrusion detection and big heterogeneous data: a survey," *Journal of Big Data*, pp. 90–107, 2015.
[30] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ch. 14, pp. 485–585. Springer, 2009.
[31] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network anomaly detection: Methods, systems and tools," *IEEE Communications Surveys Tutorials*, vol. 16, pp. 303–336, First Quarter 2014.
[32] A. Lazarevic, L. Ertoz, V. Kumar, A. Ozgur, and J. Srivastava, "A comparative study of anomaly detection schemes in network intrusion detection," in *Proceedings of the 2003 SIAM International Conference on Data Mining*, pp. 25–36, 2003.
[33] A. K. Jain, "Data clustering: 50 years beyond k-means," *Pattern recognition letters*, vol. 31, no. 8, pp. 651–666, 2010.
[34] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1, pp. 281–297, 1967.
[35] D. Steinley, "K-means clustering: a half-century synthesis," *British Journal of Mathematical and Statistical Psychology*, vol. 59, no. 1, pp. 1–34, 2006.
[36] F. Murtagh and P. Contreras, "Algorithms for hierarchical clustering: an overview," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 2, no. 1, pp. 86–97, 2012.
[37] D. Beeferman and A. Berger, "Agglomerative clustering of a search engine query log," in *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 407–416, 2000.
[38] C.-H. Lung and C. Zhou, "Using hierarchical agglomerative clustering in wireless sensor networks: An energy-efficient and flexible approach," *Ad Hoc Networks*, vol. 8, no. 3, pp. 328–344, 2010.
[39] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *KDD-96 Proceedings*, pp. 226–231, 1996.
[40] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander, "Optics: ordering points to identify the clustering structure," *ACM SIGMOD Record*, vol. 28, no. 2, pp. 49–60, 1999.
[41] J. Gan and Y. Tao, "DBSCAN revisited: Mis-claim, un-fixability, and approximation," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 519–530, 2015.
[42] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN revisited, revisited: why and how you should (still) use DBSCAN," *ACM Transactions on Database Systems*, vol. 42, no. 3, pp. 1–21, 2017.
[43] R. Xu and D. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 645–678, 2005.
[44] D. Xu and Y. Tian, "A comprehensive survey of clustering algorithms," *Annals of Data Science*, vol. 2, no. 2, pp. 165–193, 2015.
[45] "hdbscan clustering library: Benchmarking performance and scaling of python clustering algorithms." Available at https://hdbscan.readthedocs.io/en/latest/performance_and_scalability.html, accessed: 2020-08-20.
[46] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[47] "MITRE ATT&CK, uncommonly used port." Available at https://attack.mitre.org/techniques/T1065/, accessed: 2020-06-28.
[48] D. Whyte, P. C. Van Oorschot, and E. Kranakis, "Tracking darkports for network defense," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, pp. 161–171, IEEE, 2007.
[49] D. T. Pham, S. S. Dimov, and C. D. Nguyen, "Selection of K in K-means clustering," *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 219, no. 1, pp. 103–119, 2005.
[50] N. Rahmah and I. S. Sitanggang, "Determination of optimal epsilon (Eps) value on DBSCAN algorithm to clustering data on peatland hotspots in Sumatra," in *IOP Conference Series: Earth and Environmental Science*, vol. 31, 2016.
[51] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. CreateSpace, 2009.
[52] W. McKinney, "Data structures for statistical computing in Python," in *Proceedings of the 9th Python in Science Conference*, pp. 51–56, 2010.
[53] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.
[54] L. F. Dias and M. Correia, "Big data analytics for intrusion detection: an overview," in *Handbook of Research on Machine and Deep Learning Applications for Cyber Security*, pp. 292–316, IGI Global, 2020.