NetSlicer: Automated and Traffic-Pattern Based Application Clustering in Datacenters

Liron Schiff[1] Ofri Ziv[1] Manfred Jaeger[2] Stefan Schmid[3,2]
[1] GuardiCore Labs, Israel [2] Aalborg University, Denmark [3] University of Vienna, Austria

ABSTRACT

Companies often have very limited information about the applications running in their datacenter or public/private cloud environments. As this can harm efficiency, performance, and security, many network administrators work hard to manually assign actionable description to (virtual) machines.

This paper presents and evaluates *NetSlicer*, a machine-learning approach that enables an *automated* grouping of nodes into applications and their tiers. Our solution is based solely on the available network layer data which is used as part of a novel graph clustering algorithm, tailored toward the datacenter use case and accounting also for observed *port numbers*. For the sake of this paper, we also performed an extensive empirical measurement study, collecting actual workloads from different production datacenters (data to be released together with this paper). We find that our approach features a high accuracy.

CCS CONCEPTS

•**Networks** →*Network algorithms;*

KEYWORDS

Computer Networks, Machine Learning, Big Data

ACM Reference format:
Liron Schiff[1] Ofri Ziv[1] Manfred Jaeger[2] Stefan Schmid[3,2]
[1] GuardiCore Labs, Israel [2] Aalborg University, Denmark [3] University of Vienna, Austria. 2018. NetSlicer: Automated and Traffic-Pattern Based Application Clustering in Datacenters. In *Proceedings of Big-DAMA '18: Workshop on Big Data Analytics and Machine Learning for Data Communication Networks, Budapest, Hungary, August 20, 2018 (Big-DAMA '18),* 6 pages.
DOI: 10.1145/3229607.3229614

1 INTRODUCTION

Network administrators of datacenters are often faced with the challenge of *poor workload and application visibility.* Their datacenters may host thousands of applications, many of which the administrators have never heard of. Frequently, it is not even known whether two (virtual) machines belong to the same application or tier.

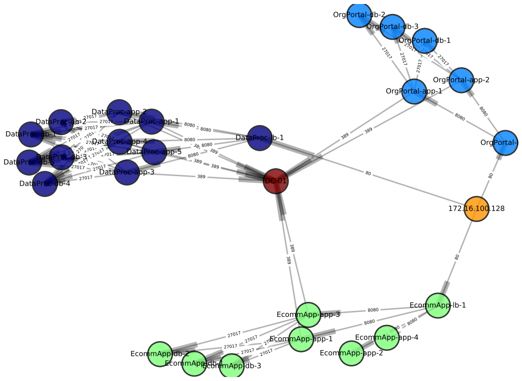

Figure 1: Example of three applications (*dark blue, light blue* and *green*), each of which consists of three tiers: load-balancer, servers implementing application and database servers (in this case *MongoDB*). The domain controller communicates to most app tier servers. The IP address *172.16.100.128* (orange) indicates where client requests enter the datacenter.

An improved *application-awareness* however could benefit and simplify many tasks, e.g., concerning *resource management* (e.g., to reduce bandwidth consumption, to improve network latency, to release unused resources, etc.), *regulations* (to comply with regulations, it can be necessary to encrypt communication channels over "unsafe" segments or to store data in a specific territory), *security* (detect *anomalous* communication patterns and applications), etc. Application-awareness also allows to enforce "smarter" policies (allowing to define tighter policy rules without the risk of, e.g., harming essential connectivity), detect unwanted/forbidden/bad connections or links, as well as remote (malicious) applications.

Today, identifying applications is mostly a manual effort on a per machine basis, for example it may consists of live inspection of machine activity (e.g., processes and network connections) and offline analysis of configurations and the non-standardized procedures led for machine deployment. Moreover, interaction with the many owners or developers of the different applications deployed in the network, is required as well.

This paper explores opportunities for a *more informed* and *automated* approach to inferring cloud applications (i.e., which nodes belong to the same application), leveraging *network layer data* as part of (unsupervised) *machine learning*, and hence aiming to reduce or completely overcome the required manual work.

Big-DAMA '18, Budapest, Hungary

978-1-4503-5904-7/18/08…$15.00
DOI: 10.1145/3229607.3229614

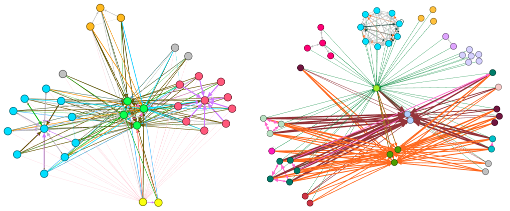

Figure 2: Example traffic patterns between tiers and applications. *Left:* An excerpt from a Windows-based datacenter, *Right:* From a Linux datacenter. Node colors indicate applications, link colors server ports.

The Challenge. Figure 1 depicts the communication pattern of three applications (in green, blue, dark blue) observed in a datacenter. Here, a high-degree (Windows) domain controller (in red) which communicates with most servers; client requests arrive at the load-balancer (in orange) when entering the datacenter. Each of the three applications consist of three tiers: a tier for load-balancing, a tier for the servers implementing the application, and a tier of databases.

The problem of application inference boils down to partitioning the communication graph into applications and their constituting tiers. This is challenging, for several reasons. To start with, while clustering and "community detection" algorithms have been studied intensively in the literature, interactions and communications of cloud applications can be very diverse and different from those typically observed in biological and human social networks. For example, while it seems reasonable to assume that components of the same application tier communicate more frequently among themselves than with other components, this may not always be the case. Some applications, e.g., related to infrastructure monitoring, anti-virus, intrusion detection, a Splunk server, Zookeeper, Kafka, or a Windows domain controller, usually serve multiple other applications and hence feature a large number of incoming/outgoing edges (i.e., have high in- and out-degrees); other application structures may emerge from small batch processing applications whose servers may connect only to a small subset of other servers; servers of applications related to databases and message queues are usually highly connected among themselves (high internal density); etc.

The traffic patterns also depend on the cloud operating system and type of the datacenter. For example, some datacenters may be *auto-deployed* and automate the generation of VMs (e.g., based on OpenStack and using *Cheff*, *Puppet*, etc.), others may serve as *backoffice datacenters* which run mail servers, *Active Directories (ADs)* mananged by the domain controller, *dns*, etc., or as *multi-environment datacenters* that run production, development and testing environments together. Depending on the applications that the datacenter hosts and the protocols the applications use, traffic patterns will look different: applications such as *Splunk*, *Active Directory* and its protocols like *LDAP*, or Linux SSH connections from *jumpboxes* are star-shaped and, e.g., have very high-degree nodes to which many other low-degree nodes connect (we will say that the corresponding server ports are *noisy*, e.g., have thousands of connections). Other protocols, e.g., used for file shares, like *SMB* used in a backoffice datacenter, are *peer-to-peer* and hence are rather symmetric. Figure 2 depicts two examples of communication patterns observed in our datacenters.

Another challenge but also an opportunity is introduced by the fact that traffic patterns usually contain more information than mere connectivity. In particular, the observed packet headers include *port numbers*. While we do not want to make any assumptions on the meaning of these port numbers, we want to consider them as part of our machine learning approach. In other words, our goal is to infer and cluster applications by leveraging not only topological data *but also metadata*.

Our Contributions. This paper presents *NetSlicer*, a machine-learning approach to support the automated, accurate and efficient application inference (i.e., which nodes belong to the same application), relying solely on the available network layer data. After gathering machine interactions from network traces and modeling it as an annotated graph (a datacenter map), we employ customized unsupervised machine learning techniques to cluster similar machines into groups, based on communication patterns expressed by the graph. At the heart of our approach lies a novel graph clustering algorithm that is designed specifically for reasoning about datacenter workloads, using traffic pattern based application inference. It accounts for the *topological features* of the constructed communication graph but also the *annotated port numbers*.

To evaluate *NetSlicer*, we conducted extensive measurements in multiple and diverse private and production datacenters, leveraging a large-scale deployment of our monitoring infrastructure. Parts of the data collected for this study will be released together with this paper. Our evaluation, which also includes a case study on network segmentation, shows that the accuracy of our algorithm is high, also compared to more sophisticated but "out-of-the-box" clustering algorithms which are not tailored towards our use case. Our algorithm is also attractive in that it does not rely on any pre-processing or learning phase.

Finally, note that while our work tackles the problem of grouping nodes into applications, and not to find the application *type*, this is a first step towards the latter goal.

Related Work and Novelty. The problem of automatic inference of communities resp. computation of clusters is an evergreen topic which has received much attention over the last decades. Our work specifically revolves around communication networks and topological data which is enriched with meta-data (namely network ports), so it is related to works on IP-to-IP network clustering [6] (without considering ports however) and the clustering of annotated graphs (e.g., [9], however, without considering communication networks). Our model can also be seen as a classification problem and is related to literature on (hierarchical) classification [11]. We are also not the first to observe the great potential of machine learning and big data analytics in networking, which is currently a very active field, see e.g., [1, 2, 7, 12, 14, 15], to just name a few.

However, to the best of our knowledge, we are the first to propose and use fine-grained traffic monitoring in datacenters to identify cloud applications and their tiers. Our approach is tailored to the specific case study and workload of production datacenters, accounting for node interactions as well as ports. Our first empirical insights reveal that generic state-of-the-art techniques (e.g., stochastic block models [4] ignoring meta information) do not perform well for our problem.

2 NETSLICER

We present *NetSlicer* in two stages. This section first presents our monitoring approach. Subsequently, we describe our algorithm in details.

2.1 Traffic Monitoring System

Deploying a monitoring system in a large production datacenter is a non-trivial task and an interesting topic in itself. Since it is not the main focus of our work, in the following, we will only discuss some basic aspects. Generally, our algorithm can work with data obtained by any kind of monitoring system capable of providing the basic metadata we require (e.g., using AWS Flow Logs). Our monitoring approach intercepts all traffic between the datacenter VMs themselves as well as from the datacenter VMs to the external hosts, for example between the PCs of the organization's employees and Internet servers. To achieve this, we deploy packet collectors either at the VMs or at core network locations, such as hypervisor servers and switch taps. All collected data needs to be correlated to match duplicated flow reports.

That said, for our approach to work, only a subset of the information is required. In particular, the *destination* port of the TCP connection setup SYN packet is sufficient (as the connection is initiated by the client, we will sometimes call the source node the *client* and the destination node the *server*). We emphasize that we currently do not collect any information about the frequency of interactions between two nodes nor about the volume or rate of communication (but we may do so in the future). Nevertheless, as we will see, an accurate application inference is possible. Also, since we only collect the destination port, we only store one port per (directed) connection, and will sometimes annotate the corresponding edge by the (destination) port number.

2.2 Key Concepts: Similarity and Variance

Our approach to identify application tiers relies on two key concepts: *similarity* and *variance*. As these concepts are of independent interest, we present them upfront, in a dedicated section. Intuitively, a tier combines "similar" VMs. Similarity is characterized by two main aspects:

(1) *Similar neighbors:* If two VMs are connected to similar sets of other VMs, then they likely belong to the same tier.
(2) *Direction and ports:* In addition to the topological information and mere connectivity between two VMs, also the *direction* of connection is relevant: i.e., from client to server, as well as the number of the *destination port*.

In the following, we will use the term *nodes* to describe the communicating VMs. The set of such nodes is denoted by V and the set of *port numbers* is denoted by P.

As both the similarity of neighboring nodes as well as the port is relevant, we define the set of connections between nodes as a set of annotated and directed edges $E \subseteq V \times V \times P$. We also introduce the notion of endpoint. An endpoint is an element of $\Phi = V \times P \times R$, where $R = \textcircled{c}, \textcircled{s}$ represents the role of a node as either client or server. An edge $e = (u, v, p)$ defines the two endpoints $(u, p, \textcircled{c})$ and $(v, p, \textcircled{s})$.

We define a feature vector W_v characterizing each node v using the remote endpoints incident to it. W_v contains a non-negative value for each endpoint in Φ, i.e., $W_v : \Phi \rightarrow \mathbb{R}_{\geq 0}$. W_v is non-zero only for endpoints (either client or server) incident to v. For example, the endpoint $\phi = (x, 21, \textcircled{c}) \in \Phi$ is non-zero in W_v if $e = (x, v, 21)$ is an edge in E.

Based on their feature vectors W_u and W_v, we can compute the similarity of the two nodes u and v as the *normalized scalar product*:

$$Sim(u, v) = \sum_{\phi \in \Phi} W_u(\phi) W_v(\phi) / (|W_u| \cdot |W_v|),$$

where $|W_x| = \sqrt{\sum_{\phi \in \Phi} W_x(\phi)^2}$.

In a simple setting of binary W_v values, and $|W_v|$ as summation, the similarity measure expresses the ratio of shared endpoints; however while some endpoints provide valuable insights to the application clustering algorithm, others are of limited use, e.g., since almost every node connects to them, uniformly. Indeed, it turns out that binary W_v values are not effective in identifying applications and their tiers.

Accordingly, we introduce the notion of *variance* per endpoint: the variance $Var(\phi)$ of endpoint $\phi = (v, p, r)$ is defined as the *maximum "distance" of any two nodes connected to* ϕ, where the distance in turn depends on the similarity. Intuitively, the less similar the neighbors of an endpoint the more "noisy" and the less useful it is in turn to compute similarity. Succinctly:

$$Var(v, p, r) = \max_{x, y} (1 - Sim(x, y))$$

where x and y are two nodes connected to v at a port of number p and in the role inverse to r.

To account for endpoint variance, we define the weight of an endpoint (x, p, r) for node v as: $W_v(x, p, r) = (1 - Var(x, p, r))^{\alpha}$. Here, α is a small constant (between 2 and 6).

2.3 Traffic-Based Clustering

Given the available network layer data, we now present our algorithm to reliably identify tiers and applications. Our method is based on gathering machine interactions from network traces and modeling it as an annotated graph. We then use our customized unsupervised machine learning technique to cluster similar machines into groups, based on communication patterns expressed by the graph. The algorithm combines three ideas:

(1) **Iterative merging to find tiers:** Our algorithm revolves around similarity and variance of nodes, as introduced above. During its execution, our algorithm will *iteratively merge* similar nodes into bigger ones, eventually forming a cluster representing a tier.
(2) **Similarity/variance fixpoint:** Our approach is based on an iterative computation of similarity and variance, converging toward a fixpoint. When nodes express tiers their outgoing links are connected to other types of tiers or apps and therefore has high variance preventing farther merges.
(3) **Clustering tier graph to find applications:** The result of the iterative merging process described above is a *tier graph*: the nodes in this graph are tiers and the connections denote that the two tiers communicate but are not sufficiently similar to be merged. Based on this tier graph, we then weigh edges and apply a weight based graph clustering algorithm, to merge tiers into applications.

With these intuitions in mind, we now describe our algorithm in more details. Since our algorithm iteratively merges similar machines into larger and larger clusters, to eventually form tiers, also the (cluster) graphs over time evolve: $G_0, G_1, G_2, \ldots, G_t, G_{t+1}, \ldots$. For ease of presentation, we will use the term *node* for both individual (virtual) machines and to sets of (virtual) machines: the set of such nodes at time t is denoted by $V_t = \{v_1, v_2, \ldots, v_{n(t)}\}$. The *size* of a node v_i is defined as the number of individual machines it represents; similarly, the number of edges of a machine v_i is defined by the sum of the edges of its individual machines.

For nodes representing multiple machines, we consider the union of the edges of the machines contained in this node. The edge is labeled with the port p. Note, there can be multiple parallel edges between two nodes (but only one per port and direction).

Subsequently, we want to find the applications. Applications are harder to cluster, and require connectivity among their constituting tiers. Accordingly, we consider one round of clustering of the graph of tiers: intuitively, we merge two tiers into an application if they are highly connected, considering edge weights that can be based on the nodes and edges properties as computed during the previous phase.

In summary, Algorithm 1 relies on the following functions:

- *ComputeSim*(G_t, Var): Computes similarity of nodes in current cluster graph G_t, based on the pairwise similarity measure introduced above (which in term depends on the variance).
- *ComputeVar*(G_t, Sim): The variance of an endpoint is inversely related to the maximal similarity between all nodes connected to this specific endpoint. See above for details.
- *FindSimilar*(G_t, Sim): We use a simple criterion whether similarity is above a threshold. Any two nodes that are similar enough (i.e., above threshold 0.6) are connected and we return the resulting connected components as groups to be merged.
- *Merge*(G_t, *groups*, Var): We replace each group of nodes with a new node with merged endpoints. In order to improve the next similarity calculation we also set each new (merged) endpoint variance as the maximum for all old endpoints merged to it.

In the pseudo code, we describe a simplified approach where we merge tiers to apps based on predefined threshold (θ_2) on link weights. In practice, we use small threshold values (e.g., $0.01 - 0.1$ depending on weight function) to filter out insignificant edges and also included a *hubs filtering* mechanism, that prevents highly connected nodes (in terms of number of neighbors and/or port numbers) to merge many tiers. Moreover, in the experiments, we will also evaluate the benefits of using other state-of-the-art graph clustering algorithms on the tiers graph (in particular *node2vec* [5] and *louvain modularity* [3]). We use two weighing techniques for merging applications:

(1) **Edge intensity:** The weight of edge $e = (u, v, p)$ is proportional to the logarithm of the link size divided by the logarithm of node sizes, i.e., $w_{EI}(e) = \frac{\log e.count}{\log u.count \cdot v.count}$, where node (edge) count denotes how many nodes (edges) were merged to the node (edge) during the tier merge phase.

(2) **Edge endpoints variance:** The weight of edge $e = (u, v, p)$ is proportional to a sigmoid function applied to the sum of variances of the two endpoints of the edge, i.e., $w_V(e) = 1 - f(var(u, p, \textcircled{c}) + var(v, p, \textcircled{s}))$ where f is a high slope sigmoid function, e.g., $f(x) = \tanh 10x$.

Algorithm 1 Application Inference

```
Require: labeled connection graph $G = (V, E, L)$
Ensure: Set of applications $S = \{a_1, \ldots, a_k\}$
(* Phase 1: Infer Tiers *)
initialize $G_0 = G$, $Var = 0$, $t = 0$, $groups = \{\}$
while $t = 0$ or $|groups| > 0$ do
    $Sim \leftarrow ComputeSim(G_t, Var)$
    $Var \leftarrow ComputeVar(G_t, Sim)$
    $Sim \leftarrow ComputeSim(G_t, Var)$
    $groups \leftarrow FindSimilar(G_t, Sim)$
    $G_{t+1}, Var \leftarrow Merge(G_t, groups, Var)$
    $t \leftarrow t + 1$
end while
(* Phase 2: Infer Applications *)
$W \leftarrow ComputeEdgeWeights(G_t, Var)$
initialize $S = \{\{v\} | v \in V_t\}$,
for all $u, v \in E_t$ do
    if edge weight between $u$ and $v$ higher than $\theta_2$: then
        merge the sets of $u$ and $v$ in $S$
    end if
end for
return $S$
```

We discovered that our merging technique worked well when used with a combination (multiplication) of the two functions, and other algorithms worked well when used with edge intensity.

3 EVALUATION

To evaluate our approach, we conducted a large scale measurement study across different private and production datacenters. Moreover, for the sake of this study, we implemented alternative algorithms "out-of-the-box" for clustering communication patterns. In the following, we first present our methodology in more details. Subsequently, we report on the main insights of our measurement study.

3.1 Methodology and Implementation

We collected large amounts of temporal communication patterns in private and production datacenters. Some datacenters are mainly based on Windows machines and others mainly on Linux. In order to obtain a ground truth, we inferred the applications for these data sets manually, together with an analyst. The following table (Table 1) summarizes the network traces of the three datacenters used in our evaluation in the following.

When considering the application sizes in each network (see Figure 3), we observe that net1 has lower ratio of standalone VMs and higher ratio of big apps compared to net2 and net3.

The agents in our experiments were restricted to collect data on TCP connection setup: our data comes in the form of the TCP connection establishment and closedown control packets SYN and

	net1	net2	net3
dominant OS	linux	windows	windows
# nodes	**3003**	**2507**	**4588**
# monitored	153	143	174
# unmonitored	2850	2364	3414
# ground-truth apps	**40**	**58**	**85**

Table 1: Datacenter data characteristics

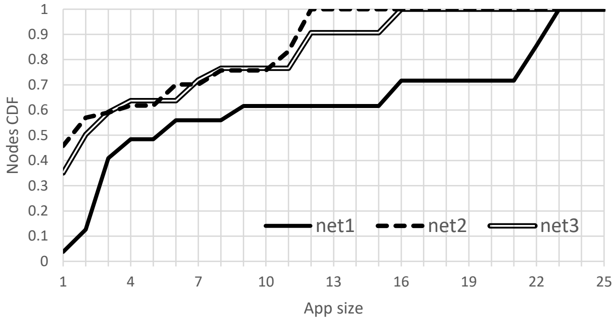

Figure 3: Distribution of network nodes according to application sizes (in the ground truth).

FIN. In particular, we know the source and destination IP addresses and established ports by these packets. As an additional challenge, the agents collecting this data were not installed on all customer virtual machines.

We compare the NetSlicer algorithm against node2vec [5] and louvain modularity [3]. For [5], we employ Agglomerative Hierarchical Clustering over the vectors, choosing the best number of clusters. Moreover, we tested two algorithm variations of running our algorithm for the tiers clustering phase, namely *NS+N2V* and *NS+MOD*. For *NS+N2V* we used HDBSCAN [8] (which does not require to set the number of clusters), to cluster (tier) nodes to apps based on node2vec vectors.

For each of the networks, we compared the results of the algorithms with a solution prepared by analysts assisted by information obtain from the network operators. We used Adjusted Random Index (ARI) [13] as a measure of clustering distance of the nodes to (ground-truth) applications. For each clustering of nodes $C : V \longrightarrow N$, as we motivate in Section 4, we also considered the projected clustering of the edges, $C^{‘} : E \longrightarrow N \times N$, where an edge from v to u is clustered with id (C_v, C_u), where C_x is the cluster id assigned to node x.

In terms of implementation, NetSlicer is written in python from scratch, using the following packages: networkx for some of the graph operations, numpy and scipy for math operation and scikit-learn[10] and HDBSCAN for ML related tools. The heavy parts of the algorithm, the similarity and variance computations, are implemented to run in parallel in different processes.

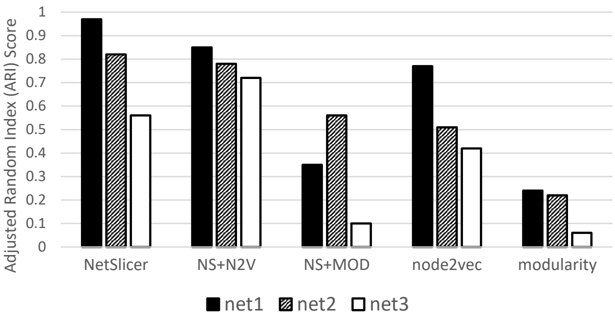

Figure 4: Comparison of the VMs clustering scores for the different algorithms and datacenters. Scores computed by the ARI distance of the alg. result from the ground truth.

3.2 Results

Our main results are summarized in Figures 4 and 5. Overall NetSlicer performed the best compared to other algorithms tested, except for the third network, which was clustered better by NS+N2V: after running NetSlicer for the net3 tiers, it was better to run node2vec to cluster the tiers to apps instead of of the simplified merges performed by NetSlicer. In both cases, running NetSlicer even in the first phase, was better than running just node2vec.

As can be seen in Figure 4, node2vec achieves decent VM clustering results, but as can be seen in Figure 5, when projected on links, the scores are very low. This may indicate that node2vec clusters well nodes with low connectivity (degree), but not highly connected ones. Of all algorithms tested, modularity performed the worst. Thus, apparently, datacenter applications do not follow community behavior, as approximated by modularity.

Another observation we have is that most algorithms performed better on net1 than on net2 and net3. This can be attributed to the fact that net1 is dominated by linux servers which are less noisy compared to windows servers that dominate net2 and net3. Moreover, as expressed in Figure 3, almost half of the VMs in net2 and net3 are standalone application which make them more prone to clustering errors: most clustering techniques usually aim for cluster sizes larger than one.

4 CASE STUDY: SEGMENTATION

To further evaluate our approach, we also conducted a small case study on network segmentation. Security teams are always looking to find ways to minimize the attack surface, and limit threats from propagating within datacenters. Best security practices for creating policies are along the following lines: starting with all the given nodes in the network, a first high-level grouping (subnet, environment etc.) is performed. Subsequently, the security experts aim to *label* applications, i.e., identify which application is associated with each machine. Based on these groups and applications, rules are defined, i.e., one can then define sets of nodes or groups that should not communicate to each other.

Our interviews with analysts confirm that *given* a labelling, defining policies is often straight-forward: given the labels, it is sometimes even possible to generate policies (i.e., rules) *automatically*.

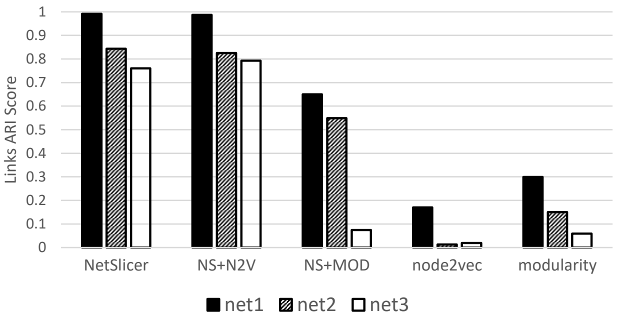

Figure 5: Comparison of the link clustering scores for the different algorithms and data centers. Score computed by the ARI distance of the alg. result (projected on links) from the ground truth.

However, computing the labels and classifying servers into applications can be very challenging, requiring the security team to meet and interact with the many owners or developers of the different applications deployed in the network.

Considering NetSlicer clustering results (Figure 4), we find that NetSlicer can produce labels that are competitive with a security analyst with domain knowledge. Analysts, when presented with our results, said that they require only a short verification and fix time, compared to the tedious work of labeling manually from scratch. Moreover if we consider the quality of the deny/allow rules between nodes of different apps (labels), it is more interesting to look on the correctness of labeling of pairs of VMs that communicate (instead of the labeling of each VM). *NetSlicer* achieves this with high accuracy, as shown in Figure 5.

5 CONCLUSION

We presented *NetSlicer*, a fine-grained approach to monitoring traffic patterns in datacenters and showed that it can facilitate the automated inference of which nodes belong to the same applications and tiers, which in turn can be used, e.g., for application type inference or network segmentation. We understand our work as a first step, and believe that it opens an interesting new perspective on the well-studied clustering problem occurring in many other contexts, including social networks and complex networks, by focusing on communication patterns occurring in cloud applications.

In our future work, we plan to compare our solution to additional approaches, such as the Ward method (which however requires the specification of the number of clusters) and stochastic block models (which however is a purely topological approach).

To facilitate future research and as a contribution to the research community, we will soon release parts of our data sets (anonymized) at https://www.guardicore.com/labs/datacenter-traces/. Please refer to the current paper when using this data set for your own research.

Acknowledgments. The authors would like to thank Avishag Daniely and Lior Neudorfer from GuardiCore for taking time for going through our questionnaires and providing useful inputs during the conducted interviews. We would also like to thank Kensuke Fukuda and the anonymous reviewers.

REFERENCES

[1] William Aiello, Charles Kalmanek, Patrick McDaniel, Subhabrata Sen, Oliver Spatscheck, and Jacobus Van der Merwe. 2005. Analysis of communities of interest in data networks. In *International Workshop on Passive and Active Network Measurement*. Springer, 83–96.

[2] Andreas Blenk, Patrick Kalmbach, Stefan Schmid, and Wolfgang Kellerer. 2017. o'zapft is: Tap Your Network Algorithm's Big Data!. In *Proc. SIGCOMM Big-DAMA*.

[3] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008 (2008).

[4] Peter Chin, Anup Rao, and Van Vu. 2015. Stochastic block model and community detection in sparse graphs: A spectral algorithm with optimal rate of recovery. In *Conference on Learning Theory*. 391–423.

[5] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *Proc. 22nd ACM SIGKDD*. 855–864.

[6] Marios Iliofotou, Brian Gallagher, Tina Eliassi-Rad, Guowu Xie, and Michalis Faloutsos. 2010. Profiling-By-Association: a resilient traffic profiling solution for the internet backbone. In *Proc. of the 6th International COnference*. ACM, 2.

[7] Anestis Karasaridis, Brian Rexroad, and David Hoeflin. 2007. Wide-scale Botnet Detection and Characterization. In *Proc. HotBots*.

[8] Leland McInnes, John Healy, and Steve Astels. 2017. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software* 2, 11 (2017), 205.

[9] M. E. J. Newman and A. Clauset. 2016. Structure and inference in annotated networks. *Nature Communications* 7 (June 2016), 11863. DOI: http://dx.doi.org/10.1038/ncomms11863 arXiv:1507.04001

[10] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[11] Tiago P. Peixoto. 2014. Hierarchical block structures and high-resolution model selection in large networks. *Physical Review X* 4, 1 (2014), 011047. http://journals.aps.org/prx/abstract/10.1103/PhysRevX.4.011047

[12] R. Gonzalez et al. 2017. Net2Vec: Deep learning for the network. In *Proc. SIGCOMM Big-DAMA*.

[13] Douglas Steinley. 2004. Properties of the Hubert-Arable Adjusted Rand Index. *Psychological methods* 9, 3 (2004), 386.

[14] Juan Vanerio and Pedro Casas. 2017. Ensemble-learning approaches for network security and anomaly detection. In *Proc. SIGCOMM Big-DAMA*.

[15] K. Xu, F. Wang, and L. Gu. 2011. Network-aware behavior clustering of Internet end hosts. In *Proc. IEEE INFOCOM*. 2078–2086.