This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

AWESoME: Big Data for Automatic Web Service Management in SDN

Martino Trevisan† Idilio Drago† Marco Mellia† Han Hee Song⋆ Mario Baldi†⋆
†Politecnico di Torino, e-mail: {first.last}@polito.it
⋆Cisco, Inc., e-mail: {mariobal,hanhsong}@cisco.com

***Abstract*—Software Defined Network (SDN) has enabled consistent and programmable management in computer networks. However, the explosion of cloud services and Content Delivery Networks (CDN) – coupled with the momentum of encryption – challenges the simple per-flow management and calls for a more comprehensive approach for managing web traffic. We propose a new approach based on a "per service" management concept, which allows to identify and prioritize all traffic of important web services, while segregating others, even if they are running on the same cloud platform, or served by the same CDN.**

We design and evaluate AWESoME, Automatic WEb Service Manager, a novel SDN application to address the above problem. On the one hand, it leverages big data algorithms to automatically build models describing the traffic of thousands of web services. On the other hand, it uses the models to install rules in SDN switches to steer all flows related to the originating services.

Using traffic traces from volunteers and operational networks, we provide extensive experimental results to show that AWESoME associates flows to the corresponding web service in real-time and with high accuracy. AWESoME introduces a negligible load on the SDN controller and installs a limited number of rules on switches, hence scaling well in realistic deployments. Finally, for easy reproducibility, we release ground truth traces and scripts implementing AWESoME core components.

I. Introduction

The Software Defined Network (SDN) paradigm has changed the way networks are managed [1]. Thanks to a logical centralized controller and well-defined interfaces to program forwarding devices, SDN controls the traffic in a consistent manner and dramatically eases interoperability across different vendors. Yet, network managers face complex traffic engineering and policing requirements when operating the network to meet quality levels, prioritize traffic and enforce polices. Traditionally, such requirements might translate into complex matching on packets or flows, e.g., to drop P2P packets or regulate flows related to specific services.

The complexity of the web has introduced more challenges in the overall picture. On the one hand, the widespread adoption of cloud services and CDNs puts into question the identification of the services behind the traffic flows because a single server supports multiple services, e.g., providing content for several sites. On the other hand, the convergence towards encrypted protocols – i.e., HTTP(S) [2] – has rendered Deep Packet Inspection (DPI) based matching ineffective. Nowadays, the access to a single service might result in the generation of several traffic flows to multiple servers, e.g., CDN nodes, advertising platforms, video servers, etc., that are shared by different services and, as such, cannot be easily associated to the specific web service originating the traffic.

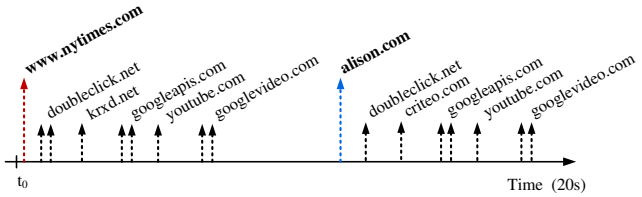


Fig. 1: Flows opened when visiting two websites. We search flexible mechanisms to independently manage all traffic triggered by each site – e.g., for traffic engineering and policing.

Figure 1 illustrates this problem by showing the diverse servers contacted by a user after visiting two simple web sites, i.e., an e-learning platform and a news website. Arrows mark flows to the contacted domains of first- and third-party platforms involved in the services. Both sites rely on the same third parties for video services (i.e., YouTube), analytics and web tracking. This poses unique challenges to a network manager wanting to give higher priority to the e-learning platform (on the right), while segregating the news site traffic (on the left). Prioritizing only the first-party servers would fail to give the intended treatment for the video content of the e-learning platform that is hosted on YouTube, whereas prioritizing *all* YouTube traffic would give high-priority also to leisure videos triggered by the news site.

Our goal is to allow administrators to manage *all* traffic of a service comprehensively, i.e., steering all traffic generated by the user accessing a given service, and not just the traffic related to first-party servers. A novel approach to traffic management is required where policies are based on the *services* that users are contacting, which in turn must be translated into rules that can be imposed on *packets* and *flows*.

We solve this problem by proposing AWESoME. It defines a novel paradigm in which the network administrator imposes policies based on the *service* being accessed, e.g., giving priority to alison.com in Figure 1, while segregating nytimes.com, and treating third-party traffic according to the accessed first-party service. Using big data approaches, AWESoME automatically learns *groups* of flows related to the services and steers them despite being served by the same CDNs, servers, clouds, and with the same (encrypted) protocols.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

AWESoME is a SDN application that leverages standard SDN functionalities to steer traffic in the network. At the core of the SDN application is a novel annotation-module operating at edge elements, which is able to associate each flow to the originating service in real-time and with high accuracy. It leverages DNS information and big data to automatically learn from the traffic. It achieves an overall accuracy higher than 90%, that, despite not suitable for security purposes, is well-suited for traffic engineering and management goals.

In contrast to previous works that also aimed at bringing service-awareness to SDN, but focused on per-flow management [3], [4], [5], [6], AWESoME addresses the challenge in the more comprehensive and seamless way based on the following three premises:

• **Comprehensive policing of services**: AWESoME creates forwarding rules that cover complex relations among flows (e.g., as in Figure 1). It achieves that by learning which domains are typically contacted when accessing each service. Models to translate high-level descriptions of services into low-level rules are learned automatically from traffic with unsupervised algorithms, minimizing human intervention. Flow dependencies have already been studied and exploited for data-center management [7], [8], but we extend those methodologies to operate at the edge of the network.

• **Early classification with low overhead**: AWESoME takes final forwarding decisions since the very first packet of each flow. This limits the load on the controller and application, making it compatible with actual technology. This is achieved by extending methodologies that rely on the DNS for traffic annotation [9], [10], [11], [12].

• **Compliance with SDN specifications**: AWESoME has been designed to be fully compliant with the basic SDN architecture and the latest version of OpenFlow [13], although it could also be deployed with other communication protocols between controller and forwarding elements. It requires no changes to existing APIs and SDN controllers, hence allowing adoption of AWESoME to existing SDN platforms to be simple.

We thoroughly evaluate accuracy and scalability of AWESoME in the classification and steering of web service traffic using traces collected from both volunteers (which offer us ground truth) and operational networks (which challenge AWESoME in both ISP and corporate environments). Results show that AWESoME (i) identifies traffic per service with accuracy greater than 90%, more than adequate for traffic management; (ii) limits decision time to less than a hundred microseconds, with negligible load overhead to SDN controllers; (iii) adds a compatible number of rules to forwarding devices and, therefore, it is feasible for real deployments.

To allow other researchers to reproduce and validate our results, we release to the public ground truth traces and Python scripts implementing the core components of AWESoME.[1]

Next we introduce terminology, deployment scenarios, and AWESoME architecture (Section II). We then detail the core annotation algorithms (Section III), before introducing the dataset (Section IV) that we use to validate performance (Section V). We conclude by discussing related work (Section VI) and summarizing our findings (Section VII).

[1]Available at: https://bigdata.polito.it/content/open-datasets

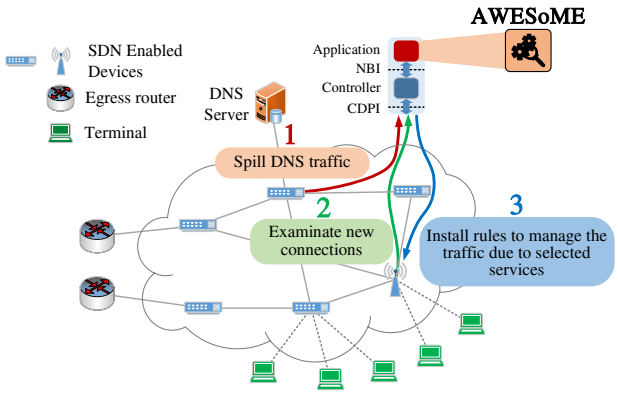


Fig. 2: Typical corporate SDN deployment.

II. Definitions and Architecture

A. Per service management approach

We aim at enabling management operations that target the control of all traffic involved in the access to web services, i.e, all objects a browser or a terminal downloads when users access the given web service. We call this *per service* management. We envision several scenarios where the *per service* approach will help administrators to manage the network. To name some examples, AWESoME allows network managers (i) to block non-authorized services in the network, (ii) to route traffic of given services on specific paths with performance guarantees, (iii) to route suspicious traffic of unknown services to specific devices (e.g., through a security firewall), (iv) to regulate the traffic of pre-selected services.

We will use corporate networks as a running example in the paper (see Figure 2) although AWESoME is applicable to other scenarios too. In the scenario depicted in Figure 2 the corporate network has two links to external networks that deliver different performance, potentially at different costs. In this example, the network administrator may want to forward priority services (e.g., the e-learning platform illustrated in Figure 1) to the best performing link, whereas traffic from non-priority services is forwarded to the best-effort link. AWESoME must guarantee that all traffic of the selected services flow to the desired path. Therefore, all network elements in the corporate network must be programmed to forward traffic according to the per service management approach.

B. Core and support domains

Servers being contacted by clients are identified by their IP addresses, but they are typically reached using their Fully Qualified Domain Names, or *domains* for short. Services that people (or applications) intentionally access are identified by their *Core Domain*: www.nytimes.com, alison.com are core domains (tall arrows in Figure 1). Unfortunately, only a minor fraction of traffic related to a service is served by the core

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

TABLE I: Traffic generated by visiting 10 popular services.

Service	Percentage of flows to			Total domains
	Core domain	Related support domains	Generic support domains	
www.bbc.com	19.4	35.3	45.3	90
www.nytimes.com	17.4	43.7	38.9	63
washingtonpost.com	34.8	21.2	44.0	90
www.ieee.org	37.8	24.3	37.9	17
www.acm.org	43.5	0.0	56.5	8
researchgate.net	5.2	75.5	19.3	29
www.facebook.com	21.9	63.0	15.1	12
www.google.com	8.9	77.8	13.4	141
twitter.com	6.8	86.8	6.4	6
www.youtube.com	5.8	76.9	17.2	30

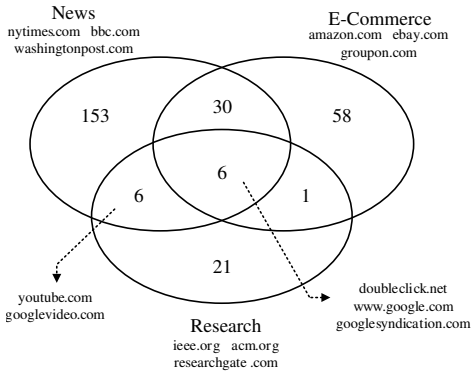


Fig. 3: Support domains shared across different categories of sites. Analytics and advertisement domains are always present.

domain, with *Support Domains* (short arrows in Figure 1) being contacted for analytics, ads, video and image download.

Table I quantifies the traffic related to core and support domains for popular sites. It details the breakdown of flows served by the core domain, by support domains whose name is trivially linked to the core domain (e.g., www.nytimes.com and css.nytimes.com), and by generic support domains (e.g., ads.com). Notice how a large fraction of flows is exchanged with support domains, and that a simple approach taking into account only traffic to the core domain would fail in identifying most of the flows.

More than that, generic support domains are often shared across different websites, and some core domains also appear as support domains for other services (e.g., online social networks). Figure 3 quantifies these cases again for a set of popular sites. It shows 9 websites, grouped into 3 categories. By visiting each site we have collected all contacted support domains. Over 275 total domains, 43 are shared by websites of different categories and 6 domains are present in all categories.

Per service management, therefore, is required to identify *core* and *support* domains. We call *Bag of Domains (BoD)* the set of all support domains contacted when accessing the given core domain. For each core domain, its Bag of Domains must be automatically built from traffic using big data approaches.

C. SDN as enabling technology

We consider an SDN, where users access the Internet via their devices connected to SDN enabled switches or wireless access points, as sketched by Figure 2. The SDN controller manages the network, translating the requirements from the SDN applications to SDN datapath commands. AWESoME interacts with the SDN controller via the NorthBound Interface (NBI), as a standard SDN application. AWESoME operates by installing three types of rules in the network elements: (i) default rules, (ii) per flow rules, and (iii) policing rules.

Default rules are installed on edge switches to forward selected packets to the AWESoME application running on the controller. These rules are summarized in Table IIa: (1) all DNS response packets are normally forwarded, and mirrored to the controller, (2, 3) the first packet of all TCP and UDP flows are forwarded to the controller.[2] The first rule is used to maintain a database that allows AWESoME to associate a flow with a domain name via previously issued DNS requests [9], [10], [11], [12]. The latter rules let AWESoME handle each new flow to subsequently impose the most appropriate actions.

Like any SDN solutions based on such reactive paradigm, the default rules may force the controller to examine a large number of packets. In Section V-E we will show that the load is still limited for a network with moderate number of users. For very large deployments, controller load-balancing solutions should be considered [14], [15]. Reactive SDNs are also exposed to Denial-of-Service attacks – e.g., malicious nodes that exploit rules to overwhelm controllers with lots of packets. Different solutions have been proposed to tackle the issue [16], [17], and they could be employed in our scenario.

Once AWESoME has taken the decision about a new flow, it installs a *per flow rule* on the edge switch to handle the packets of the new flow. Per flow rules aim at guaranteeing that different flows associated to a single service are treated equally in the network. They are transient and thus maintained only while the given flow is active. Table IIb lists rules installed to handle the example presented in Figure 2. Flows that are identified as belonging to selected applications are *tagged* as priority (i.e., Gold class, implemented as VLAN tag `0x001`), whereas the remaining flows are tagged as best-effort class (i.e., Silver, implemented as VLAN tag `0x002`).

Notice that only the first packet of each flow needs to be inspected by AWESoME. Per flow rules guarantee that subsequent packets of the flow do not transit through the controller, but are directly forwarded by edge switches. Since the system adopts a reactive SDN paradigm, packets transiting through the controller are retained by the switch until the controller takes a decision. More in detail, when such a packet arrives to the switch, a copy is sent to the controller using a *PacketIn* message, and holds it in a local buffer. When (eventually) the controller answers with a *PacketOut* message, it is actually forwarded in the network. As a result, clients can only establish connections *after* AWESoME has programmed the edge switch.

Finally, AWESoME programs core switches with predefined policing rules. These rules are stable and installed when the manager deploys an application based on AWESoME. In Figure 2, traffic of each category needs to be

[2]The flow must match the TCP flags using the `OFPXMT_OFB_TCP_FLAGS` field available since OpenFlow 1.5.0. These rules are given low priority to avoid overriding more specific rules.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

forwarded to the particular reserved path. As such, rules to handle and forward the classes are installed in core switches (see Table IIc). For this example on traffic engineering, policing rules are built based on the VLAN tags determined at edge switches. Other mechanisms can be exploited too, such as MPLS labels or Provider Backbone Bridges (PBB) tags.

D. AWESoME architecture

Figure 4 provides a schematic diagram of the AWESoME SDN application. Four elements are identified, each in charge of a logically independent operation, which together enable per service management:

1) *BoD-Training* automatically learns and updates the BoDs in background;
2) *Flow-to-Domain* tags flows with domains;
3) *Domain-to-Service* links named-flows to services;
4) *Service-to-Rule* translates the service into the appropriate actions (i.e., the rules described in previous section).

Below, we describe each of them, while performance and parameter tuning are discussed in Section V.

1) BoD-Training — automatically building BoDs: The BoD-Training block is responsible to automatically build the BoD associated to each core domain. This is the key module in the AWESoME approach, and, thus, we detail it in Section III.

2) Flow-to-Domain — flow labeling using DNS: This step associates a server domain to each flow, i.e., to create *named-flows*. This helps the association of a flow to a given service, since the information offered by IP addresses is much coarser than the one carried by the domain of the server being contacted [18]. This is because single a cloud (and CDN) server may host many services. Intuitively, the same server IP address hosts a multitude of web services which are better identified by their domains. It has been shown that this operation can be solved by leveraging DNS traffic [9], [10], [11], [12]. The Flow-to-Domain block builds a local cache of domains that terminals have resolved in the past, maintained as a key-value store. Below, we describe the two actions of building and using the key-value pairs as *Insert and update* and *Lookup*.

Insert and update: For each DNS response forwarded by the controller, AWESoME extracts the $ClientIP$ address, the domain being queried ($QueriedDomain$), and, from each Answer record, the list of resolved $\{ServerIP_i\}$ addresses. For each key $\{ClientIP, ServerIP_i\}$, it inserts (or rewrites) an entry with value $QueriedDomain$. The time such entries must be preserved in the store (and expired when old) is discussed in Section V.

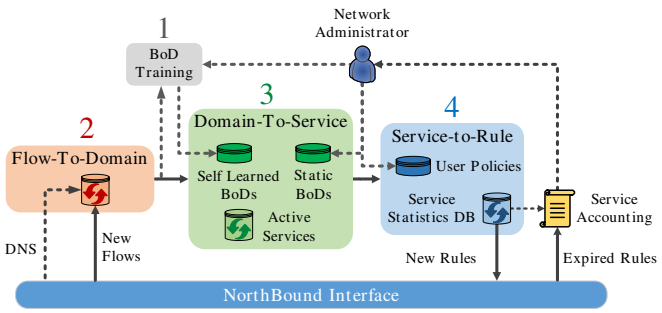


Fig. 4: AWESoME architecture. Databases with arrows are maintained in real-time.

Lookup: Whenever a TCP or UDP packet is forwarded to the SDN controller, AWESoME parses the IP and layer 4 headers and accesses the name store with the key $\{ClientIP, ServerIP\}$ to fetch the original $QueriedDomain$ the client previously resolved. In case there is not such key, the store returns the $ServerIP$. The packet is then forwarded to the Domain-to-Service block, along with the $QueriedDomain$ or, if not available, the $ServerIP$.

Using DNS information has several advantages with respect to more intrusive flow classification methods. First, it does not require to use costly DPI technology to extract hostname or SNI (Server Name Indication) from HTTP or HTTPS requests. Second, DNS information is not protected by encryption, and even DNSSEC does not provide confidentiality. Most importantly, the lookup is done on the very first packet of each flow, eliminating the need of keeping per flow state and waiting for more packets to take a final decision at the controller. As such, when clients finally open a connection, the network is already programmed to handle the traffic accordingly. On the downside, erroneous domain associations can happen due to collisions (i.e., a rewrite operation) – the same $serverIP$ being contacted by the same $clientIP$ for two different $QueriedDomain$. As we will see later, AWESoME is robust to such events.

3) Domain-to-Service — associating services to flows: Once the flows are labeled with DNS names, AWESoME associates the named-flows to services. This is the core of AWESoME engine and its details are provided in Section III. Leveraging a well-known text mining technique, Bag-of-Words, we model semantics of the domain names, namely, *Bag-of-Domains (BoD)*. A BoD is created for each core domain and includes all support domains that are contacted when the service identified by the core domain is accessed.

We further group the domains into two types: *Self Learned BoDs*, and *Static BoDs* based on the characteristics of the domains. Automatically built by AWESoME while analyzing the traffic, Self Learned BoDs are BoDs comprised of interactive web services, which users *explicitly* access from their browsers, e.g., interactive web applications. On the other hand, manually built by network operators, static BoDs are comprised of *background* services that are periodically accessed by terminals without user intervention (e.g., software updates, file sync with cloud storage services, calendar or mail services, etc).[3] The traffic generated by such services is quite different from the interactive ones where core domains and support domains are expected to appear close in time (see Figure 1). Background services challenge the assumption of temporal correlation between flows, and extending AWESoME to learn Static BoDs automatically is left for future work.

The list of recently accessed core domains by each $ClientIP$ is stored in the *Active Service* database. Keeping a cache of active services is important since the same domain

[3]In the current implementation, regexp and wildcards are supported in the specification of static BoDs to simplify the administrator's task.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

TABLE II: Rules to be installed on the SDN switches across the network.

(a) Default rules installed at edge switches control the traffic that needs to pass the controller for taking decisions.

N	Match	Action	Description
1	IP_PROTO=UDP and UDP_SRC=53	Forward, Forward to Controller	Spill DNS responses
2	IP_PROTO=TCP and TCP_FLAGS=PURE_SYN	Forward to Controller	Intercept new TCP connections
3	IP_PROTO=UDP and UDP_DST!=53	Forward to Controller	Intercept all UDP non-DNS traffic

(b) Transient per-flow rules are installed at edge switches to tag each flow (e.g., f_1 and f_2) with the respective service label.

N	Match	Action	Description
1	IP_PROTO=TCP and IPV4_SRC=$IP_{SRC}^{f_1}$ and TCP_SRC=$TCP_{SRC}^{f_1}$ ...	Push VLAN tag, VID=0x001	Tag as *Gold*
2	IP_PROTO=TCP and IPV4_SRC=$IP_{SRC}^{f_2}$ and TCP_SRC=$TCP_{SRC}^{f_2}$ ...	Push VLAN tag, VID=0x002	Tag as *Silver*

(c) Stable policing rules are installed in core switches to steer packets according to the application scenario faced by AWESoME. In this example, traffic of each class is forwarded to particular network path (see also Figure 2).

N	Match	Action	Description
1	VLAN_VID=0x001	Output on P_1	Forward *Gold* traffic towards the reliable link
2	VLAN_VID=0x002	Output on P_2	Forward *Silver* traffic towards the best-effort link

normally appears in multiple BoDs – cf. Figure 1 – and thus it must be associated to a core domain that has been actually visited. Given a flow, the Domain-to-Service block checks if its domain appears in the BoDs of $ClientIP$ Active Services, so to associate to the most likely service the user has recently accessed. In case a domain is not in the active services, it falls back to match against Static BoDs.

The packet is then forwarded to the Service-to-Rule block, along with the service corresponding to the BoD the packet was associated with.

4) Service-to-Rule — policy enforcement: Once a flow has been associated to a service, the Service-to-Rule block is a classic policing module which enforces actions by requesting the SDN controller to install rules on the switches. Policies are stored in a *User Policies* database, which is accessed with the service name as key, and returns the corresponding rules.

Policing rules are installed (e.g., in core switches) when the AWESoME application is started, whereas the per flow rules are pushed whenever a flow must be steered. Table II has already exemplified the rules created for the particular traffic engineering case used as illustration, but other rules can be defined too – e.g., to block services, route traffic to security devices or to regulate the traffic per service. Rules expire using the `Idle Timeout` standard OpenFlow feature.

In case of "default" action, no extra rule has to be added for TCP flows since only the SYN-TCP packet will be forwarded to the controller. The lack of explicit connection indication in UDP forces AWESoME to insert a rule for each UDP flow. However, only the first packet of UDP flows transits through the controller, while the others are directly forwarded by the switch, as a transient per-flow rule is inserted.

Scalability is evaluated in Section V-E. Again, *per flow* rules have to be installed on the edge switches only – i.e., those switches that are directly connected to clients or work as ingress point to the SDN. Upstream devices instead operate on a *per service* basis, e.g., using IP Type of Service, MPLS labels or PBB tags, which are all supported by SDN.

The Service-to-Rule block additionally maintains the *Service Statistics DB*, with flow identifier (e.g., the classic 5-tuple) as key, and service information as value. When a rule expires at switches, its flow identifier is passed along with statistics (byte and packet amount) to the SDN controller (standard in `FlowRemoved` messages) that, in turn, exposes them to the AWESoME application. Such statistics are collected in the Service Statistics DB, later used for BoD training, and exposed to the network administrator. This enables for instance per service accounting, anomaly detection, billing, etc.

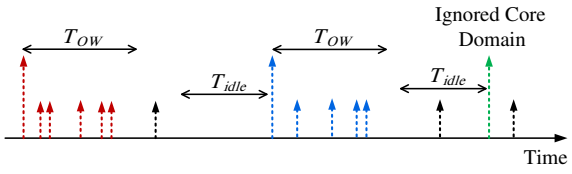
III. How Service Association Works

The core of the service management is the ability to associate each flow to the originating core domain, i.e., the service the user originally intended to access. AWESoME solves this by leveraging the bag-of-words model which is commonly used to succinctly representing complex textual data in natural language processing [19]. In the context of AWESoME, we extract bag-of-words features from each domain and "classify" it into a service. Hence we call the process Bag-of-*Domain* (*BoD*) training. Due to the complex composition of web pages and the intertwined nature of the Internet, it is not trivial to design the BoD training with minimal human intervention.

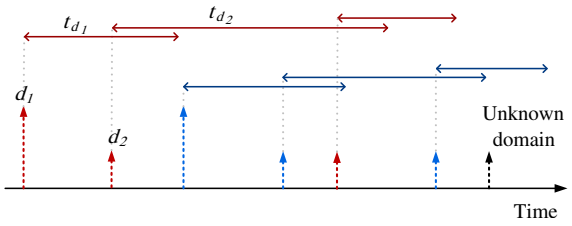
A. Automatic BoD training

Let C be the set of core domains of interest provided by the network administrator. AWESoME training consists of building a BoD_c, for each core domain $c \in C$. One possible solution could be using active crawling, e.g., artificially visiting the service/website of c and collecting domains being contacted. Unfortunately, this does not work in practice (cf. Section V) since (i) the same service/website changes when accessed from different identities, locations, time, browsers, devices, configuration, etc., (ii) c may require authentication, or the usage of a specific application which complicates the cluster, and (iii) the approach poses scalability issues.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management



(a) BoD-Training: a flow to a core domain triggers a new observation window if client was idle for more than ΔT_{idle}.



(b) Domain-to-Service: Blue and red services are active at the same time; the EWs are extended as new flows are associated to the core domain.

Fig. 5: Training and annotation examples.

AWESoME leverages instead data collected from the network itself to build and update the BoDs.

The intuition is simple: when a client is observed opening a flow to domain c, the domains of flows that follow shall be inserted in BoD_c. Unfortunately, this step is challenging because the user may access multiple services at the same time (e.g., using multiple browsers), while the terminal may contact other services (e.g., for software updates, or background services). In addition, the same support domain may belong to multiple BoDs, or, worse, a domain may be both a core domain, and a support domain.

The Flow-to-Domain block outputs each named-flow f generated by each *ClientIP*. Figure 5a depicts a possible timeline of flows generated by a given client. Tall arrows are visits to core domains. AWESoME conservatively considers a flow f as a possible core domain if $f \in C$ and $ClientIP$ has been idle for more than T_{idle}. In Figure 5a, red and blue tall flows are identified as core domains, while the green flow is not.

When a new core domain is observed, AWESoME opens an *Observation Window* (OW) of duration T_{OW}. All domains of flows observed in T_{OW} are inserted in the BoD_c. In Figure 5a, this is represented by coloring flows with the same color of the core domain. The longer T_{OW}, the more information is collected, with the chance to pollute the BoD_c with *false support domains*. Algorithm 1 shows a pseudocode for the BoD update function.

To distinguish false support domains, AWESoME computes the frequency with which each domain appears in BoD_c across multiple observations: domains below a $MinFreq$ threshold are filtered. The intuition is that actual support domains emerge, whereas the false support domains that seldom appear can be filtered out by frequency. This step lets the system filter out background traffic and those flows due to (possible) concurrency in the monitored clients. $MinFreq$ is a parameter of AWESoME and its tuning is discussed in Section V-B,

Algorithm 1 $BoD\_update(f, C, BoDs)$

Input:
f ▷ *The current flow*
$C = \{c_1, ..., c_k\}$ ▷ *Core Domains*
$BoDs = \{BoD_{c_1}, ..., BoD_{c_k}\}$ ▷ *BoDs of core domains in C*

1: $t = GetTime()$ ▷ *Get current time*
2: $d_f \leftarrow parse(f)$ ▷ *Get the domain of f*
3: $(t_c, c) \leftarrow OW$ ▷ *Retrieve current OW if any*
4: **if** $OW \neq \emptyset \wedge t - t_c \geq T_{OW}$ **then**
5: $OW \leftarrow \emptyset$ ▷ *Remove the OW if expired*
6: *// Put domains in the Bag if OW exists*
7: **if** $OW \neq \emptyset$ **then**
8: $BoD_c(d_f) += 1$
9: **else**
10: **if** $d_f \in C \wedge t - t_{last} > T_{idle}$ **then**
11: $OW \leftarrow (t, d_f)$ ▷ *Open a new OW*
12: $freq_{d_f} += 1$ ▷ *Update CD frequency*
13: $t_{last} \leftarrow t$ ▷ *Update last flow time*

along with evaluation of the effect of concurrency in the clients. Traffic from all clients contributes to learn BoD_c, so that information is accumulated over time and in different conditions (i.e., different identities, time, browsers, devices, configuration, etc.).

Domains are stored in a LRU cache of limited size (e.g., 5 000 entries). This is more than adequate – cf. Table I – and limits memory usage.

At last, AWESoME needs to compute the average flow duration per each domain in BoD_c. This is done using the flow duration information as exposed by the Service-to-Rule block. For each domain d, AWESoME maintains the average flow duration t_d. To cope with possible changes in service behaviors, a standard exponential moving average estimator is used (parameter $\alpha = 0.1$). AWESoME is however almost insensitive to the parameter as site changes occur in much longer time periods than the re-training of BoDs.

Flow duration is fundamental to AWESoME as we will see next, and OpenFlow `Idle Timeout` mechanism lets the controller to derive it. After no packet has matched a rule for a configurable period, a `FlowRemoved` message is sent by the switch to the controller. Flow duration is obtained by computing the time between the rule install action and the `FlowRemoved` message, subtracting the `Idle Timeout` set in switches.

AWESoME takes advantage of the last days of traffic to build the BoDs used by the Domain-to-Service module. A discussion about the time needed to build BoDs is provided in Section V. The training dataset potentially becomes large in real scenarios, and thus the BoD-Training module is implemented in a state-of-art big data platform, namely Apache Spark. The statistics to build BoDs are continuously collected in the BoD-Training module. Periodically, e.g., once per hour, BoDs are computed and given to the Domain-to-Service for on-line annotation of traffic.

B. Domain-To-Service classification module

Armed with core domains and their respective BoDs, AWESoME has to associate named-flows with the service identified by the core domain. It first tries to associate the flow to any BoD in the Self Learned BoDs. In case of no match,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

it then tries with Static BoDs. For the sake of simplicity, we describe only the first stage as the second is identical.

Figure 5b gives an example of a possible timeline where a *ClientIP* accesses first to the red service, and then to the blue services.

AWESoME uses Algorithm 2 to annotate each flow f. It receives: (i) the current named-flow f, (ii) the set of core domains C, (iii) the BoDs, (iv) average duration t for each domain. It processes each *ClientIP* separately and keeps separate data structures. It outputs a flow annotated with the core domain, or *unknown* in case no association is found.

Algorithm 2 $annotate(f, C, BoDs, T)$

Input:
f ▷ *The current flow to annotate*
$C = \{c_1, ..., c_k\}$ ▷ *Core Domains*
$BoDs = \{BoD_{c_1}, ..., BoD_{c_k}\}$ ▷ *BoDs of core domains in C*
$T = \{t_{d_1}, ..., t_{d_1}\}$ ▷ *Domain average flow duration*
Output:
$O = (f, CoreDomain)$ ▷ *Annotated flow*

1: *// Retrieve start time and domain of f*
2: $t = GetTime()$ ▷ *Get current time*
3: $d_f \leftarrow parse(f)$ ▷ *Get the domain of f*
4: *// Remove expired Services*
5: $AS \leftarrow \{(ts, te, c_i, BoD_{c_i}) \in AS | t \leq te\}$
6: *// Obtain the best BoD among the AS*
7: $as_{best} \leftarrow \{(ts, te_{best}, c_{best}, BoD)\} \leftarrow BestBoD(d_f, AS)$
8: **if** $d_f \in C \wedge as_{best} == \emptyset$ **then**
9: *// d_f is a core domain – Start a new AS for d_f*
10: $c = d_f$
11: $AS \leftarrow AS + \{(t, t + t_c, c, BoD_c)\}$
12: $O \leftarrow (f, c)$
13: **else**
14: **if** $as_{best} \neq \emptyset$ **then**
15: $O \leftarrow (f, c_{best})$ ▷ *The flow is assigned to c_{best}*
16: *// Update the AS validity time*
17: $te_{best} \leftarrow \max(t + t_{d_f}, te_{best})$
18: **else**
19: $O \leftarrow (f, "unknown")$ ▷ *Flow not classified*

The algorithm is based on the concept of *Evaluation Window* (EW), i.e., a time during which a support flow can still appear after the observation of the core domain c. The algorithm maintains a list of Active Services, AS, i.e., those core domains previously seen, and for which it is still possible to associate some flows. The list grows as new core domains are observed (lines 8–12), and entries are aged out, i.e., window ending time t_e is passed (line 5).

First, AWESoME checks if there exists a Active Service as_{best} whose BoD contains the domain of f. In case more than one AS matches, we consider the $as_{best} = BestBoD(d_f, AS)$ as the one whose evaluation window start time is the closest in time (line 7). Intuitively, we consider the most recent visited core domain as the most likely one to associate the current support domain. We tried other choices, e.g., considering random choice, weighted choice by the frequency of occurrence in BoDs, etc., with worse results.

Next, AWESoME has to resolve the ambiguity for domains that can appear as both support and core. If d_f is a possible core domain, and there exists no AS in which it appears as support domain (line 8), then it is considered a new core domain, and a new evaluation window is opened (lines 9-12). The rationale is that the domain has been contacted because of an intentional visit from the user.

On the contrary, d_f is considered a support domain if there exist an active service as_{best} (line 14). The flow is associated to the core domain c_{best} (line 15), and the evaluation window ending time te_{best} is extended (line 17) to consider the average duration of the current flow t_{d_f}. The rationale is that flows to support domains may be observed long time after the core domain, since the terminal keeps downloading objects due to a user action, e.g., scrolling a web page that triggers the download of new elements, or the download of a new video chunk in a streaming service. This is sketched in Figure 5b where the evaluation windows are represented by horizontal arrows, which extend the AS ending time.

Finally, in case of no match with any AS, the flow is associated to the "unknown" class (line 19), and AWESoME looks for a matching in the Static BoDs.

It is important to notice that the Domain-To-Service module operates on a per-flow and per-*ClientIP* basis and, thus, the processing is amenable for per-client parallelization.

IV. Datasets

We validate AWESoME and evaluate its performance using trace-driven analysis. First, we thoroughly assess classification performance using traces where ground-truth is available – i.e., where we have information about the core domain responsible for the visit to support domains. Then, we use passive traces collected at operational networks to study realistic AWESoME deployments.

A. Ground-truth traces

We rely on ground-truth data from volunteers. We collect browsing histories of 30 users, directly extracting the URLs they intentionally visited in the past months, which are stored in a local database by their browsers. We automatically revisit each URL by instrumenting a Chrome browser. We let Chrome visit the URL and wait until the page is fully loaded (i.e., the `On Load` event is fired).

In parallel, we record all network activity in our environment to have a complete picture of the traffic that would be managed by AWESoME. The outcome of these steps is a dataset of named-flows, where each entry is annotated as a core domain, if it was a URL given as input to the instrumented browser; or as a support domain, if it was triggered by a core domain visit.

In total, we collected 973 000 flows, referring to 3 760 and 97 640 unique core and support domains, respectively. Crawling was done in December 2016 and lasted 5 days. We build three traces from this raw dataset:

- **Simple-browsing:** It mimics the original behavior of volunteers. Each volunteer is given a unique $ClientIP$ address, and we simulate page visits in the same sequence and with the same visit time of the original browsing history. The arrival time of support flows after a core domain visit respects what is seen during crawling.
- **Tab-browsing:** We create this scenario by repeating the previous steps, but starting 5 independent navigation threads per $ClientIP$ in parallel. To avoid any kind of synchronization

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

TABLE III: Traces collected from operational networks.

Dataset	Duration	Flows	Unique Domains	Client IP addresses
ISP 1	12 months	13 billion	18 million	$\approx$ 10 000
ISP 2	12 months	4 billion	6 million	$\approx$ 1 000
$Corp$ 1	1 day	6 million	$\approx$ 38 000	$\approx$ 1 600
$Corp$ 2	3 days	32 million	$\approx$ 64 000	$\approx$ 6 000

among threads, each navigation starts following the browsing histories at a random position. This scenario emulates (i) an extreme case of multi-tabbed browsing where the same user has 5 tabs concurrently and continuously browsing the web; or (ii) 5 users concurrently and continuously browsing the web behind a NAT (i.e., identified by the same client IP address). The latter is a typical setup in ISP environments where a single home gateway acts as a NAT, and a handful of household devices access the Internet contemporarily with the same identifier. Core and support domains of many visits may appear simultaneously in the trace. This challenges the disambiguation of core and support domains.

• **Simple-browsing + video streaming:** Our crawling based on volunteers' histories notably miss video streaming sessions, since videos may continue playing after the `On Load` event is fired. Traffic generated by video servers might be quite different from interactive browsing because flows to retrieve video chunks have low temporal correlation with the core domain request [20]. Using the instrumented browser, we record all traffic generated when accessing 250 arbitrary URLs from 15 sites with embedded videos. We let the video play for 5 min before moving to another page. We finally mixed the Simple-browsing trace by simulating a second parallel thread for each volunteer. This thread continuously watches videos, with the user changing page every 5 min, without any pause in between. This is again an extreme case to test.

B. Operational network traces

We capture flow-level datasets from operational networks using passive meters. Our captures include four measurement locations: two ISP networks and two corporate networks. The datasets are summarized in Table III. To preserve users' privacy, IP addresses have been anonymized, and we kept only the information required for our study. Trace collections has been approved by ISP and corporate security board.

1) ISP traces: ISP 1 and ISP 2 traces include data exported by flow exporters deployed at different Points of Presence (PoP) of a large ISP. These PoPs connect mostly residential customers to the Internet, via DSL or FTTH. The ISP assigns to each home gateway a fixed IP address, thus allowing us to isolate the traffic per home gateway. Traces include traffic generated by all users' devices connected to the Internet via Ethernet and/or WiFi.

The traces include basic features found in typical NetFlow exporters, summarizing TCP and UDP flows – e.g., byte/packet counters, client and server IP addresses, etc. The flow exporters provide the Flow-to-Domain mapping performed by AWESoME by processing the DNS traffic on-the-fly. We captured data from January to December 2016, monitoring around 11 000 households. In total, we have observed more than 17 billion flows, 18 million domains.

We additionally dumped DNS traffic in the PoPs for 6 hours in December 2016, simultaneously to the flow exporting, for some specific analysis that will follow.

2) Corporate traces: We rely on proxy logs from enterprise networks to assess AWESoME performance in corporate scenarios. They come from two different enterprises in different states of the USA. The proxies provide web connectivity to thousands of employees of two companies. They save logs for (i) each HTTP request and (ii) each `CONNECT` command for HTTPS tunnels. Clients are consistently identified by IP addresses. No UDP traffic is allowed.

We directly construct a named-flow log from each of the raw proxy logs, creating the $Corp$ 1 and $Corp$ 2 datasets. We proceeded as follows: for each `CONNECT` and for each HTTP request entry we create a flow record for the involved client and server. The domain is extracted directly from the hostname in HTTP request and from the `CONNECT` command. Naturally, this approach will over-estimate the number of flows in the network, since TCP flows are reused by clients when communicating with a HTTP server.

V. AWESoME Performance

A. Flow-to-Domain evaluation

We evaluate the Flow-to-Domain block aiming to answer two questions: (i) What is the percentage of flows that can be annotated with DNS information? (ii) How long should the DNS information be cached to perform flow annotation?

Let us start focusing on the first question to check how many flows would remain unnamed due to lack of DNS data. To answer this question we use the 6-hour-long dataset in which we have both named-flows and raw DNS traffic from the ISP probe. We look for web flows, i.e., to port 80 and 443, which have a domain associated. In particular we simulate the Flow-to-Domain module with infinite memory. To avoid boundary effects, we warm up the domain key-value store loading the initial 5 hours of the DNS trace and then we use the last hour of the flow trace to perform a lookup while still processing the DNS trace at the same time.

We found that 93% of the web flows are annotated. Manual inspection reveals two main causes for the missing domains: (i) $\approx 1\%$ of services contact support servers using directly IP addresses; (ii) possible loss of DNS packets during the passive captures. AWESoME handles the first case by adding IP addresses to BoDs, whereas the second case is a measurement artifact that should not happen in real SDN deployments.[4]

Figure 6 presents the Cumulative Distribution Function (CDF) of the delay between the flow and the previously issued DNS query. For both traces, more than 90% of the flows starts within 10 minutes since the corresponding DNS query. The percentage grows to 97% considering a 1-hour interval.[5] These large time gaps between DNS queries and the flows are mostly

[4]In SDN, packets sent to the controller are always received thanks to the usage of reliable transport protocols between switches and the controller.

[5]Differences for small x-values occur due to variations in the RTT between clients and the flow exporters.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

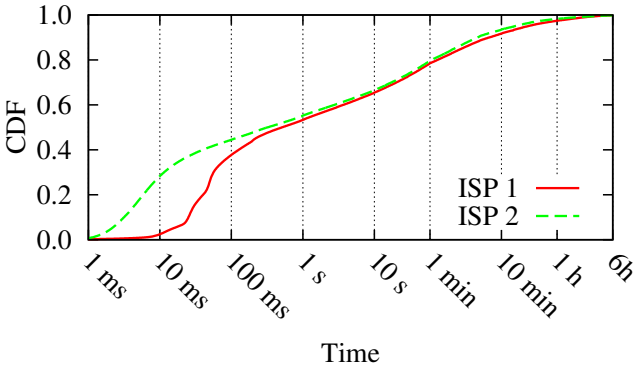


Fig. 6: Time between TCP flows and their associated DNS query. AWESoME needs to cache information about 1-hour of DNS traffic to annotate flows.

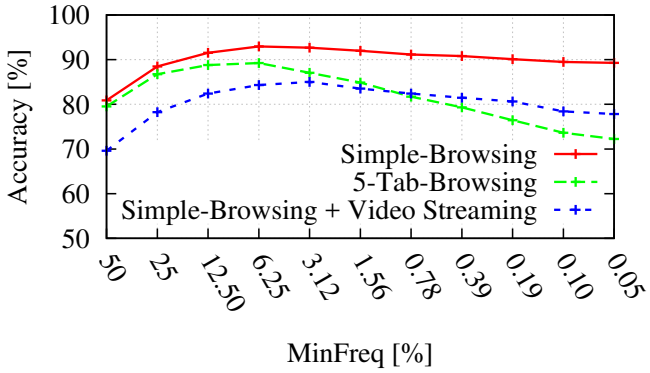


Fig. 7: Accuracy when varying the $MinFreq$ threshold. AWESoME accuracy surpasses 93% in the Simple-Browsing trace, and 85% in extreme scenarios.

due to large TTLs of DNS responses and client-side caches – i.e., clients can open flows to servers long after resolving their names thanks to the local DNS cache. Nevertheless, the figure shows that the Flow-to-Domain block must be sized to hold in its key-value store the information extracted from about 1-hour of DNS traffic in order to output high-quality named-flows. In the largest of our traces this corresponds to manage about 1 000 000 entries.

B. Domain-to-Service accuracy

We next evaluate the core part of AWESoME – i.e., the association of services to named-flows. We use the ground-truth traces for this validation. We only check the accuracy for self learned BoDs, since static BoDs are manually provided by network administrators.

We estimate the accuracy of AWESoME by checking whether the service determined for each flow matches with the ground-truth. In this experiment, learning of BoDs is performed using the ground-truth trace. All 3 760 core domains are considered with 3 760 BoDs built from the trace itself. We here test such a case in which every service would be managed independently to evaluate AWESoME performance in extreme cases. In more practical scenarios (e.g., Figure 2), one would expect only a limited number of key services to be classified and managed. We also repeat experiments with different settings to study the effects of AWESoME parameters.

Figure 7 depicts the performance of AWESoME when varying the $MinFreq$ threshold used for learning. Recall that a support domain is discarded from the BoD if it appears less frequently than $MinFreq$. Curves for three scenarios are depicted.

Focusing in $MinFreq = 6.25\%$, notice how the accuracy of AWESoME is high, reaching close to 93% in the Simple-Browsing trace. Errors are related to flows annotated with ambiguous domains (i.e., belonging to more than one BoD) or left as "unknown" (e.g., no active window during classification). Even for the extreme traces, AWESoME delivers accuracy close to 85%. That is, AWESoME can identify flows per service with high accuracy, even in challenging situations that should be uncommon in real deployments. In particular, for the 5-Tab-Browsing trace the performance penalty is very small. This means that AWESoME can successfully operate when users perform tab-browsing, or in typical ISP scenarios where devices of a household access the network with the same client identifier.

We however remark that the tested 5-Tab-Browsing scenario does not guarantee that AWESoME would work with *any* level of parallelism. Carrier Grade NAT, in which hundreds or thousands of users are aggregated, is an example where the deployment of AWESoME requires planning. Switches inside the NAT-ed network need to be part of the SDN as well, thus aggregating a moderate number of users, which will ensure AWESoME delivers performance as in Figure 7.

Notice also the importance of $MinFreq$ to filter out noise from BoDs. When $MinFreq$ is large (e.g., 50%), domains that are popular in BoDs are ignored, resulting in a sharp decrease on accuracy. On the other extreme, when $MinFreq$ is low, false support domains pollute the BoDs. Focusing on results for $MinFreq = 0.1\%$, notice how accuracy drops to 90% in the Simple-Browsing trace, and to less than 80% in extreme scenarios. This happens because BoDs get very large with lots of false support domains that hinder the annotation process.

We omit for brevity analyses with other parameters. Overall, the best parameter choices are $T_{OW} = 10$ s, $T_{idle} = 5$ s and $MinFreq = 6\%$, resulting the best figures shown in Figure 7.

C. Training set size and location

AWESoME learns BoDs by observing traffic. We now answer two practical questions regarding training in real deployments: (i) What is the amount of traffic that needs to be observed for learning representative BoDs? (ii) Should training be performed with traffic of the managed network or generic BoDs can be distributed to different deployments?

Figure 8 shows the effects of the training dataset size. For this experiment, AWESoME learns BoDs using ISP 1 trace, and performance is assessed with the Simple-Browsing trace. We extend the training dataset duration in each experiment

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

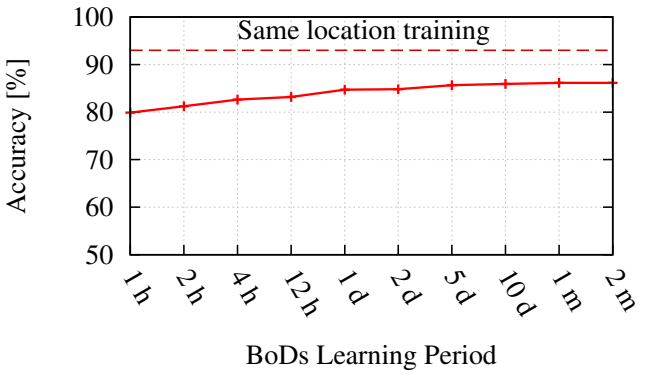


Fig. 8: Accuracy vs. training dataset size. BoDs learned with ISP 1, accuracy calculated with Simple-Browsing trace – 1-month training window is sufficient.

round. The *"same location training"* line marks the best result obtained with training performed with Simple-Browsing trace. Again, AWESoME has to learn 3 760 BoDs. Here, we want to study the effect of different learning periods, and, thus, the study is limited to the ISP 1 trace. $Corp$ 1 and $Corp$ 2 are captured very far in space, and this would lead to worse results. This effect is evaluated later in this section.

Focusing on the left-most point in Figure 8, note that AWESoME correctly identifies 80% of the flows when the training set contains 1 hour of traffic only. That is, most of the popular BoDs are learned by observing a single hour of traffic. Increasing the training set improves results, with the best accuracy at around 87%. Thus, AWESoME needs to be trained for around 10 days to reach its best performance in this scenario. Further results, omitted for brevity, show that BoDs change slowly and are well-captured by the continuous training.

Since AWESoME requires historical data for training, the size of the training dataset may become large. For ISP 1 and ISP 2, this corresponds to millions of flow records, which result in several GBs of traces. This calls for the use of scalable data processing approaches, and AWESoME training is thus built on Apache Spark to scale with the size of training dataset.

Figure 8 points to a decrease in performance when training is done with data from a different network. We explore this effect in Table IV. It reports the fraction of flows identified by AWESoME in a trace when training is done on another dataset. Columns indicate the training dataset, and rows indicate the testing dataset. We consider as core domains the top-100 Alexa sites, since most of them are common across traces.

Cells report fractions taking as reference the flows which are annotated when training and testing are done with the same network. For instance, the first row shows that when training is performed with $Corp$ 1, AWESoME annotates only 45% of the flows in ISP 1 that would be identified if both training and testing are done in ISP 1. The remaining 55% of flows are marked as "unknown". This happens because the BoDs learned from different vantage points differ because of variations in the domains used by CDN servers or different content (ads) per location. Additionally, some BoDs are completely empty in a trace because of regional browsing preferences.

TABLE IV: Fraction of flows classified by AWESoME when varying training and testing locations. The Alexa top-100 websites are core domains in this analysis.

Classif. \ Training	ISP 1	$Corp$ 1	$Corp$ 2	$Crawling$
ISP 1	1	0.45	0.48	0.32
$Corp$ 1	0.72	1	0.81	0.40
$Corp$ 2	0.42	0.47	1	0.34

Interesting, last column of Table IV shows that active crawling is not sufficient for generating comprehensive BoDs. We learned BoDs by active crawling the homepage of top-100 Alexa sites. Those classify as little as 32% of the flows for the ISP 1 trace. Therefore, AWESoME deployments must include mechanisms for in-place training.

D. Per service performance

We investigate further AWESoME performance by breaking down results for popular services in our datasets. Figure 9 shows precision and recall obtained when learning BoDs using 10 days training on ISP 1 and applying them to the Simple-Browsing trace.[6]

Figure 9 shows that precision is typically higher than 97% excluding Facebook and Linkedin. That is, false positives are generally very rare unless for those service that are (i) extremely popular and (ii) both core and support domains. AWESoME may consider a support domain as a new core in these cases. Recall is typically higher than 80% – i.e., some support domains are not associated to the right service, typically becoming unknown. For management purposes, this translates into a marginal probability of wrongly treating few flows of a service of interest. For instance, in some case, some support domains are not identified and handled as all other flows in default classes. Finally, we argue that this a pessimistic scenario, as AWESoME has been instrumented to discern all 3 760 Core Domains in the trace i.e., it must classify the traffic into the same number of classes. In a real deployment, we expect lower misclassification probability.

E. Is AWESoME scalable?

Finally, we evaluate three key aspects for a practical AWESoME deployment: (i) its overall run-time to take a decision when a new packet is received by the controller; (ii) the number of packets that need to be handled by the SDN controller; (iii) the number of rules that are installed in forwarding devices.

AWESoME has been prototyped in Python. Figure 10 shows the CDF of the execution time of our prototype for each packet that arrives at the controller. We found that AWESoME running on a commodity server takes less than 100 μs to take

[6]Precision is calculated as the percentage of flows correctly identified as belonging to a service, whereas recall indicates the percentage of flows of the service that is identified.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

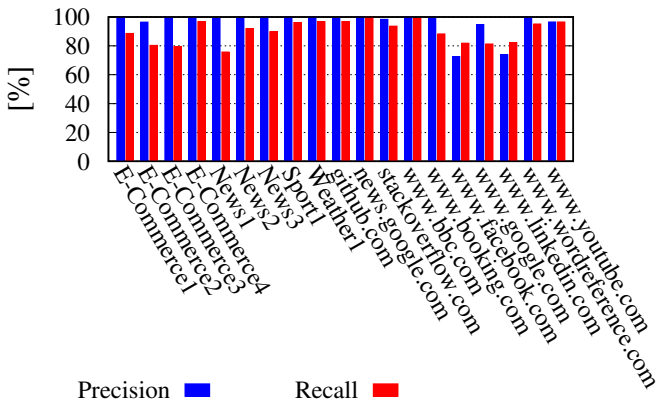


Fig. 9: Precision and recall for popular services.

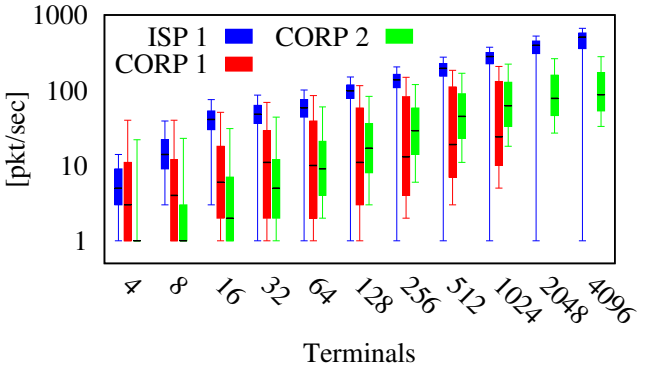


Fig. 11: Packet arrival rate at the controller. Even for large numbers of clients, the number of packets handled by the controller is limited.

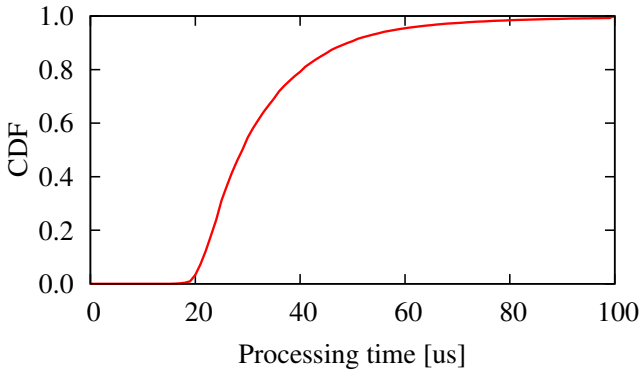


Fig. 10: Processing speed of AWESoME for each packet arriving at the controller in a single-core of a commodity server.

a decision for more than 99% of the packets reaching the SDN controller. That is, AWESoME internals add only negligible delays per flow.

Let us focus on the number of packets the controller has to handle. We use our operational network traces for this. Figure 11 depicts the number of packets per second forwarded to the controller. Different experiment rounds are executed, including the top-n most active *ClientIP* addresses in each round. Remind that client IP addresses are equivalent to home gateways in ISP traces and to unique users in corporate traces. Box plots depict the distributions of packets per 1-second time bins, with boxes ranging from the 1st to 3rd quartiles, and whiskers marking 5^{ft} and 95^{ft} percentiles. Only ISP 1, $Corp$ 1 and $Corp$ 2 are shown to improve visualization.

In summary, the packet arrival rate at the controller is very low. Focusing on the right-most point in Figure 11, notice that 4 096 terminals generate less than 1 000 packets/s for more than 95% of the time bins. For the sake of comparison, our AWESoME Python implementation can handle more than 40 000 packets per second. That is, even for large numbers of clients, AWESoME deployment is scalable thanks to its ability to take decisions using only DNS traffic and a single packet per flow.

Finally, we investigate the number of rules that are installed on the SDN switches. This aspect must be necessarily taken into account, since switches typically can host a limited number of rules (i.e., $< 10\,000$). Notice that AWESoME imposes the largest load in edge switches, where packet policing and tagging are performed on a *per flow* basis (see transient *per flow* rules in Table IIb). Other upstream elements (i.e., SDN switches in the core of the network) instead operate on a *per service* basis, and are programmed using *stable rules* as illustrated in Table IIc. As such, the loading on switches that are upstream in the network should be lower than on edge switches. In the example of Figure 2, stable rules are based on VLAN tags and impose only one rule per traffic class in the core switches.

We estimate the number of rules that would need to be installed in edge switches creating a scenario where we assume: (i) the top-n most active clients are connected to a single edge switch; (ii) the network administrators policy *all* services in the network, thus requiring rules to manage every TCP/UDP flow individually; (iii) rules stay active for `Idle Timeout` = 120 s after the last flow packet.

Figure 12 illustrates the distribution of the number of rules installed in the edge switch. The distribution is calculated monitoring the flow table while replaying ISP 1 trace. The box plots follow the same characteristics as in Figure 11. This study is limited to ISP 1 for the sake of brevity. Similar results are obtained using the available traces. Focusing on the right-most point of Figure 12, notice that the flow table occupancy is low even when more than 500 terminals are connected on a single switch. The switch would rarely observe more than 5 000 active rules. The figure also breaks down numbers per TCP and UDP flows, showing that most rules would be related to (long-lived) TCP flows. In real deployments, where only few services of interest are managed, we expect AWESoME to put a negligible number of rules per switch.

F. Limitations and future work

Previous sections have shown that AWESoME is able to steer traffic per service with an overall accuracy of about 90%.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2017.2785878, IEEE Transactions on Network and Service Management

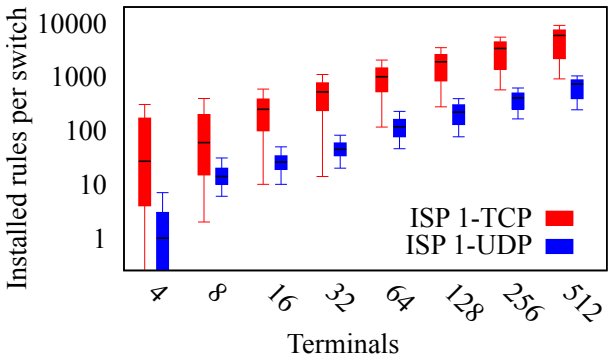


Fig. 12: Active rules assuming that the top-n clients are connected to a bottleneck-switch and *all* services are managed. The number of installed rules is limited.

Whether or not this accuracy is sufficient depends on the target application. For traffic engineering in a corporate scenario (see Figure 2) AWESoME accuracy is appropriate. AWESoME would steer 90% of the flows on the paths selected by the network administrator, with the wrongly routed flows imposing minor loads to the remaining paths. Comparing this error rate to today's alternatives (e.g., routing based on IP addresses of core domains – see Table I), we believe AWESoME is a step forward for traffic engineering.

However, some scenarios may not tolerate *any* false positives, which is the case for some security applications. Devising per service tagging with zero false positive rates for security purposes is left for future work.

AWESoME has some limitations originating from assumptions and design decisions. Those decisions are justified by our goal of keeping AWESoME as simple as possible. For instance, Algorithms I and II assume that services are interactive and, as such, core and support domains appear close in time. BoDs for background services cannot be learned by these Algorithms since the assumption does not hold for background services. Our experience with the traces suggests that background services are, in general, easier to identify thanks to the machine-to-machine nature of the traffic and the low number of domains supporting the services. While AWESoME allows administrators to specify Static BoDs, extending the system to automatically learn BoDs of background services is a promising direction for future work.

Finally, AWESoME assumes edge switches are part of the SDN and aggregate a moderate number of users – e.g., users in home NAT or in a corporate LAN. AWESoME cannot be deployed if large numbers of users are aggregated behind a single address, such as in Carrier-grade NAT, unless edge switches inside the Carrier-grade NAT are part of the SDN.

VI. Related work

A. Web service traffic identification

Many approaches for traffic identification have been proposed [21], [22], and different alternatives could be coupled with SDN to implement per service traffic management. DPI has been employed not only to classify traffic of web services [23], [24], [25], but also to bring service visibility into SDN [4]. The DPI-based approach however suffers from weaknesses when applied to SDN: (i) the number of packets to be forwarded to controllers or SDN applications can be high for common protocols; (ii) as encryption gains momentum, essential information cannot be observed, thus reducing its applicability.

AWESoME adopts a behavioral identification approach – i.e., traffic behavior is used to infer the services generating packets [26]. The AWESoME approach is innovative in that it builds models based on server hostnames as they are resolved by clients. As such, AWESoME can differentiate services even if they use exactly the same protocols (e.g., HTTPS) and are hosted in the same infra-structure (e.g., in the case of CDNs and cloud hosting).

The idea of annotate traffic on-the-fly using DNS information has appeared in [9], [10], [11], [12]. However, AWESoME not only annotates flows and classifies traffic on a per-flow basis, but also automatically clusters third-party flows triggered by services. Thus, AWESoME is able to manage traffic even if flows are annotated with uninformative or ambiguous hostnames.

AWESoME relies on the fact, exploited in other works [20], [27], that flows triggered by a service present temporal correlations. AWESoME extends the approach to named-flows, tunes it to operate in real-time scenarios, and integrates the algorithms into SDN, so to control the network based on complex traffic relationships.[7]

Finally, authors of [7], [8] exploit relationships between flows for traffic management. They leverage groups of flows, or *coflows*, to boost performance of MapReduce applications. Their solutions are designed for data centers and require application-level modifications, while AWESoME uses a completely in-network approach. Moreover, AWESoME aims at managing services at the edge of the network. Thus, AWESoME needs to manage a vast number of services that may behave differently from each other. Automatic identification of coflows is proposed in [29], but the solution is also limited to data centers, facing limitations if deployed at edge networks. AWESoME instead builds models for the services automatically, identifying service traffic based on the DNS.

B. Service-awareness in SDNs

SDN has become very popular from academic environments [30] to large-scale data-centers [31], sparking a host of applications, such as SDN-based routing [32] and Internet exchanges [33]. Most of the SDN applications proposed to date (see [1] for a comprehensive survey) however are a good fit to forwarding rules expressed using information from L2-L4 headers, as it is typical of popular SDN implementations.

Authors of [34] mention the lack of support for L7 applications in SDN. They make a first step towards it by solving in SDN traffic steering functions traditionally performed by middle-boxes – e.g., firewalls, proxies, intrusion detection systems etc. Like us, they advocate a solution that requires

[7] An off-line version of the algorithm used by AWESoME to learn bags of domains has appeared in [28].

no changes to SDN standards. AWESoME is a next step into bringing L7 support to SDN. It builds upon the traffic exchanged with the DNS to perform advanced traffic steering, enabling flexible managing of complex web services.

Few works have proposed low-level (stateless) forwarding rules to comprehensively manage complex services. Some authors have focused on specific services [35] or scenarios where communication patterns are well-known [36]. In contrast, AWESoME learns how generic services communicate based on network traffic and then relies on traditional L2-L4-based forwarding rules to handle the corresponding traffic.

Other works propose extensions to the SDN architecture either to delegate to switches (i.e., the data plane) management tasks that are based on L7 information, or to customize controllers and the data plane for stateful management [37], [38]. AWESoME instead is a SDN application that requires no particular changes in the lower layers of the SDN architecture.

We are aware of only few works that propose SDN applications to manage general web services [3], [4], [5], [6]. They use different methodologies to classify flows – e.g., forwarding the first N packets of each flow to controllers or implementing machine learning algorithms. These works however miss dependencies among flows, as depicted in Figure 1. Moreover, AWESoME requires to analyze only the first packet of each flow, which reduces the load at controllers.

VII. Conclusions

We introduced the concept of "per service" management with SDN. This allows the network administrators to define policies to handle all traffic exchanged by terminals when accessing complicated web services today served by multiple domains and servers.

We presented and evaluated AWESoME to enable the per service management with SDN. It leverages DNS and the concept of Bag of Domains to associate the first packet of each flow to the originating service. We showed that AWESoME is accurate and poses a marginal load on the SDN controllers and switches, thus enabling fine grained control in practice.

We believe the concept of per service management can foster new studies, e.g., to improve the classification up to make it compatible with security applications, where high accuracy is mandatory, or to develop anomaly detection based on BoDs and per service accounting.

References

[1] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, "The Cost of the "S" in HTTPS," in *Proc. of the CoNEXT*, 2014, pp. 133–140.

[3] A. Bianco, P. Giaccone, S. Kelki, N. M. Campos, S. Traverso, and T. Zhang, "On-the-fly traffic classification and control with a stateful sdn approach," in *Communications (ICC), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–6.

[4] S. Jeong, D. Lee, J. Choi, J. Li, and J. W.-K. Hong, "Application-Aware Traffic Management for OpenFlow Networks," in *Proc. of the APNOMS*, 2016, pp. 1–5.

[5] B. Ng, M. Hayes, and W. K. Seah, "Developing a Traffic Classification Platform for Enterprise Networks with SDN: Experiences & Lessons Learned," in *Proc. of the Networking*, 2015, pp. 1–9.

[6] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir, "Application-Awareness in SDN," in *Proc. of the SIGCOMM*, 2013, pp. 487–488.

[7] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, ser. HotNets-XI. New York, NY, USA: ACM, 2012, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/2390231.2390237

[8] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: ACM, 2014, pp. 443–454. [Online]. Available: http://doi.acm.org/10.1145/2619239.2626315

[9] I. Bermudez, M. Mellia, M. M. Munafò, R. Keralapura, and A. Nucci, "DNS to the Rescue: Discerning Content and Services in a Tangled Web," in *Proc. of the IMC*, 2012, pp. 413–426.

[10] P. Foremski, C. Callegari, and M. Pagano, "DNS-Class: Immediate Classification of IP Flows using DNS," *Int. J. Netw. Manag.*, vol. 24, no. 4, pp. 272–288, 2014.

[11] T. Mori, T. Inoue, A. Shimoda, K. Sato, K. Ishibashi, and S. Goto, "SFMap: Inferring Services over Encrypted Web Flows Using Dynamical Domain Name Graphs," in *Proc. of the TMA*, 2015, pp. 126–139.

[12] D. Plonka and P. Barford, "Flexible Traffic and Host Profiling via DNS Rendezvous," in *Proc. of the SATIN*, 2011, pp. 1–8.

[13] Open Networking Foundation, "OpenFlow Switch Specification - Version 1.5.0," https://www.opennetworking.org/technical-communities/areas/specification, 2014.

[14] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an elastic distributed sdn controller," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 7–12, Aug. 2013. [Online]. Available: http://doi.acm.org/10.1145/2534169.2491193

[15] Y. Hu, W. Wang, X. Gong, X. Que, and S. Cheng, "Balanceflow: Controller load balancing for openflow networks," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, vol. 02, Oct 2012, pp. 780–785.

[16] R. Kandoi and M. Antikainen, "Denial-of-service attacks in openflow sdn networks," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 1322–1326.

[17] H. Wang, L. Xu, and G. Gu, "Floodguard: A dos attack prevention extension in software-defined networks," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, pp. 239–250.

[18] M. Trevisan, I. Drago, M. Mellia, and M. M. Munafò, "Towards Web Service Classification using Addresses and DNS," in *Proc. of the TRAC*, 2016, pp. 38–43.

[19] Z. S. Harris, "Distributional structure," *Word*, vol. 10, no. 2-3, pp. 146–162, 1954.

[20] S. Kandula, R. Chandra, and D. Katabi, "What's Going on?: Learning Communication Rules in Edge Networks," in *Proc. of the SIGCOMM*, 2008, pp. 87–98.

[21] A. Callado, C. Kamienski, G. Szabó, B. P. Gero, J. Kelner, S. Fernandes, and D. Sadok, "A Survey on Internet Traffic Identification," *Commun. Surveys Tuts.*, vol. 11, no. 3, pp. 37–52, 2009.

[22] H. Kim, K. C. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee, "Internet Traffic Classification Demystified: Myths, Caveats, and the Best Practices," in *Proc. of the CoNEXT*, 2008, pp. 1–12.

[23] A. Tongaonkar, R. Torres, M. Iliofotou, R. Keralapura, and A. Nucci, "Towards Self Adaptive Network Traffic Classification," *Comput. Commun.*, vol. 56, no. 1, pp. 35–46, 2015.

[24] H. Yao, G. Ranjan, A. Tongaonkar, Y. Liao, and Z. M. Mao, "SAMPLES: Self Adaptive Mining of Persistent LExical Snippets for Classifying Mobile Application Traffic," in *Proc. of the MobiCom*, 2015, pp. 439–451.

[25] G. Xie, M. Iliofotou, T. Karagiannis, M. Faloutsos, and Y. Jin, "Resurf: Reconstructing web-surfing activity from network traffic," in *2013 IFIP Networking Conference*, May 2013, pp. 1–9.

[26] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "BLINC: Multilevel Traffic Classification in the Dark," in *Proc. of the SIGCOMM*, 2005, pp. 229–240.

[27] L. Popa, B.-G. Chun, I. Stoica, J. Chandrashekar, and N. Taft, "Macroscope: End-point Approach to Networked Application Dependency Discovery," in *Proc. of the CoNEXT*, 2009, pp. 229–240.

[28] M. Trevisan, I. Drago, M. Mellia, H. Song, and M. Baldi, "What: A big data approach for accounting of modern web services," in *IEEE Workshop on Big Data and Machine Learning in Telecom (BMLIT)*. IEEE, 2016. [Online]. Available: http://porto.polito.it/2656557/

[29] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 160–173. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934880

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[31] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, and others, "B4: Experience with a Globally-deployed Software Defined WAN," in *Proc. of the SIGCOMM*, 2013, pp. 3–14.

[32] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford, "Central Control Over Distributed Routing," in *Proc. of the SIGCOMM*, 2015, pp. 43–56.

[33] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "SDX: A Software Defined Internet Exchange," in *Proc. of the SIGCOMM*, 2014, pp. 551–562.

[34] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying Middlebox Policy Enforcement Using SDN," in *Proc. of the SIGCOMM*, 2013, pp. 27–38.

[35] Y. Wang, C. Orapinpatipat, H. H. Gharakheili, and V. Sivaraman, "TeleScope: Flow-Level Video Telemetry using SDN," in *Proc. of the EWSDN*, 2016, pp. 1–6.

[36] S. Zhao, A. Sydney, and D. Medhi, "Building Application-Aware Network Environments Using SDN for Optimizing Hadoop Applications," in *Proc. of the SIGCOMM*, 2016, pp. 583–584.

[37] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman, "Application-Aware Data Plane Processing in SDN," in *Proc. of the HotSDN*, 2014, pp. 13–18.

[38] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan, "Flow-level State Transition As a New Switch Primitive for SDN," in *Proc. of the HotSDN*, 2014, pp. 61–66.

BIOGRAPHIES

Martino Trevisan received his B.Sc. (2012) and his M.Sc. (2015) in Computer Science, both from Politecnico di Torino, Italy. He is currently a PhD student in Electrical, Electronics and Communications Engineering in the same university, where he joined the Telecommunication Networks Group (TNG). He has been collaborating in both Industry and European projects and spent six months in Telecom ParisTech, France working on High-Speed Traffic Monitoring. His research interest areas include Network Measurements and Traffic Monitoring while he is also particularly interested in leveraging Big Data and Machine Learning techniques in such fields.

Idilio Drago is an Assistant Professor (RTDa) at the Politecnico di Torino, Italy, in the Department of Electronics and Telecommunications. His research interests include Internet measurements, Big Data analysis, and network security. Drago has a PhD in computer science from the University of Twente. He was awarded an Applied Networking Research Prize in 2013 by the IETF/IRTF for his work on cloud storage traffic analysis.

Marco Mellia (S'08), Ph.D., research interests are in the in the area of traffic monitoring and analysis, in cyber monitoring, and Big Data analytics. Marco Mellia has co-authored over 250 papers published in international journals and presented in leading international conferences. He won the IRTF ANR Prize at IETF-88, and best paper award at IEEE P2P'12, ACM CoNEXT'13, IEEE ICDCS'15. He is part of the editorial board of ACM/IEEE Transactions on Networking, IEEE Transactions on Network and Service Management, and ACM Computer Communication Review. He holds a position as Associate Professor at Politecnico di Torino, Italy.

Mario Baldi is Director of Technology at Cisco Systems and Associate Professor at Politecnico di Torino. He was Data Scientist Director at Symantec Corp., Inc., Principal Member of Technical Sta at Narus, Inc., Principal Architect at Embrane, Inc.; Vice Dean of the PoliTong Sino-Italian Campus at Tongji University, Shanghai; Vice President for Protocol Architecture at Synchrodyne Networks, Inc., New York. Through his research, teaching and professional activities, Mario Baldi has built considerable knowledge and expertise in big data analytics, next generation network data analysis, internetworking, high performance switching, optical networking, quality of service, multimedia networking, trust in distributed software execution, and computer networks in general.

Han Hee Song was Lead Data Scientist at Cisco Systems. He was Principal Data Scientist at Symantec Corp. and Senior Member of Technical Sta at CTO office of Narus, Inc. His research focuses on privacy analysis of mobile users and protective measures for cyber terrorism. Dr. Song received a Ph.D. and M.A. in Computer Science from the University of Texas at Austin in 2011 and 2006, respectively. He received a B.S. from Yonsei University, Seoul, Korea in 2004.