SCIENCE CHINA
Information Sciences

• **Supplementary File** •

Efficient Middlebox Scaling for Virtualized Intrusion Prevention Systems in Software-Defined Networks

Junchi XING[1], Chunming WU[2,1*], Haifeng ZHOU[2,1], Qiumei CHENG[1], Danrui YU[1] & Mayra MACAS[1]

[1]*College of Computer Science, Zhejiang University, Hangzhou* 310027*, China;*
[2]*Zhejiang Lab, Hangzhou* 311100*, China*

The supplementary file has the following organization. Appendix A states the problem of this paper. Appendix B provides the design of the Middlebox Scaler module of ESBox. Appendix C describe the design of the Flow Distributor module of ESBox. Appendix D evaluates the performance of ESBox. And Appendix E discusses the related work.

Appendix A Problem Statement

Software-based middlebox (e.g. virtual machine (VM), container) transforms the implementation of network functions and has the capability of being scaled (horizontally or vertically) to make it adjust to the input, variable workload. Therefore, it can be used as the agent node of VIPS for handling traffic volumes that vary frequently and significantly [4,5]. Moreover, the SDN network paradigm can help to easily split and redistribute traffic to the scaled middleboxes rather than using traditional hardware (e.g. route) [7,15]. However, given these benefits, a problem arises: how to achieve an efficient scaling of middleboxes for VIPS, which is studied in this paper. An efficient scaling consists of two factors as follows.

First, the middleboxes should be synchronously scaled according to traffic volumes and rigorous IPS performance requirements, with high resource usage. It is important to elaborately decide the number of middleboxes to be scaled out/in to make the middleboxes just meet the current traffic volumes IPS performance requirements, simultaneously. A careless scaling may provision too many middleboxes that run at low load or even idly. This leads to low resource usage, which conflicts with the important goal of resource optimization [16]. Otherwise, a careless scaling may under-provision middleboxes, and traffic will congest the middleboxes and overload them, which results in failing to fulfill the performance requirements of VIPS. For example, overloaded middleboxes typically cause high packet loss rate, which may omit malicious packets and leave potentially attacks undetected [8]. Another example is that overloaded middleboxes need to buffer the in-flight traffic, which increases the processing time of VIPS.

Second, the traffic should be distributed to the scaled middleboxes while reducing the overhead of flow detection state sharing [14] and achieving load balance [15]. Assume that the middleboxes implement a per-flow level intrusion prevention (e.g., Snort, Suricata [1)]) for VIPS, the state of each flow must be completely maintained by a middlebox. However, in some cases, flow has to be divided into some parts and distributed to multiple middleboxes, its flow detection state must be shared by the middleboxes for semantic consistency [9,10]. In practice, the sharing of flow detection state is typically implemented based on the shared data store such as RAMCloud [17], Algo-logic [2)], which incurs overheads for storage and communication with the data store. Therefore, the traffic should be distributed to the middleboxes in the manner of avoiding flow division, thus minimizing flow detection state sharing. Moreover, ensuring load balance for flows to middleboxes while traffic distribution can improve the efficiency and reliability of the network.

Appendix B Middlebox Scaler

The middlebox scaler (*MS*) module scales out/in fewest middleboxes to match the time-varying traffic volumes during runtime while guaranteeing rigorous IPS performance requirements. In this subsection, we describe the method to compute the minimal number of middleboxes to be scaled given traffic volumes and IPS performance requirements in detail. In addition, we present how *MS* can scale the middleboxes according to the computed results.

As shown in Figure B1, the traffic processing of ESBox is modeled as several $M/M/1/K$ queueing systems [18]. $M/M/1/K$ *queueing system* refers to a non-trivial queueing system. Specifically, the first 'M' indicates the customers

* Corresponding author (email: wuchunming@zju.edu.cn)
1) Suricata: https://suricata-ids.org/
2) Algo-logic systems: https://algo-logic.com/

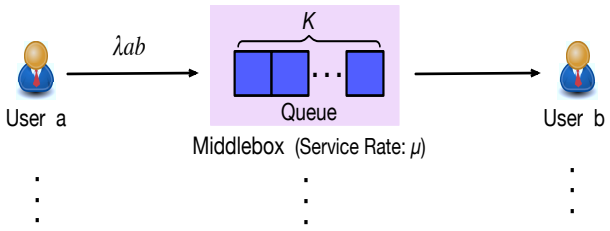

Figure B1 Modeling traffic processing of ESBox.

arrive at the queueing system according to a Poisson process, the second 'M' indicates the service time for customers are exponentially distributed, '1' indicates the number of server in the queueing system is 1, and 'K' indicates the queueing system has a capacity of to accommodate K customers simultaneously. Such modeling is rooted in the following assumptions and observations:

• Considering that each individual packet of traffic will be redirected to its allocated middlebox for detection and prevention. Thus, we regard the packet and middlebox as the customer and server in the queue theory, respectively. As widely applied in cloud performance analysis [19, 20], the packet arrival rate of a middlebox follows a Poisson distribution, and the service time follows an exponential distribution. Therefore, the traffic processing by a single middlebox can be modeled as an $M/M/1$ queueing system.

• Under heavy traffic load conditions, the Snort application buffers in the middleboxes may fill up quickly and many incoming packets may drop [21]. Hence, the capacity of a queueing system K is used to indicate the threshold after which the middlebox starts to drop packets. Therefore, the $M/M/1$ queueing system can be extended to $M/M/1/K$.

• ESBox is able to scale out/in middleboxes. Thus, it is natural that more than one middlebox become involved in the traffic processing, and we denote n as the number of middleboxes. Hence, we model the traffic processing of ESBox as n $M/M/1/K$ queueing systems.

Moreover, the inputs that the model handles are formalized as follows:

1.Traffic matrix: To formulate the dynamic traffic, we let $TM = \{\lambda_{ab} | a, b \in U\}$ be the traffic matrix (TM) at the present time, where U is the set of the users, and λ_{ab} is the packet arrival rate for the origin–destination pair user a to user b. In practice, λ_{ab} can be obtained from the data plane through the Openflow [3] protocol or other tools (e.g., sFlow [4]).

2.IPS performance metrics: Two metrics that are widely considered in IPS performance measurement [1] are as follows: (1) Processing time of prevention (PTP) describes the time lapse between launch of attack packet and being filtered by the IPS. (2) Prevention rate (PR) refers to the proportion of the packets completely processed by IPS in the total transmitted packets. We let PTP and PR be the IPS performance constraint, which is the maximal threshold specified by an administrator. Moreover, the processing time of a middlebox follows an exponential distribution with parameter μ, and K is the queue capacity of a middlebox. μ and K in our model can be determined by trial in §4.1.

The MS is in charge of computing the number of middleboxes to be scaled out/in given TM, PTP, PR, μ, and K. To guarantee high resource utilization, the middleboxes should be invested to a minimum. Thus, with the system model, our goal is to compute the minimal total number of middleboxes n.

We can obtain the total packet arrive rate λ_t in the system by accumulating the elements in TM:

$$\lambda_t = \sum_{a,b \in U} \lambda_{ab}. \tag{B1}$$

Owing to the limited capacity of each individual middlebox, processing a huge volume of traffic requires more than one middlebox. To balance the load, traffic will be assigned to the middleboxes evenly. Thus, given λ_{Total}, the packet arrival rate of each individual middlebox is as follows:

$$\lambda = \frac{\lambda_t}{n}. \tag{B2}$$

In addition, the packet process time in our system can be denoted as ϕ. Hence, given the middleboxes performances μ and K, along with the incoming packet arrival rate λ, their relationship can be derived based on [18], and we can obtain the middlebox delay:

$$\phi(n) = \frac{(1-\rho^K)(1-(K+1)\rho^K + K\rho^{K+1})}{\mu(1-\rho)(1-\rho^{K+1})^2}, \tag{B3}$$

where $\rho = \frac{\lambda}{\mu}$, representing the server utilization of the queueing system.

Furthermore, the packet loss rate ψ of the system can be expressed as follows [18]:

$$\psi(\lambda) = \frac{\rho^K}{\sum_{k=0}^{K} \rho^k} \tag{B4}$$

3) OpenFlow Switch Specification: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

4) sflow: https://sflow.org/

Subsequently, we substitute formula (B2) and into formula (B3) and (B4), to derive the relationship between n and ϕ, ψ, respectively:

$$\phi(n) = \frac{1-(K+2)(\frac{\lambda_t}{\mu n})^K + K(\frac{\lambda_t}{\mu n})^{K+1} + \dots}{\mu(\frac{\lambda_t}{\mu n}) \times \dots} \frac{(K+1)(\frac{\lambda_t}{\mu n})^{2K} - K(\frac{\lambda_t}{\mu n})^{2K+1}}{(1-(\frac{\lambda_t}{\mu n})^{K+1})^2} + d_n, \tag{B5}$$

$$\psi(n) = 1 - \frac{1-\frac{\lambda_t}{\mu n}}{(\frac{\mu n}{\lambda_t})^K - \frac{\lambda_t}{\mu n}}, \tag{B6}$$

where d_n is the network delay. Based on the formula (B5) and (B6), the computation of the minimum number of middleboxes can be transformed into an optimization problem, which is formulated as follows:

$$mini. \quad n \tag{B7a}$$

$$s.t. \quad \phi(n) \leqslant PTP \tag{B7b}$$

$$\psi(n) \leqslant PR \tag{B7c}$$

$$n = 0, 1 \dots N_{max}, \tag{B7d}$$

where N_{max} is the maximum number of middleboxes that can be invested in our system. This problem can be easily computed by iterating on the values of n.

Afterwards, given n middleboxes in total, the number of middleboxes that each flow demands can be determined. To approach this, Middlebox Demand Matrix (MDM) is used. Specifically, let $MDM = \{n_{ab} | a, b \in U\}$ denote the number of middleboxes demanded by the flow from user a to user b. For adapting to the traffic volume, n_{ab} is in proportion to λ_{ab}, thus we have:

$$n_{ab} = \frac{\lambda_{ab}}{\lambda t} \times n. \tag{B8}$$

Note that n_{ab} can be a decimal number, which means several flows may share a same middlebox. And MDM and n will be output to the next module (i.e. Flow Distributor) for further distribution instruction.

In addition, to scale middleboxes according to the planned number n, this module should be able to adjust the number of middleboxes. As mentioned before, a middlebox is a Docker container that encapsulates a Snort application. Hence, the *MS* module calls the Docker Daemon commands to implements the adjustment. More specifically, to scale out middleboxes, the *MS* must execute `docker run`. Similarly, for scaling in middleboxes, `docker rm` can be used. Furthermore, the Docker Daemon commands will execute in parallel with the GNU Parallel [5)] tool to accelerate the scaling process.

Appendix C Flow Distributor

The flow distributor (FT) module is devoted to distributing the flows in TM to the n scaled middleboxes based on MDM from the *MS* module. Dedicated forwarding rules will be installed into the SDN switches for distributing traffic to the middleboxes and sending legitimate traffic to its original destination. Therefore, a careful flow distribution scheme is needed to achieve the goal of 1) minimizing sharing of flow detection state and 2) load balance. In the subsection, firstly, a flow distribution algorithm is introduced for reducing flow division, in order to minimize sharing of flow detection state. After that, we present how the group table in SDN switches is used for load balance.

Appendix C.1 Flow distribution algorithm

A bad flow distribution algorithm will lead to unnecessary flow divisions that increase the sharing of flow detection state meaninglessly. We herein summarize two kinds of unnecessary flow divisions. The first one is rooted in the improper flow distribution priority (Example 1). The other one stems from the mismatch of the middlebox to the flow (Example 2). The following toy examples explain the two unnecessary flow divisions.

Example 1: We consider two flows $\{f_1, f_2\}$ with the required middlebox numbers of $\{0.6, 0.8\}$, respectively. Our goal is to distribute these two flows to M_1 and M_2 with the capacities of 0.8 and 0.7, respectively. If f_1 takes priority of f_2, that is, if we obtain that M_1 can accommodate f_1, we then distribute f_1 to M_1 directly and the spare capacity of M_1 is 0.2. Subsequently, we try distributing f_2; however, neither M_1 nor M_2 can accommodate f_2. Thus, we have to divide f_2, which results in one division compared with distributing f_1 to M_2 first and then f_2 to M_1.

Example 2: Consider the same flows and middleboxes as in example 1. Suppose we try distributing f_2 first, but if we mismatch M_2 to f_2, we have to divide f_2 into two parts (i.e. 0.1, 0.7) and then allocate them to both M_1 and M_2. Subsequently, we distribute f_1 to M_1. Consequently, one division is generated.

Motivated by the toy examples above, we find that the unnecessary flow divisions are avoidable. Thus, we regard Proposition 1 as the principle for avoiding the unnecessary flow divisions. We herein describe a proof process for this proposition.

5) GNU Parallel: https://www.gnu.org/software/parallel/

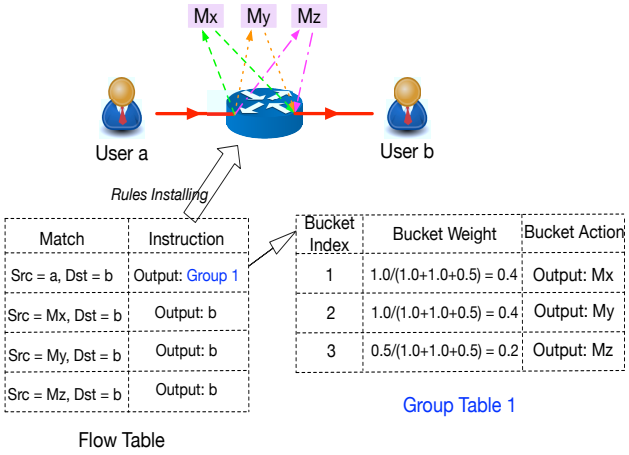

Figure C1 Example of rules of group table for traffic redirection.

Proposition 1. The larger flows are prioritized over the smaller flows during the flow distribution. Further, provided that a middlebox can accommodate the flow, we distribute the flow directly to the middlebox rather than divide the flow.

Proof. We assume $F = \{f_1, ..., f_t\}$ is the set of flows with the required middlebox number of $\{n_1, ..., n_t\}$. Further, we use NFD_i to denote the Number of Flow Division for f_i. We let $M = \{M_1, M_2, ...M_n\}$ denote the middleboxes with the corresponding capacities of $\{m_1, ..., m_n\}$. Our goal is to distribute a flow $f_i : \forall f_i \in F$ to M.

① We divide f_i into $k(k \geqslant 2)$ parts with the needs of $\{n_i^1, ..., n_i^k\}$ and then distribute them to the middleboxes $\{M_1, ..., M_k\}$, respectively. Consequently, $NFD_i = k - 1$.

② Alternatively, we distribute f_i to M_r directly (where $m_r \geqslant n_i$), and $NFD_i = 0$. We assume that $\exists f_j \in F : n_j \leqslant M_r \bigwedge n_j > (M_r - n_i)$ and the entire f_j is distributed to M_r, that is, $NFD_j = 0$. Meanwhile, the entire f_j cannot be distributed to M_r because f_i has fully or partially occupied M_r in ②. Therefore, based on the distribution priority, we have $n_j \leqslant n_i$; hence, in the worst case, we can divide f_j into k parts with the needs of $\{n_i^1, ..., n_i^k - (n_i - n_j)\}$ and then distribute them to the middleboxes $\{M_1, ..., M_k\}$. Thus, $NFD_j \leqslant k - 1$. Therefore, $NFD_i + NFD_j$ in ② $\leqslant NFD_i + NFD_j$ in ①. Otherwise, we assume that f_j mentioned above does not exist, and naturally we have NFD_i in ② $< NFD_i$ in ①.

Based on Proposition 1, we present the flow distribution algorithm (Algorithm C1) for reducing the number of flow divisions. A brief explanation is as follows. We use FDS to store the flow distribution scheme, where each distribution is represented as $[ab, i, c_i]$; this means we will allocate the c_i capacity of M_i to the flow with the origin–destination pair of ab. Lines 1 to 9 conduct the initialization of $\mathbb{F}$ and $\mathbb{C}$, where $\mathbb{F}$ is the set of flows to which the middleboxes have not yet been allocated, and $\mathbb{C}$ is the set of the available capacity of the middleboxes. Lines 10 to 30 are the loops to impose FDS with the constraint of $\mathbb{F} \neq \emptyset$; this means the loop continues until all the flows are middlebox allocated. Lines 11 to 19 handle the flows that cannot be accommodated by any middlebox capacity in $\mathbb{C}$, thus requiring necessary divisions. Each such flow is divided into several parts to reach the middlebox capacity. Lines 21 to 29 handle the flows sharing the middlebox capacity, which may yield unnecessary divisions based on Proposition 1. Line 20 sorts $\mathbb{F}$ in descending order according to the flow volumes to give priority to larger flows (we use merge-sort in this study). Subsequently, lines 22 to 29 search $\mathbb{C}$ to obtain the middlebox capacity that can accommodate the entire flow and allocate it to the flow. In addition, flows that cannot be accommodated by any element in $\mathbb{C}$ remain and will be handled in the next loop.

Because each loop of *Algorithm 1* consists of the traversal of $\mathbb{C}$ in the traversal of $\mathbb{F}$ and the merge-sort of $\mathbb{F}$, its complexity is $\mathcal{O}(l(|\mathbb{F}||\mathbb{C}| + |\mathbb{F}|log_2|\mathbb{F}|))$, and l is the loop count.

Appendix C.2 Rules of Group Table

Based on the flow distribution scheme FDS, the FT installs group table rules into the SDN switches for traffic redirection while achieving load balance. Note that the rules can be installed into any switches as long as there is a resource pool for middlebox scaling.

Indeed, the forwarding rules are the `OFPT_FLOW_MOD` messages defined by the Openflow protocol to modify the previous forwarding rules. Furthermore, the traffic redirection to multiple scaled middleboxes is performed by the group table of SDN switches, defined by the Openflow protocol. Rules in the group table include a list of action buckets, each of which forwards a flow part to a middlebox. In addition, the flow volumes allocated to each action bucket are proportional to its bucket weight, which can be used to achieve load balance. Moreover, the FT installs the forwarding rules into transport the traffic from the middleboxes to its original destination.

Figure C1 shows an example of the forwarding rules and how they operate. Given a traffic flow from user a to user b, we obtain M_x, M_y, and M_z from FDS as the middleboxes with the corresponding capacities of 1.0, 1.0, and 0.5. For simplicity, we replace the port names with the switches that they are connected to, e.g., instead of the name of the port connected to user b, we simply write b. The flow from user a is instructed by Rule 1 to output to Group 1. Subsequently, the flow is divided into three flow parts with different flow volumes and is distributed to M_x, M_y, M_z respectively, which is instructed

Algorithm C1 Middlebox allocation algorithm

Require: MDM, the middleboxes demand matrix. n, the number of middleboxes.
Ensure: FDS, the flow distribution scheme.

Initialization:
for all n_{ab} in MDM **do**
  $f_{ab} \leftarrow n_{ab}$
  $\mathbb{F}$.Append(f_{ab})
end for
for $i = 1$; $i <= n$; $i++$ **do**
  $c_i \leftarrow 1$
  $\mathbb{C}$.Append(c_i)
end for
while $\mathbb{F} \neq \emptyset$ **do**
  for all f_{ab} in $\mathbb{F}$ **do**
    for all c_i in $\mathbb{C}$ **do**
      if $m_i \leqslant f_{ab}$ **then**
        FDS.append($[ab, i, c_i]$)
        $f_{ab} \leftarrow f_{ab} - c_i$
        $\mathbb{C}.Remove(m_i)$
      end if
    end for
  end for
  Sort $\mathbb{F}$ in descending order.
  for all f_{ab} in $\mathbb{F}$ **do**
    for all c_i in $\mathbb{C}$ **do**
      if $c_i > f_{ab}$ **then**
        FDS.append($[ab, i, c_i]$)
        $c_i \leftarrow c_i - f_{ab}$
        $\mathbb{F}.Remove(f_{ab})$
      end if
    end for
  end for
end while

by Group 1. Finally, the flow parts that have been handled in the middleboxes are distributed to user b, instructed by Rules 2 to 4.

In addition, the rules installed in the last distribution interval will be deleted by the `OFPT_FLOW_MOD` message ahead of a new flow distribution.

Appendix D Evaluation

Several experiments were performed to evaluate the performance of ESBox with our prototype. We are interested in answering the following questions: **Q1:** Can ESBox scale out/in middleboxes according to the dynamic traffic volumes and guarantee the IPS performance requirements? (Experiment 1) **Q2:** Can ESBox scale out/in middleboxes while achieving high resource usage? (Experiment 2) **Q3:** Can ESBox save flow detection state sharing? (Experiment 3) **Q4:** To what extent can ESBox affect the throughput of the users? (Experiment 4)

Appendix D.1 Experimental Setup

Control plane construction: ESBox was implemented as an SDN application on the Ryu 4.23, a popular SDN controller. ESBox contains approximately 800 lines of python code. And the Ryu was run on a server with 16GB of memory and Intel 4-core i5-5287u 2.0GHz processor.

Data plane construction: The Docker 17.03.2 was used, a lightweight container with the image of Ubuntu 14.04, configured to use 200MB RAM, to create the operating environment of each user and middlebox. Snort 2.9.11.1 tool was encapsulated in the middlebox and configured in in-line mode to enforce the detection and prevention for the received

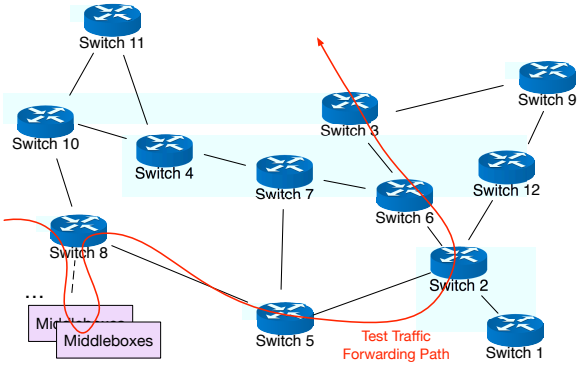

Figure D1 Experimental network topology.

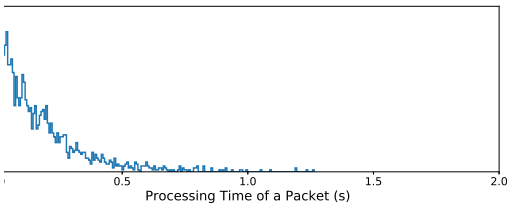

Figure D2 Probability Density Function of the Processing Time.

traffic. Lightweight MySql Database [6] is used as the database for storing the shared flow detection states of Snort. CICIDS2017 dataset [7], the common-used dataset for intrusion detection test including both normal and abnormal traffic was chosen as our test traffic. Moreover, Tcpreplay [8] tool running on the user container was used to send the test traffic. Open vSwitch (OVS) 2.3.2 was used as the SDN switch to interconnect the users as well as the users and the middleboxes. The layout of the SDN switches was based on the real topology from Abilene dataset, as shown in Figure D1. The data plane of our experiment ran on a server with 16GB of memory and Intel 4-core Xeon E5-2609 v2 2.5GHz processor. This server was linked to the server above with the mean delay of 0.754ms.

Parameter setting: We determined the parameters that are fundamental to ESBox by three trials. The first trial is to determine the network delay d_n. We measured the mean value of d_n using the iPerf [9] tool, and obtained $d_n = 0.189$ ms. The second trial is to determine the parameter μ of the exponential distribution of the processing time. We randomly sampled 5,000 packets with various protocol and packet length from NSL-KDD dataset, sent the packets to one middlebox at a rate of 1 packet/s to ensure each packet was fully processed, and measured the processing time of each packet. (The measured probability density function (PDF) of processing time is shown in Figure D2.) Then, based on the measured data, we obtained $\mu = 5.96$ by using Maximum Likelihood Estimation [10]. The third trial is to K, the capacity of the queue in a middlebox. We randomly sampled the test traffic and input it at different rates to a middlebox for 50 times, we found that the middlebox approximately started to drop packet when the rate is set to 21 packet/s. Thus, we determined $K = 21$ in our prototype. Furthermore, the scaling time scale was set to 1 min.

Appendix D.2 Experiment 1: Scalability and IPS Performance Guarantee

This experiment is a proof-of-concept experiment to reveal the fact that ESBox can scale out/in middleboxes according to the traffic while guaranteeing given IPS performance requirements. First, the traffic demands of common-used Abilene dataset [11] were studied to generate the test traffic for driving the experiment. To highlight the traffic dynamics, the traffic demand from *Los Angeles* to *Chicago* of in Abilene was focused. As per the dynamics of the traffic demand above, the test traffic was sent at mean rates varying with time intervals of 1 min, as depicted in Figure D3.

As shown in Figure D1, the traffic from *Los Angeles* to *Chicago* was sent by the user attached with Switch 3 to the user attached with Switch 8. Consequently, ESBox gathered the traffic statistics from Switch 3, adjusted the number of middleboxes to handle the dynamics of the test traffic and installed forwarding rules into Switch 3, to redirect the test traffic to the middleboxes. Further, the IPS performance constraints were set as follows: PTP was set to 0.5 s, 1 s, and 2 s, in order to fulfill service level agreements (SLA) for legitimate users in different common scenarios, such as time-sensitive application, the worst situation in time-sensitive application, and non-time-sensitive application. Moreover, PR was set to 90% and 95%, according to the experimental results of [21], the basic processing rate of a Linux-kernel-based IPS is around 90% while processing malicious traffic of the same order of magnitude as our test traffic. And we use '95%' to raise the bar of the basic processing rate to test our prototype.

Figure D4 shows the number of middleboxes for the test traffic at different mean rates for 200 min and under the different IPS performance constraints. By comparing Figure D4 with Figure D3, it is noticed that the number of middleboxes under any IPS performance constraint is adjusted dynamically to follow the dynamic test traffic rates closely. Also, it is found that there are approximative overlaps of the curves when $PTP = 0.5$ s and 1 s, respectively. This is because ESBox searches the minimal number of middleboxes to simultaneously satisfy PTP and PR, but PR is satisfied before PTP under the above conditions, thus the number of middleboxes is completely up to PTP.

6) MySQL: https://www.mysql.com/
7) CICIDS dataset: https://www.unb.ca/cic/datasets/ids-2017.html
8) Tcpreplay: https://tcpreplay.appneta.com/
9) Iperf: https://iperf.fr/
10) Maximum Likelihood Estimation: https://www.itl.nist.gov/div898/handbook/apr/section4/apr412.htm
11) Abilene: http://www.cs.utexas.edu/~yzhang/research/AbileneTM/

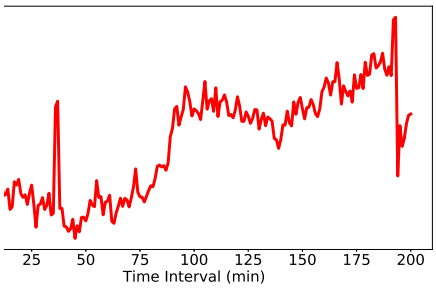

Figure D3 Mean rate of the test traffic.

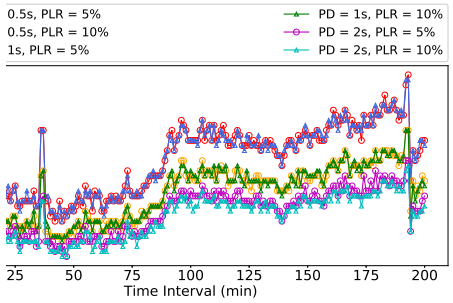

Figure D4 Number of middleboxes under different IPS performance constraints.

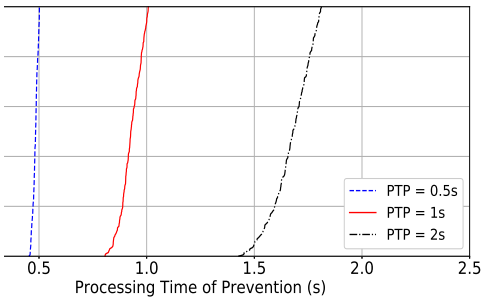

Figure D5 CDF of the processing time of prevention.

Table D1 Prevention rate under different IPS performance constraints.

IPS Performance Constraints		
PTP (s)	PR (%)	Prevention Rate (%)
0.5	95	99.9
0.5	90	99.9
1	95	99.7
1	90	99.7
2	95	96.9
2	90	95.1

Moreover, the actual processing time of prevention and prevention rate were measured using the iPerf tool. Figure D5 shows the cumulative distribution function (CDF) of the measured processing times of prevention across the 3 PTPs. This is sampled 60 times every minute. The results show that 93%,94.4%, and 100% of the traffic satisfy their PTPs. Table D1 shows the prevention rate under the different IPS performance constraints. The results indicate that all the PRs are satisfied.

Appendix D.3 Experiment 2: High Resource Utilization

To measure the resource utilization of ESBox, two alternative scheme are used as comparison: (1) *Static*: This scheme can not scale middleboxes, to simulate the non-scalable VIPSes (e.g., [7, 14, 22, 24, 27, 28, 32]). Thus the number of middleboxes provisioned is static, which is set equal to the mean value of the number of middleboxes scaled in ESBox. (2) *BroFlow* [11,12]: This scheme scales out another middlebox whereas a middlebox is overload, and evenly redistributes the traffic to the middleboxs. Moreover, it scales in two underloaded middleboxes to one and aggregates traffic to the middlebox remained. We used *BroFlow* as our comparison example of scalable VIPS because it provides detailed scaling strategy, whereas other schemes like [12, 31] do not. The same test traffic was used as Experiment 1. For each scheme, we recorded the average number of middleboxes used per minute, the mean value of prevention rate and the processing time of prevention, based on the log from the control plane.

Figure D6 presents the average number of middleboxes used under different IPS performance constraints. Moreover, Figure D7 and Figure D8 illustrate the prevention rate and the processing time of prevention under different IPS performance constraints, respectively. We make the following observations. First, ESBox uses fewer middleboxes than *BroFlow*, and nearly same number of *Stacic*. Second, in any case, ESBox performs better in terms of higher prevention rates, and in most case more stable and shorter processing time of prevention, than the other two schemes. Therefore, the result indicates that ESBox has a higher resource utilization.

Appendix D.4 Experiment 3: Minimizing Flow Detection State Sharing

The test traffic was the same as that in Experiment 1. Note that this experiment did not compare with other VIPS scheme, it is because that none of other VIPS achieves minimizing flow detection state sharing after the middlebox scaling. The key difference with this experiment from Experiment 1 was that we created different numbers of origin–destination pairs of users attached with Switch 3 and Switch 8 to send and to receive the test traffic, respectively. Further, the source and destination users that send and receive the test traffic were selected following a Pareto distribution (ParetoD) [23] or a random distribution (RandomD). The numbers of shared flow detection state that ESBox generated for redirecting test traffic were recorded from the database we used. Considering that ESBox prototype lies primarily in the flow distribution

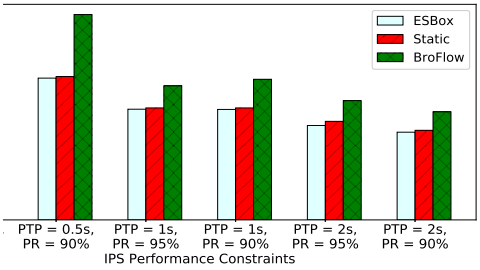

Figure D6 Average number of middleboxes.

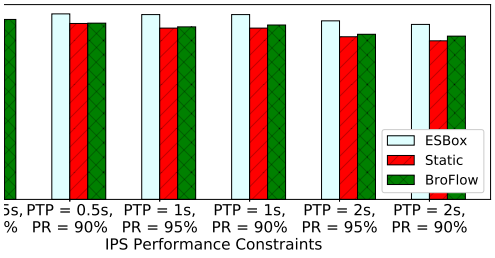

Figure D7 Prevention rate measurement.

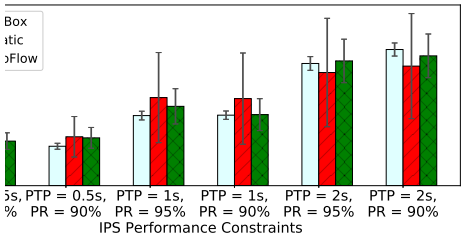

Figure D8 Processing time of prevention measurement.

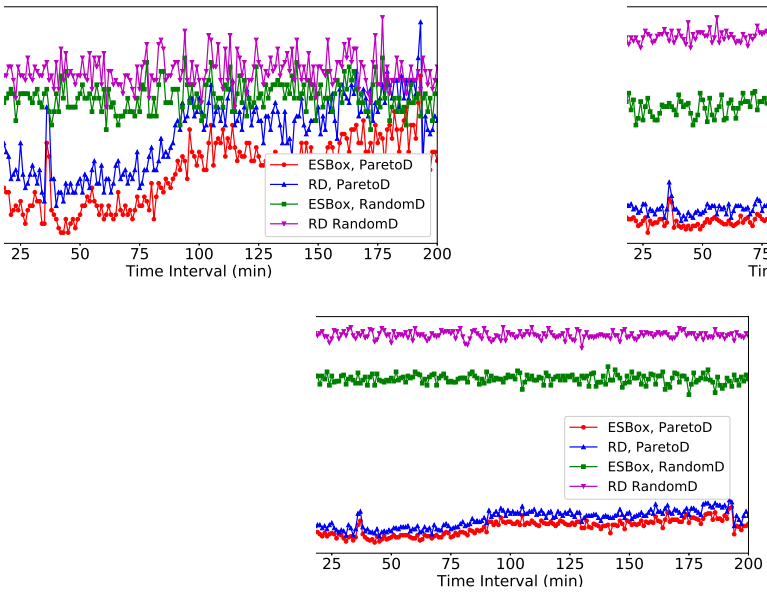

Figure D9 Number of shared flow detection state. (a) 10; (b) 20; (c) 30 origin–destination pairs of users.

algorithm of its *FT* module to reduce the number of shared flow detection state. Thus, as a baseline, we replaced our flow distribution algorithm with a random distribution scheme (DA) and recorded the number of such states generated.

The results are plotted in Figure D9. It is observed that our ESBox prototype always generated less flow detection state records than the random distribution scheme. This demonstrates that ESBox significantly reduces the flow detection state sharing.

Appendix D.5 Experiment 4: Impact on Traffic Throughput

During the reallocation of middlebox instructed by ESBox, the number of middleboxes may mismatch the traffic volumes until the reallocation completes. Such mismatch leads to the buffering of in-flight traffic in the middleboxes, which is

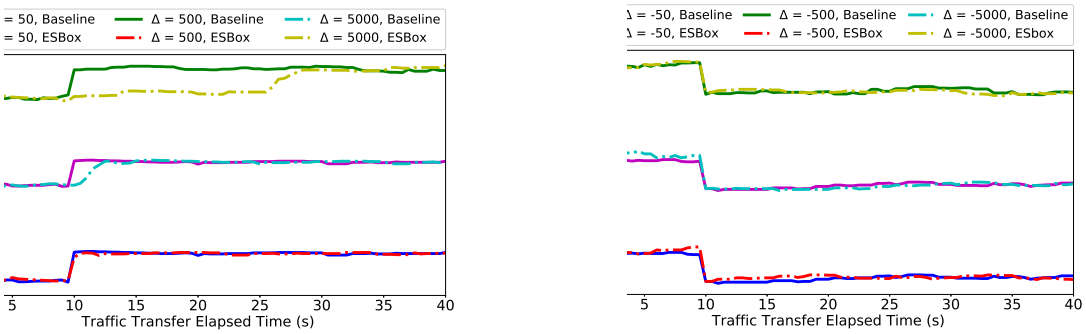

Figure D10 Impact on traffic throughput. (a) Increasing traffic volumes; (b) Decreasing traffic volumes.

expected to degrade the throughput of user traffic. In this experiment, we studied to what extent ESBox affects the throughput. It is noteworthy that we did not compare ESBox with other VIPS scheme, because: according to Section 4.2, the prevention rate of ESBox is higher than the Static and Browflow schemes, which indicates the packet loss rate of these two schemes is higher than ESBox. It can be inferred that these two schemes have a more serious traffic degradation. Thus, these two scheme were not involved. The traffic forwarding path was the same as that in Experiment 1. The test traffic highlights the change in traffic volumes (the increment is given by Δ) to incur reallocation, which includes the following: (1) *Increasing traffic volumes:* The traffic volumes increase from 50, 500, 5k packets/s to 100,1k,10k packets/s ($\Delta = 50, 500, 5k$), respectively. (2) *Decreasing traffic volumes:* The traffic volumes decrease from 100,1k, 10k packets/s to 50,500,5k packets/s ($\Delta = -50, -500, -5k$), respectively.

Figure D10(a) shows the impact on the throughput for the increasing traffic volumes. The solid lines, as baselines, indicate the throughput of the traffic without ESBox. The dotted lines indicate the throughput of the traffic with ESBox. The traffic increased at $t = 10s$ with $\Delta = 50, 500, 5k$; thus, the reallocations were scheduled accordingly. We observed two key findings from the experiment. First, ESBox demonstrates no impact on the throughput of the traffic when the traffic volumes have no increment. Second, the duration of throughput degradation increases as the increment of the traffic volumes increase. However, the duration is short (≈ 0.5 s) when $\Delta = 50$. Further, when $\Delta = 5k$, the duration is acceptable: (≈ 18 s). It is remarked that the durations are the result of the time it takes our server to initialize the middleboxes, and in practical situations, the degradation can be mitigated or even eliminated by using a high-performance server. Figure D10(b) shows the effect on the throughput for the decreasing traffic volumes. It is observed that no throughput degradation in ESBox when processing traffic with the decreasing volumes.

Appendix E Related Work

A body of solutions have used SDN techniques to enhance the security of network. Some SDN-based IPSs relied completely on the SDN controller to implement its security functions. These solutions applied the SDN controller to aggregate statistical information from the SDN switches in the data plane, analyze the information by a variety of means (e.g. deep learning [26], extracting and aggregating features [28], sFlow analyzer [29], and Advanced Support Vector Machine [30]). Nevertheless, these solutions may cause a heavy burden in the control plane.

Some SDN-based VIPSs can avoid the heavy burden aforementioned by deploying middleboxes within a function of IDPS to the data plane. The control plane was simply responsible for steering the composition of middleboxes and redirecting traffic to them. [14, 27] focuses on improving the structural design of VIPS to enable effective intrusion detection and non-monolithic NIDS provisioning. [22] proposes a traffic sampling rate adjustment method for fully utilizing the inspection capability of malicious traffic while the total aggregate volume of the sampled traffic is kept below the maximum processing capacity of the IDS middlebox. [7] maximally uses existing middleboxes and optimally routes traffic to the middleboxes. [24] provides a management architecture of VIPS, which enables control distributed middleboxes in switches. [25] proposes a flow-table sharing approach to protect the SDN-based cloud from flow table overloading DDoS attacks. [28] presents a framework that automatically detects and mitigate DDoS attacks by the SDN controller. [32] designs a 2-phase algorithm that can quickly select DPI middlebox and find routing paths for incoming flows in SDN. These approaches hardly support scalability for handling ubiquitous traffic volume variations. However, they can be orthogonal to our approach in middlebox management for VIPS.

Furthermore, existing VIPS solutions such as BroFlow [11, 12], DEIDtect [13], and AsIDPS [31] support horizontal scaling of middlebox resources. However, the resource adjustment reacts to the awareness of overloaded or underloaded middleboxes, not to the dynamics of traffic volume, which makes these solutions ex-post. Also, these solutions cannot compute and then adjust the number of required middleboxes according to the traffic volumes, which leads to coarse-grained scaling of middleboxes. The major difference between these work and our work lie in that ESBox reacts to real-time traffic and provides fine-grained scaling with high resource usage.

References

1 Scarfone K, Mell P. Guide to intrusion detection and prevention systems (idps). NIST special publication, 2007, 800:94

2 Malathi V, Takehiro S, Molly B, et al. Network Function Virtualization: A Survey. IEICE Transactions, 2017, 100-B:1978–1991

3 Preeti M, Emmanuel S, Vijay V, et al. Intrusion detection techniques in cloud environment: A survey. J. Network and Computer Applications, 2017, 77:18–47

4 Wei X, Hanping H, Naixue X, et al. Anomaly secure detection methods by analyzing dynamic characteristics of the network traffic in cloud communications. Inf. Sci., 2014, 258:403–415

5 Srikanth K, Sudipta S, Albert G, et al. The nature of data center traffic: measurements analysis. In: Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference, Chicago, Illinois, USA, 2009. 202–208

6 Diego K, Fernando M R, Paulo J E V, et al. Software-Defined Networking: A Comprehensive Survey. Proceedings of the IEEE, 2015, 103:14–76

7 Seungwon S, Haopei W, Guofei G. A First Step Toward Network Security Virtualization: From Concept To Prototype. IEEE Trans. Information Forensics and Security, 2015, 10:2236–2249

8 Holger D, Anja F, Vern P, et al. Predicting the Resource Consumption of Network Intrusion Detection Systems. In: Recent Advances in Intrusion Detection, 11th International Symposium, Cambridge, MA, USA, 2008. 135–154

9 Lorenzo D C, Robin S, Somesh J. Beyond Pattern Matching: A Concurrency Model for Stateful Deep Packet Inspection. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, 2014. 1378–1390

10 Matthias V, Robin S, Jason L, et al. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In: Recent Advances in Intrusion Detection, 10th International Symposium, Gold Goast, Australia, 2007. 107–126

11 Martin A L, Otto C M. Providing elasticity to intrusion detection systems in virtualized Software Defined Networks. In: 2015 IEEE International Conference on Communications, London, United Kingdom, 2015. 7120–7125

12 Martin A L, Diogo M F M, Otto C M. An elastic intrusion detection system for software networks. Annales des Telecommunications, 2016, 71:595–605

13 Praveen K S, Naveen D S, Joe B, et al. DEIDtect: towards distributed elastic intrusion detection. In: Proceedings of the 2014 ACM SIGCOMM workshop on Distributed cloud computing, DCC '14, Chicago, Illinois, USA, 2014. 17–24

14 Hongda L, Hongxin H, Guofei G, et al. vNIDS: Towards Elastic Security with Safe and Efficient Virtualization of Network Intrusion Detection Systems. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, 2018. 17–34

15 Vladimir A O, Felipe H, Costin R. Lost in Network Address Translation: Lessons from Scaling the World's Simplest Middlebox. In: Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, HotMiddlebox@SIGCOMM2015, London, United Kingdom, 2015. 19–24

16 Zhihua L, Chengyu Y, Lei Y, et al. Energy-aware and multi-resource overload probability constraint-based virtual machine dynamic consolidation method. Future Generation Comp. Syst., 2018, 80:139–156

17 John K O, Parag A, David E, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. Operating Systems Review, 2009, 43:92–105

18 Sztrik J. Basic queueing theory. University of Debrecen, Faculty of Informatics, 2012, 193

19 Shui Y, Yonghong T, Song G. Can We Beat DDoS Attacks in Clouds? IEEE Trans. Parallel Distrib. Syst., 2014, 25:2245–2254

20 Hamzeh K, Jelena V M, Vojislav B M. Performance of Cloud Centers with High Degree of Virtualization under Batch Task Arrivals. IEEE Trans. Parallel Distrib. Syst., 2013, 24:2429–2438

21 Khaled S, A. Kahtani. Performance evaluation comparison of Snort NIDS under Linux and Windows Server. J. Network and Computer Applications, 2010, 33:6–15

22 Taejin H, Sunghwan K, Namwon A, et al. Suspicious traffic sampling for intrusion detection in software-defined networks. J. Computer Networks, 2016, 109:172–182

23 Heng C, Ghassan O, Felix K, et al. On the Fingerprinting of Software-Defined Networks. IEEE Trans. Information Forensics and Security, 2016, 11:2160–2173

24 Wen W, Wenbo H, Jinshu S. Network intrusion detection and prevention middlebox management in SDN. In: IEEE International Performance Computing and Communications Conference, IPCCC 2015, Nanjing, China, 2015. 1–8

25 Kriti B, Brij B. Distributed denial of service (DDoS) attack mitigation in software defined network (SDN)-based cloud computing environment. J. Journal of Ambient Intelligence and Humanized Computing, 2019, 10:1985–1997

26 Qingyue M, Shihui Z, Yongmei C, et al. Deep Learning SDN Intrusion Detection Scheme Based on TW-Pooling. J. Journal of Advanced Computational Intelligence and Intelligent Informatics, 2019, 23:396–401

27 Nuyun Z, Hongda L, Hongxin H, et al. Towards Effective Virtualization of Intrusion Detection Systems. In: Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, Scottsdale, USA, 2017. 787–793

28 Georgi A, Nareg A, Imad H, et al. Flow-based Intrusion Detection System for SDN. In: 2017 IEEE Symposium on Computers and Communications, ISCC 2017, Heraklion, Greece, 2017. 787–793

29 Babatunde H, A. T. Real-time detection and mitigation of distributed denial of service (DDoS) attacks in software defined networking (SDN). In: Signal Processing and Communications Applications Conference, SIU, Izmir, Turkey, 2018. 787–793

30 Myo M, Sinchai K. Advanced Support Vector Machine- (ASVM-) Based Detection for Distributed Denial of Service (DDoS) Attack on Software Defined Networking (SDN). J. Journal Comp. Netw. and Communic., 2019, 2019:1–12

31 Junchi X, Haifeng Z, Jinfan S, et al. AsIDPS: Auto-Scaling Intrusion Detection and Prevention System for Cloud. In:

25th International Conference on Telecommunications ICT, Malo, France, 2018. 207–212
32 Huawei H, Peng L, Song G, et al. Traffic scheduling for deep packet inspection in software-defined networks. J. Concurrency and Computation: Practice and Experience, 2017. 29:1-8