Deep Packet Inspection as a Service

Anat Bremler-Barr
School of Computer Science
The Interdisciplinary Center
Herzliya, Israel
bremler@idc.ac.il

Yotam Harchol
School of Computer Science and Engineering
The Hebrew University
Jerusalem, Israel
yotamhc@cs.huji.ac.il

David Hay
School of Computer Science and Engineering
The Hebrew University
Jerusalem, Israel
dhay@cs.huji.ac.il

Yaron Koral
School of Computer Science and Engineering
The Hebrew University
Jerusalem, Israel
ykoral@princeton.edu

ABSTRACT

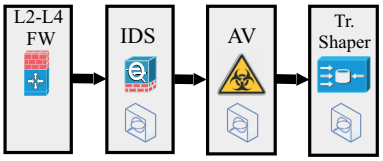
Middleboxes play a major role in contemporary networks, as forwarding packets is often not enough to meet operator demands, and other functionalities (such as security, QoS/QoE provisioning, and load balancing) are required. Traffic is usually routed through a sequence of such middleboxes, which either reside across the network or in a single, consolidated location. Although middleboxes provide a vast range of different capabilities, there are components that are shared among many of them.

A task common to almost all middleboxes that deal with L7 protocols is Deep Packet Inspection (DPI). Today, traffic is inspected from scratch by all the middleboxes on its route. In this paper, we propose to treat DPI as a service to the middleboxes, implying that traffic should be scanned only once, but against the data of all middleboxes that use the service. The DPI service then passes the scan results to the appropriate middleboxes. Having DPI as a service has significant advantages in performance, scalability, robustness, and as a catalyst for innovation in the middlebox domain. Moreover, technologies and solutions for current Software Defined Networks (SDN) (e.g., SIMPLE [41]) make it feasible to implement such a service and route traffic to and from its instances.

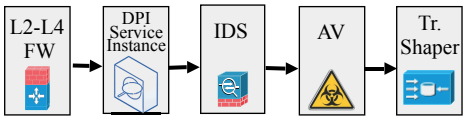
1. INTRODUCTION

Inspired by current suggestions for *Network Function Virtualization (NFV)* and the flexible routing capabilities of *Software Defined Networks (SDN)*, this paper calls for finding common tasks among middleboxes and offering these tasks as services.

Specifically, we focus on *Deep Packet Inspection (DPI)*, where the payload of packets is inspected against a set of patterns. DPI is a common task in many middleboxes. A partial list is shown in Table 1. In many of these devices, DPI is the most time-consuming task and it may take most of the middlebox processing time.[1] Thus, great effort was invested over the years in optimizing it.



(a) Without DPI service: Multiple middleboxes perform DPI on packets.



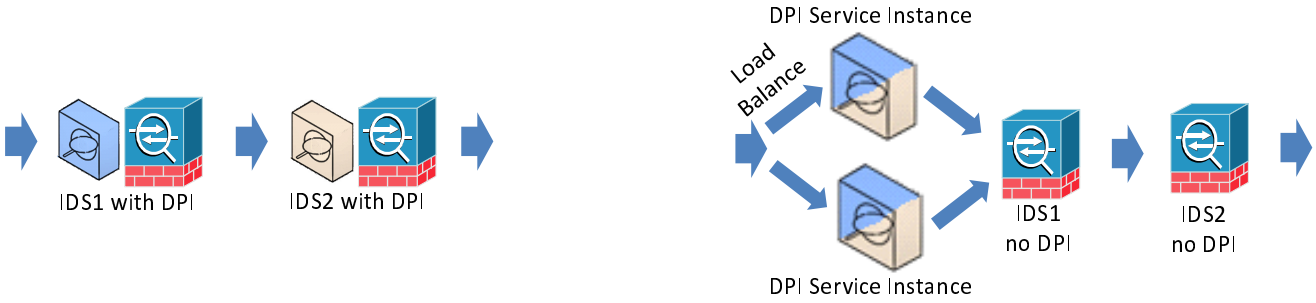
(b) With DPI service: Packets go through DPI once.

Figure 1: Examples of the chain of middleboxes (a.k.a. policy chains [41]) with and without DPI as a service.

As detailed in [41], and referenced therein, traffic nowadays goes through a chain of middleboxes before reaching its destination. This implies that traffic is scanned over and over again by middleboxes with a DPI component [39] (see Figure 1(a)). Alternatively, an opposite trend is to consolidate middleboxes in a single location (or even a hardware device) [4, 45]. However, the different components of this consolidated middlebox perform DPI separately, from scratch.

Our proposed framework extracts the DPI engine from the different middleboxes and provides it as a service for various middleboxes in the network. This service is provided by deploying one or more *service instances* around the network, all controlled by a logically-centralized *DPI Controller*. Thus, a packet in such network would go through a single DPI service instance and then

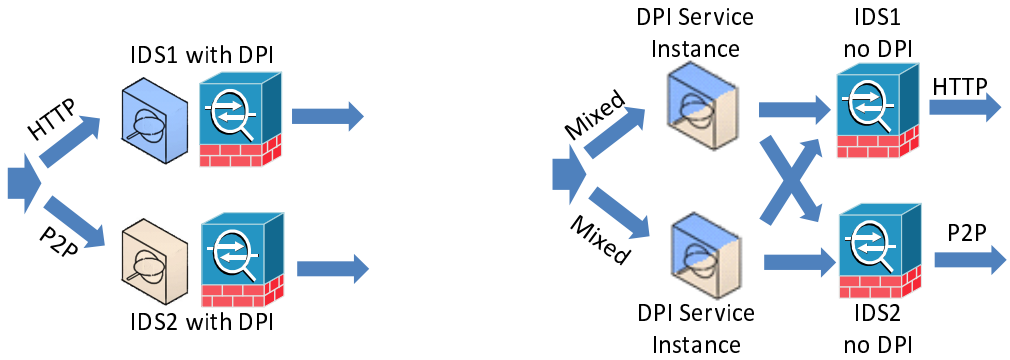
[1] In an experiment we conducted on Snort IDS [48], DPI slows packet processing by a factor of at least 2.9.



(a) Without a DPI Service

(b) With a DPI Service

Figure 2: Pipelined middlebox scenario. With DPI Service, resources are used for running multiple instances of the service while middleboxes do not have to re-scan the packets.



(a) Without a DPI Service

(b) With a DPI Service

Figure 3: An example of multiple service chains scenario. With DPI Service, flows are multiplexed to multiple DPI Service instances. This allows dynamic load balancing on the DPI engines without adding middleboxes.

Middlebox	DPI patterns	Examples
Intrusion Detection System	Malicious activity	SNORT [48], BRO [10]
AntiVirus/SPAM	Malicious activity	ClamAV [12]
L7 Firewall	Malicious activity	Linux L7-filter, ModSecurity [34]
L7 Load Balancing	Apps/URLs	F5 [37], A10 [35]
Leakage Prevention System	Leakage activity	Check Point DLP [11]
Network Analytic	Protocol IDs	Qosmos [13]
Traffic Shaper	Applications	Blue Coat PacketShapper [8]

Table 1: DPI in different types of middleboxes.

visit middleboxes according to its policy chain.[2] Its payload would not have to be scanned again as the results of the scan are provided along with the original packet (or instead without the original packet if the latter is not needed anymore).[3] Upon receiving the results from the DPI Service, each middlebox applies the rules corresponding to the matched patterns according to its internal logic. Figures 2 and 3 show two configurations for using DPI as a service. We elaborate on these scenarios and provide their corresponding experimental results in Section 6.4.

It is important to note that while fields in a packet header might be modified along the path of the packet, the payload is usually not changed, and thus, the DPI service may be used even in policy chains that contain NATs and other middleboxes that modify header fields.

Our approach is shown to provide superior performance and also reduces the memory footprint of the DPI engine data-structures. The framework also allows dynamic resource sharing as the hardware used for DPI is decoupled from the specific middlebox, as shown in Section 6.4. Since DPI is performed once, the effect of decompression or decryption, which usually takes place prior to the DPI phase, may be reduced significantly, as these heavy processes are executed only once for each packet.

[2]We assume that patterns are either not proprietary or can be disclosed to our service over a secure channel.

[3]Section 4.2 discusses this mechanism in detail.

We believe that having DPI and other shared functionalities as network services also creates necessary room for innovation in the middlebox domain, which is usually closed and proprietary. For example, when consolidating DPI to a single piece of software, one might find it beneficial to implement more advanced DPI functionalities, such as decryption and decompression, or use special hardware accelerators.

Moreover, as a central component, security devices, and specifically their DPI engines, are a preferred target for denial-of-service attacks [50]. Recent works show that DPI components within the NIDS (such as Snort [48] and Bro [10]) expose the entire system to attacks that may knock down the device [1]. Having DPI as a service is especially appealing in this case, since a developer is now focused on strengthening the DPI only at a single implementation, rather than tailoring the security solution for each middlebox that uses DPI. Furthermore, as most DPI solutions nowadays use software, one can easily deploy more and more instances of the DPI to servers across the network, and thus mitigate attacks and unintentional failures.

Our contribution in this paper is two-fold. First, we detail a framework in which DPI is deployed as a service, including an algorithm that combines DPI patterns from different sources. Second, we implemented in Mininet over OpenFlow an SDN system that enables deploying DPI as a service. This includes both the DPI service itself and the control components required for the service to work correctly. We show via experiments that our framework provides performance improvements of 67% and more.

This paper is organized as follows: In Section 2 we discuss related work. Section 3 provides the necessary background on middlebox design and DPI. Our proposed system overview is described in Section 4, and algorithmic aspects of how to perform DPI simultaneously for multiple middleboxes are discussed in Section 5. Experimental results are presented in Section 6. Finally, we conclude in Section 7.

2. RELATED WORK

2.1 Middlebox Design

Over the last few years, much effort was invested in redesigning middlebox architecture. In this section, we go over some new directions in this field, highlight the deficiencies they intended to solve, and compare them with our work.

Forwarding traffic through policy chains: In traditional networks, middleboxes are placed at strategic places along the traffic path, determined by the network topology; traffic goes through the middleboxes as dictated by the regular routing mechanism. SDN makes it possible to perform *traffic steering*, where routing through a chain of middleboxes is determined using middlebox-specific routing considerations that might differ significantly from traditional routing schemes. Our paper uses this flexibility, as was shown in [41], to route traffic to a DPI service when needed.

Virtualizing network functionalities: Recently, telecommunication vendors launched the *Network Functions Virtualization (NFV)* initiative [16] that aims to virtualize network appliances at the operator. The main objective of NFV is to reduce the operational costs of these appliances (which are traditionally implemented in middleboxes) by obtaining the same functionality in software that runs on commodity servers. NFV provides easier management and maintenance by eliminating the need to deal with multiple hardware types and vendors; moreover, as NFV is implemented in software, it promotes innovation in this domain. DPI is a significant example of an appliance or functionality that may be virtualized [16].

There are several pioneer works on middlebox virtualization. Rajagopalan et al. [44] present a mechanism to place a middlebox, such as the Bro NIDS, in a virtual environment, where the VM might migrate between different machines. Gember et al. [23] deal with standardization of unified control to middleboxes, inspired by the SDN paradigm. Nevertheless, virtualizing middleboxes raises several issues that should be carefully dealt with, such as efficient fault tolerance [26], availability [43], and management [22]. Our work mostly addresses algorithmic aspects of virtualized DPI, which are orthogonal to these works.

Middlebox consolidation and programmability: A different approach to tackle the problem raised by managing multiple middleboxes is to offer a consolidated solution [4, 14, 25, 45] consisting of a single hardware, possibly programmable, that consolidates multiple logical middleboxes. DPI as a service may complement such consolidated architectures to improve the overall performance, either by extracting the DPI engine from these boxes to a virtual service, or by using our algorithm to merge multiple DPI engines and utilize shared hardware better. Note that in such a case, several parts of our framework (such as message passing and routing between middleboxes) may be eliminated.

Outsourcing middlebox functionality: To reduce the high equipment and operating costs of middleboxes, several works proposed outsourcing middlebox functionalities [24,47] as a service [17] provided by an entity outside the network. Note that our suggestion is that the DPI, as a critical building block, will be a service for the middleboxes but would be placed in the same network.

2.2 Deep Packet Inspection

The classical algorithms for exact multiple string matching used for DPI are those of Aho-Corasick [2] and Wu-Manber [51]. For regular expression matching, two common solutions are using Deterministic Finite Automata (DFA) or Nondeterministic Finite Automata (NFA) [6, 28]. Efficient regular expression matching is still an active area of research [7, 19, 28, 29, 52].

There is extensive research on accelerating the DPI process, both in hardware [5, 15, 33] and in software [19, 28]. Most software-based solutions [19, 28] accelerate the DPI process by optimizing its underlying data structure (namely, its DFA). To the best of our knowledge, no specific design for accelerating DPI in a virtual, consolidated, or NFV environment has been proposed. The only exception is an industrial product of QOSMOS [13], which specializes in protocol classification and does not deal with the general DPI problem. Moreover, no details are disclosed on its implementation. DPI optimization and acceleration are orthogonal to this work, as they may be applied as a part of the DPI service, for further acceleration. Multicore optimization may also benefit from having DPI as a service, as instead of splitting the work between cores, it may be split among instances running over different machines.

We are not aware of any work directly performed on combining the data of several middleboxes' DPI to a single service. However, a similar concept was studied in the context of virtual IP-lookup, where trie-based data structures are considered [21, 31, 49]. This research is not applicable to DPI as the underlying algorithms are significantly different.

3. BACKGROUND

DPI lies at the core of many middlebox applications (see Table 1), and is based on *pattern matching*, in which the payload of the packet is compared against a predetermined set of *patterns* (with either strings or regular expressions).

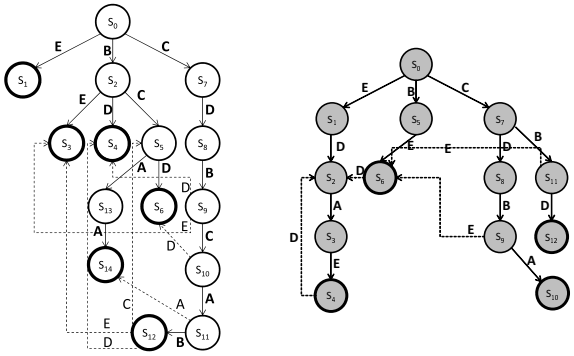


Figure 4: The Aho-Corasick Automata corresponding to the pattern sets {*E,BE,BD,BCD,BCAA,CDBCAB*} and {*EDAE,BE,CDBA,CBD*}. Solid edges are forward transitions while dashed edges are other transitions. Non-forward transitions to DFA depths 0 and 1 are omitted for brevity.

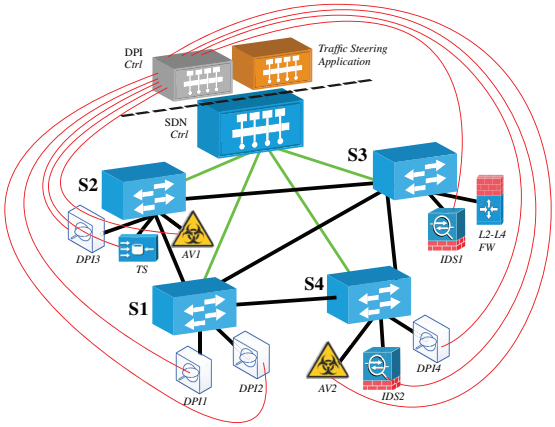
String matching is an essential building block of most contemporary DPI engines. In many implementations (such as Snort [48] and our own implementation, discussed in Section 5), even if most patterns are regular expressions, string matching is performed first (namely, as a pre-filter) and constitutes most of the work performed by the engine. Specifically, Snort extracts the strings that appeared in the regular expressions (called anchors). Then, string matching is performed over these anchors, and if all anchors originating from a specific regular expression are matched, then a regular expression matching of the corresponding expression is performed (e.g., using PCRE [38]).

This is a common procedure since regular expression engines work inefficiently on a large number of expressions. The aforementioned DFA solutions suffer from memory explosion especially when combining a few expressions into a single data structure, while the NFA solutions suffer from lower performance due to multiple active state computation for each state transition. Efficient regular expression matching is still an active area of research. Any future novel regular expression matching algorithm may easily be incorporated into our system.

The Aho-Corasick (AC) algorithm [2] is the de-facto standard for contemporary network intrusion detection systems (NIDS). It matches multiple strings simultaneously by first constructing a DFA that represents the pattern set (also known as signature set); then, with this DFA at its disposal, the algorithm scans the text in a single pass.

The DFA construction is done in two phases. First, a tree of the strings is built, where strings are added one by one from the root as chains (each node in the tree corresponds to a DFA state). When patterns share a common prefix, they also share the corresponding set of states in the tree. The edges of the first phase are called *forward transitions*. In the second phase, additional edges deal with situations where, given an input symbol b and a state s, there is no forward transition from s using b. Let the *label* of a state s, denoted by $L(s)$, be the concatenation of symbols along the path (of forward transition) from the root to s. Furthermore, let the depth of a state s be the length of the label $L(s)$. The transition from s given symbol b is to a state s', whose label $L(s')$ is the longest suffix of $L(s)b$ among all other DFA states. For example, Figure 4 depicts two DFAs that were constructed for pattern sets *{E,BE,BD,BCD,BCAA,CDBCAB}* and {*EDAE,BE,CDBA,CBD*}.

The DFA is traversed starting from the root. Traversing to an *accepting state* indicates that some patterns are a suffix of the input; one of these patterns always corresponds to the label of the accepting state.



	Policy Chain	Middlebox Chain	Physical Sequence
1.	L2L4_FW-DPI-IDS	L2L4_FW-DPI3-IDS1	S1 S3 **L2L4_FW** S3 S2 **DPI3** S2 S3 **IDS1** S3 S4
2.	DPI-IDS-AV-TS	DPI3-IDS2-AV1-TS	S1 S2 **DPI3** S2 S4 **IDS2** S4 S2 **AV1** S2 **TS** S2 S4

Figure 5: System Illustration. The DPI controller abstracts the DPI process to other network elements and controls DPI service instances across the network. Packets flow through the network as dictated by policy chains.

4. SYSTEM OVERVIEW

This section details the underlying architecture that supports DPI as a service. The main idea is to insert the DPI service in the middlebox chain prior to any middlebox that requires DPI. The DPI service scans the packet and logs all detected patterns as meta-data to the packet. As the packet is forwarded, each middlebox on its route retrieves the DPI scan results instead of performing the costly DPI task. We assume an SDN environment with a *Traffic Steering Application (TSA)* (e.g., SIMPLE [41]) that attaches policy chains to packets and routes the packets appropriately across the network. Naturally, our solution will negotiate with the TSA, so that policy chains are changed to include DPI as a service (see Figure 1).

4.1 The DPI Controller

DPI service scalability is important since DPI is considered a bottleneck for many types of middleboxes. Therefore, we envision that DPI service instances will be deployed across the network. The *DPI controller* is a logically centralized entity whose role is to manage the DPI process across the network and to communicate both with the SDN controller and the TSA to realize the appropriate data plane actions. Logically, the DPI controller resides at the SDN application layer on top of the SDN controller as in Figure 5.

Two kinds of procedures take place between the DPI Controller and the middleboxes, namely: registration and pattern set management. The first task of the DPI controller is to register middleboxes that use its service. Communication between the DPI Controller

and middleboxes is performed using JSON messages sent over a direct (possibly secure) communication channel. Specifically, a middlebox registers itself to the DPI service using a registration message . The DPI Controller address and the middlebox's unique ID and name are preconfigured (we have not deployed any bootstrap procedures at the current stage). A middlebox may inherit the pattern set of an already registered middlebox. A middlebox may state that the DPI service it requires should maintain their state across the packet boundaries of a flow, or that it operates in a read-only mode, in which it performs no actions at the packet itself and it therefore requires receiving only pattern matching results and may avoid unnecessary routing of the packet itself. An IDS is an example of a read-only middlebox as opposed to an IPS, which performs actions over the packets.

Abstractly, middleboxes operate by *rules* that contain *actions*, and *conditions* that should be satisfied to activate the actions. Some of the conditions are based on *patterns* in the packet's content. The DPI service responsibility is only to indicate appearances of *patterns*, while resolving the logic behind a condition and performing the action itself is the middlebox's responsibility. Patterns are added to and removed from the DPI controller using dedicated messages from middleboxes to the controller. The DPI Controller maintains a global pattern set with its own internal IDs. If two middleboxes register the same pattern (since each one of them has a rule that depends on this pattern), it keeps track of each of the rule IDs reported by each middlebox and associates them with its internal ID. For that reason, when a pattern removal request is received, the DPI Controller removes the middlebox reference to the corresponding pattern. Only if there are no other middleboxes referrals to that pattern, is it removed.

One concern is the traffic incurred by transmitting the pattern sets. However, as opposed to DPI DFAs, which are large, the pattern sets themselves are compact: Recent versions of pattern sets such as Bro or L7-Filter are 12KB and 14KB, respectively. Larger pattern sets such as Snort or ClamAV are 2MB and 5MB, respectively. Still, if the patterns are compressed, their size is no more than two megabytes (55KB and 2MB, respectively). The construction of the data structure that represents the patterns is the responsibility of the DPI instance, and therefore does not involve communication over the network.

The DPI controller also receives from the TSA the relevant policy chains (namely, all the sequences of middlebox types a packet should traverse). It assigns each policy chain a unique identifier that is used later by the DPI service instances to indicate which pattern matching should be performed. Usually, the TSA pushes some VLAN or MPLS tag in front of the packet to easily steer it over the network ([41]). DPI service instances can then read these tags in order to identify the set of patterns a packet should be matched against. In case this tag is not available, the DPI controller can push such a tag, for example using an OpenFlow directive.

Finally, the DPI controller is also responsible for initializing DPI service instances (see Section 5.1), deployment of different DPI service instances across the network (see Section 4.3), and advance features that require a network-wide view (e.g., as described in Section 4.3.1).

4.2 Passing Pattern Matching Results

Passing the pattern matches results to the middleboxes should take into account the following three considerations: First, it should be oblivious to the switches and not interfere with forwarding the packet through the chain of middleboxes and then to its destination. Second, the meta-data is of a variable size as the number of matches varies and is not known in advance. Third, the process should be oblivious to the middleboxes (and hosts) that are not aware of the DPI service. Having these considerations in mind, we suggest three solutions that may suit different network conditions:

- Adding match result information as an additional layer of information prior to the packet's payload. This allows maximal flexibility and the best performance. Publicly available frameworks such as Network Service Header (NSH) [42] and Cisco's vPath [46] may be used to encapsulate match data, also in an SDN setting [27]. Several commercial vendors support this method in service chain scenarios (e.g. Qosmos [13]). The downside of this approach is that middleboxes that refer to the payload on the service chain should be aware of this additional layer of information. However, if all middleboxes that use the DPI service are grouped and placed right after the DPI service instance in the service chain, the last middlebox can simply remove this layer and forward the original packet.

- An option that does not require reordering of service chains relies on using some flexible pushing and pulling of tags (e.g., MPLS labels, VLAN tags, PBB tags). This method is supported in current OpenFlow-based SDN networks [20]. A similar alternative is to use the FlowTags mechanism [18]. The downside of the tagging option is that it might be messy as each matching result may require several such tags, which in turn must not collide with other tags used in the system.

- When middleboxes on the service chain are all in *read-only* mode, where the middlebox requires only the DPI results rather than the packet itself, it may be appealing to send only the match results using a dedicated packet without the packet itself. As most packets do not contain matches at all, this option may dramatically reduce traffic load over the middlebox service chain. For example, in Big Switch Networks' Big Tap [36] fabric, the traffic is tapped from production networks to a separate monitoring network, where monitoring is done while the original packet is forwarded at the production network regardless of the monitoring results.

In all three options one may use a single bit in the header to mark whether patterns were matched. Specifically, a packet with no matches is always forwarded as is without any modification.

As our experimental environment is based on Mininet over OpenFlow 1.0, which supports neither NSH nor MPLS, in our implementation we passed the matching results using dedicated packets.

4.3 Deployment of DPI Service Instances

The DPI controller abstracts the DPI service for the TSA, SDN controller, and the middleboxes. Hence, one of its most important tasks is to deploy the DPI instances across the network. There might be many considerations for such deployment and in this section we discuss only a few.

First, we emphasize that not all DPI instances need to be the same. Thus, a common deployment choice is to group together similar policy chains and to deploy instances that support only one group and not all the policy chains in the system. The DPI controller will then instruct the TSA to send the traffic to the right instance. Alternatively, one might group the middlebox types by the traffic they inspect. For example, sets of patterns that correspond to HTTP traffic may be allocated to some DPI service instances, while a set of patterns that corresponds to FTP is allocated to other DPI service instances.

Additionally, the DPI controller should manage the DPI instance resources, so that an instance is not overwhelmed by traffic, and

therefore, performs poorly. Thus, the DPI controller should collect performance metrics from the working DPI instances and may decide to allocate more instances, to remove service instances, or to migrate flows between instances. This should be done exactly in the same manner as suggested in [44]. Notice that, in general, performing operations on the DPI service instances rather than the middleboxes themselves is easier as most of the flow's state is typically kept within the middlebox, while the DPI instance keeps only the current DFA state and an offset within the packet.[4] Finally, we note that allocation, de-allocation, and migration affect the way packets are forwarded in the network. Thus, the DPI controller should collaborate with the TSA (and the SDN controller) to realize the changes and take into account other network considerations (such as bandwidth and delay).

The ability to dynamically control the DPI service instances and to scale out provides the DPI controller great flexibility, which can be used for powerful operations. Section 4.3.1 shows how this ability is used to enhance the robustness of the DPI service and its performance.

4.3.1 Enhancing Robustness and Security

DPI engines, as a core building block of many security appliances, are known to be the target of attacks [1, 32]. A recently-suggested architecture, called MCA^2 [1], mitigates such attacks by deploying several copies of DPI engines over multiple cores of the same machine. The key operation of MCA^2 is to detect and isolate the *heavy* packets that cause the degraded performance, and divert them to a dedicated set of cores. Moreover, the dedicated cores may run a different AC implementation (other than the full-table AC described in Section 3) that is more suitable for handling this kind of traffic (see [1, 9] for more details).

MCA^2 can be implemented as-is in each DPI service instance, provided it runs on a multi-core machine. In addition, our architecture may implement MCA^2, while scaling out to many DPI service instances. As in the original MCA^2 design, each DPI service instance should perform ongoing monitoring and export telemetries that might indicate attack attempts. In the MCA^2 design, these telemetries are sent to a central *stress monitor* entity. Here, the *DPI controller*, described in Section 4.1, takes over this role. This is illustrated in Figure 6: Under normal traffic, all DPI service instances work regularly. Whenever the *DPI controller* detects an attack on one of the instances, it sets some of the instances as 'dedicated', and *migrates* the heavy flows, which are suspected to be malicious, to those dedicated DPI instances (these instances might also use a different DPI algorithm that is tailored for heavy traffic). Flow migration is performed as described in Section 4.3, and requires close cooperation with the traffic steering application. Moreover, dedicated DPI instances can be dynamically allocated as an attack becomes more intense, or deallocated as its significance decreases.

5. DPI SERVICE INSTANCE IMPLEMENTATION

This section describes the implementation of a DPI service instance. At the core of the implementation, we present a *virtual DPI algorithm* that handles multiple pattern sets. We first focus on *string matching* and then extend it to handle *regular expressions*.

5.1 Initialization

We first show how to combine multiple pattern sets, originating from different middleboxes such that each packet is scanned only once.

[4]Notice that flow migration might require some packet buffering at the source instance, until the process is completed.

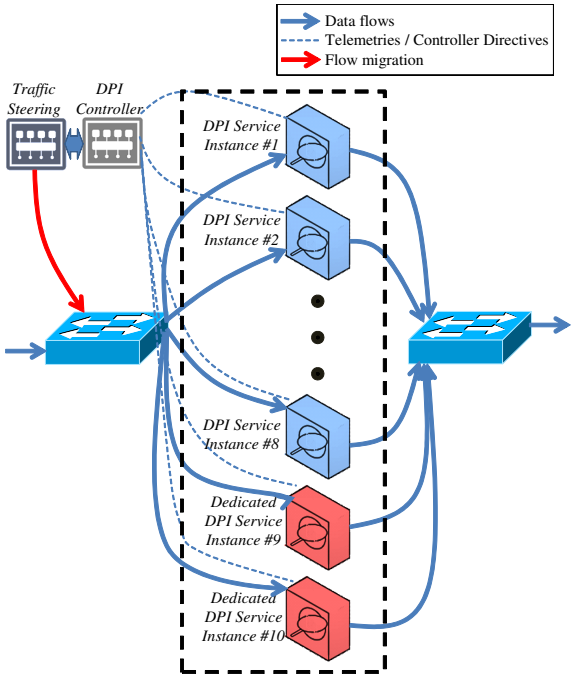


Figure 6: MCA^2 system design for virtual DPI environment.

Each middlebox type has a unique identifier and it registers its own pattern set with the DPI controller (see details in Section 4). As the DPI controller is a logically-centralized entity that allocates the identifiers, we may assume identifiers are sequential numbers in $\{1,\ldots,n\}$, where n is the number of middlebox types registered to the DPI service. Let P_i be the pattern set of middlebox type i.

Upon instantiation, the DPI controller passes to the DPI instance the pattern sets and the corresponding middlebox identifiers. Along with these sets, the DPI controller may pass additional information, such as a stopping condition for each middlebox (namely, how deep into L7 payload the DPI instance should look[5]), or whether the middlebox is stateless (scans each packet separately) or stateful (considers the entire flow, and therefore, should carry the state of the scan between successive packets). Moreover, the DPI controller passes the mapping between policy chain identifiers and the corresponding middlebox identifiers in the chain.

Our simple algorithm works in two steps. First, we construct the AC automaton as if the pattern set was $\bigcup_i P_i$. Note that the number of accepting states in the resulting DFA, denoted by f, is $|\bigcup_i P_i|$, as there is an accepting state for each pattern, no matter whether it originates in one or more middlebox. Further note that the state identifier in the DFA is meaningless; we use this degree of freedom and map the identifiers of the accepting states to the range $\{0,\ldots,f\}$; this will make the resolution stage for matched patterns more efficient in terms of time and space.

The second step is to determine, for each accepting state, which middleboxes have registered the pattern and what the identifier of the pattern is within the middlebox pattern set. This is done by storing a pre-computed direct-access array `match` of f entries such

[5]The stopping condition is useful, for example, when middleboxes only care about specific application-layer headers with a fixed or bounded length.

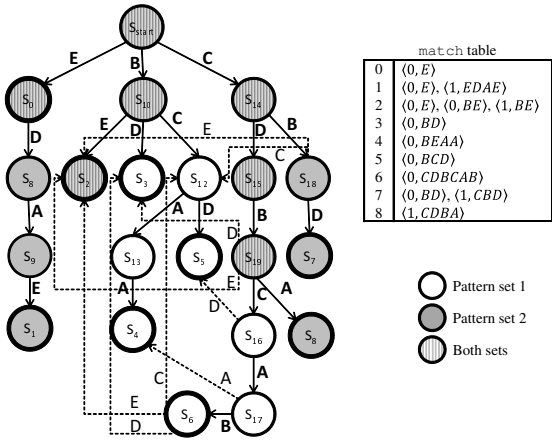


Figure 7: The DFA and `match` table for $P_0 = \{E, BE, BD, BCD, BCAA, CDBCAB\}$ and $P_1 = \{EDAE, BE, CDBA, CBD\}$ (as in Figure 4). States common to both sets are marked in gray. White accepting states are marked with bitmap 10, gray accepting states with bitmap 01, and striped accepting states with bitmap 11. Non-forward transitions to s_{start}, s_0, s_9, and s_{13} are omitted for brevity.

that its i^{th} entry holds the information corresponding to the pattern of accepting state i, as a sorted list of ⟨middlebox id, pattern id⟩ pairs. It is important to note that if we have a pattern i (e.g., DEF) that is a suffix of another pattern j (e.g., ABCDEF), we should add all the pairs corresponding to pattern i also to the j^{th} entry.

Furthermore, traditional DFA implementations mark accepting state using one bit; when n is relatively small, it is beneficial to mark the accepting state by a bitmap of the middlebox identifiers in its corresponding list; in such a case, a simple bitwise-AND operation can indicate if we need to check the table at all, or may continue scanning the packets, since the matching pattern is not relevant to the packet. In our implementation, it is also possible to check whether the state ID is less than a predefined constant whose value is the number of accepting states in the automaton.

An example of the resulting DFA and the `match` table is depicted in Figure 7.

We also store in a separate table the mapping between a policy chain identifier and the corresponding middlebox identifiers. Another table holds the mapping between a middlebox identifier and its properties (namely, its stopping condition and whether it is stateless or stateful). Finally, if at least one of the middleboxes is stateful, we will initialize an empty data structure of active flows,[6] which will hold the state and offset of scans done on that flow up until now.

5.2 Packet Inspection

Packets should be compared with several (but, in general, not all) pattern sets. The relevant identifiers for pattern set selection are marked at the packet header (as explained in Section 4.1), and the DPI service uses them to determine which pattern sets apply to each packet. Hence, upon packet arrival, the DPI service first resolves (using the mapping saved in the initialization phase) what the relevant middlebox identifiers are (we shall call them the *active* middleboxes for the packet). Moreover, the stopping condition for the packet is determined as the most conservative condition among all active middleboxes and an empty match-list for each active middlebox is initialized, as well as a global counter variable `cnt` (which counts the number of bytes scanned so far). When n is sufficiently small, a bitmap of size n is constructed such that the i^{th} bit is set if and only if middlebox i is active.

Finally, if the packet is part of a flow that has already been scanned, its DFA state is restored. The offset of the packet within the flow is stored in another variable, called `offset` (in case of stateless scan, `offset=0`).

Then, the packet is scanned against the combined DFA, while maintaining the value of `cnt`. When reaching an accepting state j, the bitmap of the packet is compared against the bitmap stored at the state; if a match is found then all pattern identifiers corresponding to active middleboxes in `match[j]` are added to the corresponding match-lists, along with the value of `cnt`. In the end of the scan, irrelevant matches are deleted from the match-lists: For stateful active middleboxes, a match is deleted if the value of `cnt+offset` exceeds the stopping condition of the specific middlebox. For stateless middleboxes, in which the packet scan should have started at s_{start} but instead started at the restored state for the stateful middleboxes, we delete patterns whose length is smaller than their value of `cnt`, [7] as well as patterns whose stopping condition is smaller than the value of `cnt`.

After the packet scan is finished, the match-lists are sent to the corresponding active middleboxes as described in Section 4; along with the pattern identifier, we pass the value of either `cnt` (for stateless middleboxes) or `cnt+offset` (for stateful middleboxes). If at least one active middlebox is stateful, the state of the DFA in the end of the scan is recorded and `offset` is incremented by `cnt`.

5.3 Dealing with Regular Expressions

As explained in Section 3, we take an approach similar to the one implemented in Snort NIDS and use a string matching process as pre-filtering for regular expression matching. Specifically, for each regular expression, we first extract sufficiently long strings (which we call *anchors*) from each regular expression. These anchors must be matched for the entire regular expression to be matched. Short strings of length less than 4 characters are not extracted. For example, in the regular expression "`regular\s*expression\s*\d+`", the anchors "`regular`" and "`expression`" are extracted. We add the anchors extracted from the regular expressions of middlebox i to pattern set P_i. In addition, we hold a mapping between the regular expression and its anchors. The packet is scanned as before (with the DFA obtained by the new pattern set). Upon completion, we check if there are regular expressions of an active middlebox for which all anchors were found. If there are, an off-the-shelf regular expression engine (e.g., PCRE [38]) is invoked on these regular expressions (one by one). Otherwise, no further operation is needed. Note that ideally, if a regular expression can be matched by only relying on its anchors, it is recommended to add it as an exact string matching rule rather than as a regular expression. In Snort, for example, 57% of the rules are regular expression rules, 99.7% of which invoke PCRE only if all of its underlying anchors were matched. The remaining 0.3% rules could have been written as exact string matching rules at the first place.

Finally, we note that sometimes there are middleboxes whose regular expressions contain almost no anchors (or, alternatively, very short anchors). In such a case, we use a regular expression

[6] The set of active flows is maintained in the same manner as for stateful middleboxes today, and is outside the scope of this paper.

[7] This implies that the pattern has started in the previous packet, and therefore should be ignored in a stateless DPI

matching algorithm (e.g., as suggested in [28] and references therein), and run it in parallel to our string matching algorithm.

6. EXPERIMENTAL RESULTS

In this section we evaluate the feasibility and performance of our virtual DPI algorithm.

6.1 Implementation

We implemented a complete system with a simple controller and tested it inside a Mininet [30] virtual machine. The code is available at `https://github.com/DeepnessLab/moly`. Our basic experimental topology includes two user hosts, two middlebox hosts, and a DPI service instance host that can utilize multiple cores. All hosts are connected through a single switch and the TSA, implemented as a POX [40] module, steering traffic from one user host to the other according to the defined policy chains.

Our experimental DPI service instance receives match rules in JSON format and builds an AC DFA as described in Section 5. If a packet matches one or more rules, the DPI service instance marks it so that middleboxes will know it has matches (we use the IP ECN field for this purpose) and constructs a result packet that is sent right after the marked data packet.

We decided to send match information in our experiments as a separate packet since POX only implements OpenFlow version 1.0, which does not support the other methods suggested in Section 4.2. We also find it easier to debug and trace.

In addition, we implemented a sample virtual middlebox application that receives traffic from the DPI service instance and if necessary, buffers packets until their corresponding results or data packet arrives. This sample implementation only counts the total number of rules that were reported to it.

We compare our design to a system where middleboxes perform DPI and counting of rules (and possibly much more). To do so, we also implement an application that does both and use it as a baseline for comparison.

In the URL above we also provide a prototype implementation for a Snort 2.9.6.2 plugin that parses DPI Service results instead of scanning the packets using Snort's traditional pattern matching engines. The plugin itself requires less than 100 lines of code and in order to use it, *only six lines of code were added* to existing Snort code files, in addition to our plugin files. A similar plugin can be written for other software middleboxes with the appropriate adaptations for the internals of each middlebox's DPI module.

6.2 Experimental Environment

Experiments were performed on a machine with Intel Sandybridge Core i7 2600 CPU, quad-core, each core having two hardware threads, 32 KB L1 data cache (per core), 256 KB L2 cache (per core), and 8 MB L3 cache (shared among cores). The system runs Linux Ubuntu 11.10, on which we run one or more instances of a Linux Ubuntu virtual machine using VMWare Player. We use exact-match patterns of length eight characters or more from Snort [48] (up to 4,356 patterns) and Clam-AV [12] (31,827 patterns). As input traffic we used two traces: a campus wireless network tapped trace of about 9GB, and a 170MB HTTP traffic trace crawled from most popular websites [3][8]. Our code provided comparable results on both traces and thus, unless specifically noted, results shown are of the second trace.

Note that we did not use Mininet for performance testing as it incurs major overheads and we found the variance between results to be too high. We verified our solution in Mininet and then tested each component by feeding it with the appropriate traffic and measuring its performance.

[8]This trace contains various website data such as HTML, JavaScript, images, etc. In some cases we repeated the trace to create a longer experiment.

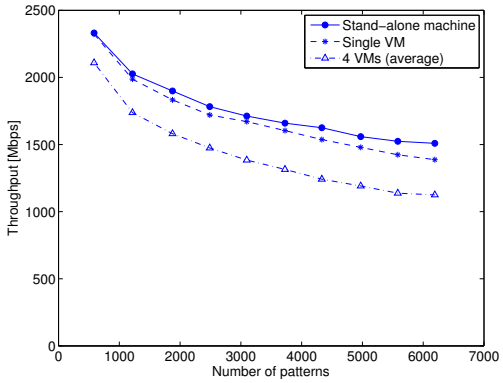


Figure 8: The effect of virtualization and number of patterns on the throughput of the AC algorithm.

Sets	Patterns	Space	Throughput
Snort1	2169	36.73 MB	981 Mbps
Snort2	2187	37.69 MB	931 Mbps
Snort1+Snort2	**4356**	**71.18 MB**	**768 Mbps**

Table 2: Comparing the performance of two middleboxes, one running on pattern sets of *Snort1* and the other on pattern sets of *Snort2*, to one virtual DPI instance with the combined pattern sets of *Snort1* and *Snort2*.

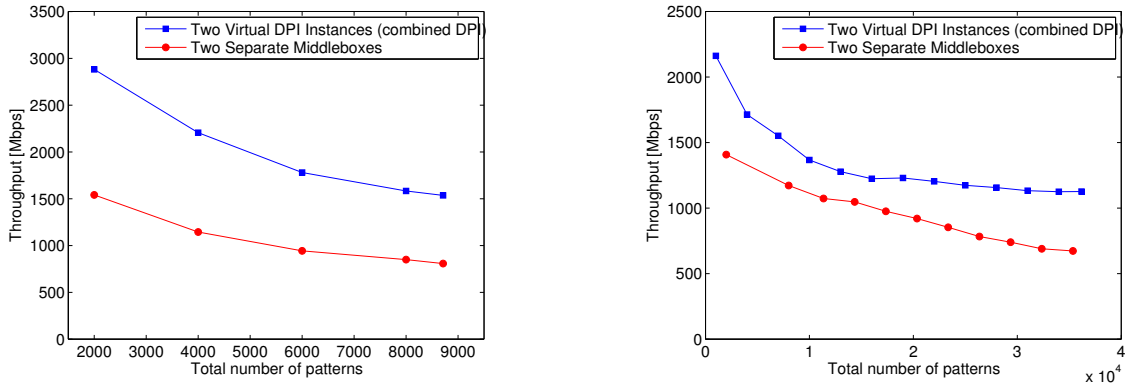
6.3 Virtual DPI Performance

As a first step, we evaluate the impact of the virtualization environment on DPI in order to ensure that DPI is suitable to run as a VM. This test is done on the original AC algorithm (and not our virtual DPI algorithm). We run three different scenarios: first, when the DPI runs on a stand-alone machine; second, when the DPI runs on a VM while the other cores are idle; third, when four instances of the DPI are running, each of them on a separate VM that uses a separate core (such that they occupy all cores of the machine), and throughput is calculated as the average throughput of the four cores. The tests were done for different numbers of patterns. Figure 8 shows that virtualization has a minor impact on DFA throughput. The number of patterns has a major impact. From here on, we will focus on running our virtual DPI algorithm as an instance, where the instance runs on a VM in all our experiments.

In addition to the virtual DPI instance, we always have one or more instances of a virtual middlebox application that receives the results from the DPI instances, counts and reports them. In our experiments, these applications operate much faster than the virtual DPI instances and thus are not a bottleneck in the system. For this reason, the overhead of buffering and reordering packets in these applications do not impose significant delays or any throughput degradation.

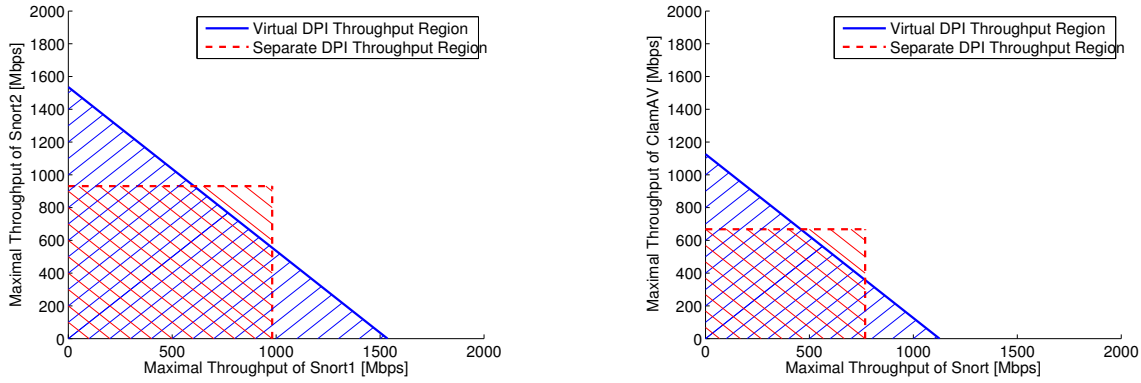
6.4 Comparison to Different Middlebox Configurations

Next, we check and evaluate the benefit of our algorithm. In order to evaluate the savings of our mechanism, we took the patterns of Snort and randomly divided them into two sets, *Snort1*



(a) *Snort1* and *Snort2* (b) Full Snort and ClamAV

Figure 9: Comparing the throughput that can be handled by two pipelined middleboxes, and by our Virtual DPI.



(a) *Snort1* and *Snort2* (b) Full Snort and ClamAV

Figure 10: Actual achievable throughput for two separate middleboxes that handle different traffic (see red, dashed rectangle), compared to the theoretical achievable throughput of our combined instances of virtual DPI (see blue, solid triangle).

and *Snort2*, simulating a configuration where we have two standalone middleboxes, *Snort1* and *Snort2*. Table 2 shows the space requirement and throughput of each of the middleboxes when using a regular DPI process, compared with a single virtual DPI that runs the DPI for the combined set of patterns *Snort1* and *Snort2*. The throughput of the combined machine is just 12% less than that of each separate machine. As we previously showed, this is mainly due to the impact of the number of patterns.

To understand the gain from the virtual DPI, we simulate two scenarios: in the first scenario, shown in Figure 2, traffic should go through a pipeline of two middleboxes, one with pattern set A and the other with pattern set B (for example, *Snort1* and *Snort2*, or full Snort and ClamAV). In the second scenario, shown in Figure 3 there are two service chains, for example for two types of traffic: one should be handled by a middlebox with pattern set A and the other by a middlebox with pattern set B. In both cases we compare the naïve solution of two instances, where each instance runs the DPI with different sets of patterns (A or B), to the case of using two instances of our virtual DPI solution.

Figure 9 shows the throughput in the first scenario and compares it to a setup of two virtual DPI instances that run on both machines simultaneously, where the load is equally distributed between them. It is clear that our virtual DPI solution is at least 86% faster in the first case, and *more than* 67% *faster* in the second case.

Figure 10 evaluates the savings in the second scenario. The dashed rectangle is the actual achieved throughput region of traffic that the naïve DPI solution can handle, given that each pattern set is handled by a single middlebox. The triangle is the theoretical achievable throughput region that our virtual DPI solution can handle, given that both machines run our virtual DPI, based on the actual achieved throughput in our experiments.

Consider two such middleboxes as appear in Figure 10(a). The motivation to use virtual DPI in this scenario is that most of the time not all middleboxes handle full load, and thus combined virtual DPI machines could make use of free resources from one middlebox to provide higher capacity for another middlebox. This can be seen in the figure as the areas inside the triangle but outside the rectangle. For example, in Figure 10(b), if Snort is under-utilized and Clam-AV faces high load, Clam-AV could actually exceed 100% of its original capacity without adding more resources (see blue triangular area above the red rectangle), and the same is true for the opposite situation where Clam-AV is under-utilized and Snort is over-utilized.

Since the DPI is now a service, additional throughput can be gained by deploying additional virtual DPI instances. This can be

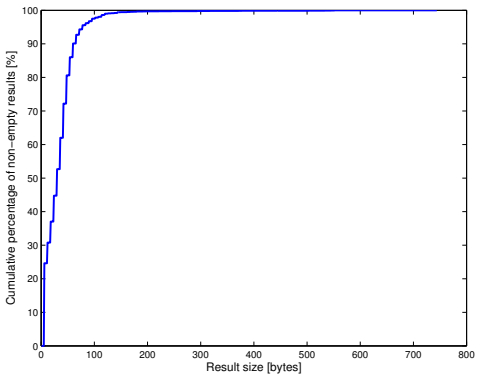


Figure 11: Cumulative distribution of non-empty match report size per packet.

done dynamically as it only requires starting another virtual machine and steering some of the traffic to it. Moreover, DPI service instances can be quickly migrated to specific points in the network to handle transient loads in specific areas.

6.5 Analysis of Match Report Size

A single match can be reported with up to 4 bytes. Occasionally, when a pattern consists of the same character one or more times, and this character appears in a packet multiple times sequentially, multiple matches of the same pattern (or set of patterns) should be reported. For these cases we also allow reporting ranges of matches, with a given starting position and length. Such ranges can be reported with up to 6 bytes.

Figure 11 shows the cumulative distribution of non-empty match reports in the campus network trace, when using 6 bytes per match report (to allow faster encoding and decoding of both regular and range reports). Note that in both traces we used, more than 90% of the packets have no matches, but the figure refers only to packets that do have matches. The average report size is 34 bytes, while most of the packets are smaller than that, and only 1% of the reports are above 120 bytes.

7. CONCLUSIONS

Middleboxes and monitoring tools have been known as closed, expensive and hard-to-manage boxes, though very widely deployed in all kinds of networks due to their important roles.

Virtualization, NFV, and SDN promise a revolution in the way middleboxes and monitoring tools are designed and managed. Many monitoring applications share a wide range of common tasks. We believe that these tasks should be provided as services, managed by a logically-centralized control, to allow enhanced performance, lower costs, flexible and scalable design, and easy innovation for monitoring applications.

This paper focuses on Deep Packet Inspection, which is one of the heaviest tasks of contemporary middleboxes. We design a framework for making DPI a service: each packet that requires a DPI by any of the middleboxes on its policy chain is forwarded to the DPI service, where it is inspected only once. Then, the inspection results (namely, the patterns that were matched) are communicated to the corresponding middleboxes, either on the same packet (e.g., using NSH) or on a different packet. Our framework relies heavily on virtualization and therefore includes both a *virtual* DPI service, which is instantiated across the network, and a DPI controller, whose role is to orchestrate the different DPI service instances. Making DPI a service has implications not only for the architecture and the system design of a middlebox that uses DPI, but also for the algorithmic aspects of the DPI engine (which is implemented by the virtual DPI service) itself. Specifically, this work presents one such tailor-made algorithm that benefits from the flexibility of a virtual environment.

Finally, we provide an SDN implementation for our framework and experiments to show that these ideas are realized into performance improvements and much more flexible and scalable design. In future work, we plan to investigate the possibility of also turning other common tasks, such as flow tagging and session reconstruction, into services.

Acknowledgements

This work was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement no 259085, and by the Chief Scientist Office of the Israeli Ministry of Industry, Trade and Labor within the "Neptune" consortium.

8. REFERENCES

[1] Yehuda Afek, Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. MCA^2: multi-core architecture for mitigating complexity attacks. In *ANCS*, pages 235–246, 2012.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. of the ACM*, 18(6):333–340, 1975.

[3] Alexa: The web information company, 2013. `http://www.alexa.com/topsites`.

[4] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: extensible open middleboxes with commodity servers. In *ANCS*, pages 49–60, 2012.

[5] Zachary K. Baker and Viktor K. Prasanna. Time and area efficient pattern matching on FPGAs. In *FPGA*, pages 223–232, 2004.

[6] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *CoNEXT*, page 1, 2007.

[7] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *ANCS*, pages 145–154, 2007.

[8] Blue coat packet shapper. `http://www.bluecoat.com/products/packetshaper`.

[9] Anat Bremler-Barr, Yotam Harchol, and David Hay. Space-time tradeoffs in software-based deep packet inspection. In *HPSR*, pages 1–8, 2011.

[10] The Bro Network Security Monitor. `http://bro-ids.org`.

[11] CheckPoint. Check Point DLP software blade. `http://www.checkpoint.com/products/dlp-software-blade/`.

[12] Clam AntiVirus. `http://www.clamav.net`.

[13] Intel Corp. Service-aware network architecture based on SDN, NFV, and network intelligence, 2014. `http://www.qosmos.com/wp-content/uploads/2014/01/Intel_Qosmos_SDN_NFV_329290-002US-secured.pdf`.

[14] Crossbeam. Virtualized security: The next generation of consolidation, 2012. `http://www.computerlinks.ch/FMS/14322.virtualized_security_en_.pdf`.

[15] Sarang Dharmapurikar, John Lockwood, and Member Ieee. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 24:2006, 2006.

[16] ETSI. Network functions virtualisation - introductory white paper, 2012. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[17] Seyed Kaveh Fayazbakhsh, Michael K Reiter, and Vyas Sekar. Verifiable network function outsourcing: requirements, challenges, and roadmap. In *HotMiddlebox*, pages 25–30, 2013.

[18] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. FlowTags: enforcing network-wide policies in the presence of dynamic middlebox actions. In *HotSDN*, pages 19–24, 2013.

[19] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved DFA for fast regular expression matching. *Computer Communication Review*, 38(5):29–40, 2008.

[20] Open Networking Foundation. Openflow switch specification version 1.4.0, October 2013. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf.

[21] Jing Fu and Jennifer Rexford. Efficient IP-address lookup with a shared forwarding table for multiple virtual routers. In *CoNEXT*, page 21, 2008.

[22] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *CoRR*, abs/1305.0209, 2013.

[23] Aaron Gember, Prathmesh Prabhu, Zainab Ghadiyali, and Aditya Akella. Toward software-defined middlebox networking. In *HotNets*, pages 7–12, 2012.

[24] Glen Gibb, Hongyi Zeng, and Nick Mckeown. Outsourcing network functionality. In *HotSDN*, pages 73–78, 2012.

[25] Lukas Kekely, Viktor Pus, and Jan Korenek. Software defined monitoring of application protocols. In *INFOCOM*, pages 1725–1733, 2014.

[26] Junaid Khalid and Josh Slauson. Fault tolerant middleboxes. Technical report, University of Wisconsin - Madison, 2012.

[27] Pritesh Kothari. Network Service Header support for OVS. OVS Code Patch, September 2013. http://openvswitch.org/pipermail/dev/2013-September/032036.html.

[28] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM*, pages 339–350, 2006.

[29] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *ANCS*, pages 81–92, 2006.

[30] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Hotnets-IX*, pages 19:1–19:6, 2010.

[31] Hoang Le, Thilan Ganegedara, and Viktor K Prasanna. Memory-efficient and scalable virtual routers using FPGA. In *FPGA*, pages 257–266, 2011.

[32] Wenke Lee, João B. D. Cabrera, Ashley Thomas, Niranjan Balwalli, Sunmeet Saluja, and Yi Zhang. Performance adaptation in real-time intrusion detection systems. In *RAID*, pages 252–273, 2002.

[33] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *USENIX Security*, pages 8–8, 2010.

[34] ModSecurity. http://www.modsecurity.org.

[35] A10 Networks. aFleX advanced scripting for layer 4-7 traffic management. http://www.a10networks.com/products/axseries-aflex_advanced_scripting.php.

[36] Big Switch Networks. Big tap monitoring fabric, 2014. http://www.bigswitch.com/products/big-tap-monitoring-fabric.

[37] F5 Networks. Local traffic manager. https://f5.com/products/modules/local-traffic-manager.

[38] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.

[39] Personal communication with several networking and security companies.

[40] Pox controller, 2013. http://www.noxrepo.org/pox/about-pox.

[41] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM*, pages 27–38, 2013.

[42] Paul Quinn, Puneet Agarwal, Rajeev Manur, Rex Fernando, Jim Guichard, Surendra Kumar, Abhishek Chauhan, Michael Smith, Navindra Yadav, and Brad McConnell. Network service header. IETF Internet-Draft, February 2014. https://datatracker.ietf.org/doc/draft-quinn-sfc-nsh.

[43] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico replication: a high availability framework for middleboxes. In *SoCC*, pages 1:1–1:15, 2013.

[44] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*, pages 227–240, 2013.

[45] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *NSDI*, pages 24–38, 2012.

[46] Nisarg Shah. Cisco vPath technology enabling best-in-class cloud network services, August 2013. http://blogs.cisco.com/datacenter/cisco-vpath-technology-enabling-best-in-class-cloud-network-services/.

[47] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *SIGCOMM*, pages 13–24, 2012.

[48] Snort. http://www.snort.org.

[49] Haoyu Song, Murali Kodialam, Fang Hao, and TV Lakshman. Building scalable virtual routers with trie braiding. In *INFOCOM*, pages 1–9, 2010.

[50] Sony Ericsson Latest Victim of SQL Injection Attack, 2011. http://www.eweek.com/c/a/Security/Sony-Data-Breach-Was-Camouflaged-by-Anonymous-DDoS-Attack-807651.

[51] Sun Wu and Udi Manber. A fast algorithm for multi-pattern

searching. Technical report, Chung-Cheng University, University of Arizona, 1994.

[52] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS*, pages 93–102, 2006.