A Security Orchestration System for CDN Edge Servers

Elaheh Jalalpour*, Milad Ghaznavi*, Daniel Migault†,
Stere Preda†, Makan Pourzandi†, Raouf Boutaba*

*University of Waterloo, Waterloo, ON, Canada †Ericsson Research, Montreal, QC, Canada
*{ejalalpo | eghaznav | rboutaba}@uwaterloo.ca,
†{daniel.migault | stere.preda | makan.pourzandi}@ericsson.com

***Abstract*—A Content Delivery Network (CDN) employs edge-servers caching content close to end-users to provide high Quality of Service (QoS) in serving digital content. Attacks against edge-servers are known to cause QoS degradation and disruption in serving end-users. Protecting edge-servers is vital but represents a complex task. Not only must the attack mitigation be immediately effective, but the corresponding overhead should also not negatively affect the QoS of legitimate users.**

We propose a software-based security system for CDN edge-servers to mitigate various attacks. The approach is to automatically react to threats by deploying and managing security services. These security services are realized using virtualized security function chains created, configured, and removed dynamically. The desired system behavior is governed by high-level security policies dictated by a network operator. We demonstrate how our system can be programmed using these policies to automatically handle real-world attacks. Our performance evaluation shows that our system is low-overhead, immediately responds to threats, and quickly recovers legitimate traffic throughput.

I. Introduction

Delivering digital content (e.g., video, images, and Web-pages) accounts for most of today's Internet traffic [33], [38], [28]. CDNs play a critical role in delivering digital content to end-users. Open-Connect [5] carries part of Netflix's content accounting for 35.2% of all the traffic across North America, and Akamai CDN daily delivers more than 30 Tbps of traffic [10]. A CDN contains several servers known as *edge-servers* distributed in various locations to cache content close to end-users resulting in high Quality of Service (QoS).

Attacks against edge-servers can cause disruption and QoS degradation in serving end-users, and loss in revenue for, and reputation of, a CDN provider. The main attacks against CDN edge-servers include Distributed Denial of Service (DDoS) and application-layer attacks [29], [42], [1]. DDoS attacks exhaust the resources of an edge-server. They range from network flooding (e.g., UDP fragment, SYN flood) [7] to amplification/reflection [39], and HTTP/S flooding [4]. Most CDNs host Web services [2] and are therefore prone to application-layer attacks. Common application-layer attacks include SQL injection, cross site scripting (XSS), file inclusion, and remote command execution. These attacks evolve quickly and are becoming more sophisticated (e.g., by targeting multiple layers of the protocol stack [29]). Moreover, new attacks are being introduced every day (e.g., forwarding-loop attacks [24]).

978-1-5386-4633-5/18/$31.00 © 2018 IEEE

Securing edge-servers against these main attack vectors is a complex task. Security services must be immediate in response to attacks to lessen possible damages (e.g., the later the response to a denial of service attack, the more the end-users churn). Security services affect QoS due to their processing overhead. Further, they may consume resources that are shared with CDN services. To reduce the impact of security services on legitimate end-users while responding to attacks quickly, these services must be deployed *automatically* and *dynamically* in response to threats. When a threat is detected, the relevant mitigation services must be instantiated, and these services should be removed when the threat is gone. Security services should process only relevant subsets of traffic (e.g., suspicious traffic flows). Moreover, to mitigate sophisticated and novel attacks, the protection system should allow security services to evolve (e.g., be extended with new detection or mitigation mechanisms) and novel ones to be introduced and employed.

Traditional security mechanisms do not completely meet the above requirements. Defense using *hardware*, e.g., traditional firewalls or IDSs, is expensive in terms of CAPEX and OPEX. Security attacks evolve rapidly [43], while hardware-based security capabilities do not change as quickly. These mechanisms are constrained to the resources and embedded functionality in hardware. Further, security capabilities are constrained to the limited number of available products. Protection using *scrubbing-centers* is not always applicable. Scrubbing-centers are over-provisioned cloud data-centers well-equipped with security mechanisms to filter illegitimate traffic. Redirecting to scrubbing-centers adds latency and may impact QoS. Moreover, scrubbing-centers mostly employ proprietary mechanisms that limit the ability of a CDN provider to enforce their custom security policies.

The growing movement towards network softwarization is promising. Leveraging *software defined networks* (SDN), *Network Function Virtualization* (NFV), and *service function chaining*, we can instantiate, modify, scale, and release virtualized security functions on-demand. Such flexibility in orchestration provides the means to achieve the desired protection. However, current software-defined security solutions are

insufficient. They are not tailored to the CDN environment and its security requirements. Recent work [27], [31], [19] focuses on DDoS attacks. Moreover, these solutions mainly provide *static* DDoS mitigation mechanisms that rely on a network operator to manually configure and provision. Finally, some of these solutions require deep and complex modifications to existing infrastructures.

This paper presents a security system to secure CDN edge-servers. Our system protects an edge-server where the security and content-delivery services share the same computational and networking resources. The overhead of security services is dynamically modulated to offset their negative impact on CDN services. Our system is governed by reactive high-level security policies translated into executable security orchestration actions. Security services are implemented using service function chaining. Orchestration actions dynamically create, modify, manage, and remove security services. To realize security chains, we employ general-purpose mechanisms and tools that are widely accepted in industry standards.

Our main contributions are: i) the design of a *dynamic* and *automatic* security orchestration system for protecting edge-servers, ii) a proof of concept implementation of the system, and iii) the demonstration of how our system can be flexibly programmed to handle real world use-cases. Performance evaluation results show that our system has a low overhead and can immediately respond to threats and quickly recover the throughput of legitimate traffic. In addition, using our system, we can prioritize end-users and inspect only relevant subsets of traffic.

The remainder of this paper is organized as follows. We discuss related work in Section II. In Section III, we present the design and implementation of our security orchestration system. Section IV presents two use-case scenarios in which our system dynamically manages security services. Section V presents the evaluation of our system. Finally, we conclude this paper in Section VI.

II. Related Work

Traditional Security Mechanisms. Scrubbing-centers are over-provisioned cloud data-centers that provide security mechanisms for high traffic loads. Using DNS or BGP mechanisms [3], traffic is redirected to scrubbing-centers to be inspected. Illegitimate traffic will be *scrubbed* and the remaining traffic will be forwarded to the original destination. The primary motivation for delivering content by CDNs is enhancing QoS by serving requests in the proximity of end-users. Although scrubbing-centers can provide protection, redirecting traffic to these fixed and potentially remote locations negatively impacts latency and throughput which results in QoS degradation. In addition, security services are constrained to proprietary security mechanisms offered by scrubbing-centers. In contrast, using our system, any custom security mechanism can be deployed. Also, traffic is processed locally to avoid potential latency and throughput degradation.

Software Defined Security. *DrawBridge* [32] employs end-hosts information to improve DDoS attack mitigation in an SDN-operated ISP network. End-hosts can subscribe and express their preferred traffic engineering rules. A DrawBridge controller (an SDN controller) then installs these rules in its SDN switches, or sends these rules to DrawBridge controllers of upstream ISPs. *Software Defined Security Service* (SENSS) [45] provides interfaces from ISPs that enable victims to detect and mitigate attacks across multiple SDN-operated ISP networks. However none of these proposals have seen any major deployment in real ISPs where significant upgrades have proven to be difficult or even impossible. As our system uses standard mechanisms, it can be deployed in practice without the need for major infrastructure upgrades. VFence [31] proposes a platform which performs SYN flood mitigation in a scalable manner by using dynamic allocation of virtual functions. To mitigate DDoS attacks, *Bohatei* [27] deploys a protection chain based on the attack types. The protection workflow is as follows. Bohatei i) *flags* a suspicious flow, ii) *estimates* the attack volume, iii) *places* defense functions across multiple data-centers, and iv) *steers* traffic through the chain. To mitigate SYN Flood attacks, Alharbi et al. [19] presents an NFV based platform consisting of *screening* and *resource allocation* modules. The former classifies and redirects traffic to corresponding security chains, and the latter allocates resources to chain functions. The application of these systems is limited to DDoS attacks. In contrast, our system can be used to mitigate different attacks against edge-servers.

Service Function Chaining Frameworks. CoMB [41] consolidates chain deployment by placing all functions of a chain in a single location (a *CoMB-box*). OpenBox [22] separates the data plane and the control plane of network functions and provides a northbound API for application development. Slick [20] provides a control-plane to develop applications composed of fine-grained elements performing custom packet-processing. However, to orchestrate security chains using these platforms, security functions have to be significantly modified. In contrast, standard security functions can be orchestrated using our system.

III. System Design and Implementation

Our system orchestrates security services in an edge-server environment. The edge-server can be a set of physical servers, a collection of virtual machines, or a combination of these. We refer to this environment as a *virtual edge-server*. Our goal is to design a security orchestration system that automatically and dynamically deploys and modifies security services under various environment and attack conditions. To do so, our system *reacts* to the dynamicity of the environment and attacks by instantiating and re-configuring customized security services. To minimize overhead on legitimate traffic, only relevant traffic subsets (e.g., suspicious traffic) can be processed by the security services. The security services are realized by *security chains* composed of one or multiple *virtual security functions*. The behavior of the system is regulated by *security policies* that an operator (e.g., a content provider or a CDN provider) specifies to achieve the desired security. As follows, we first

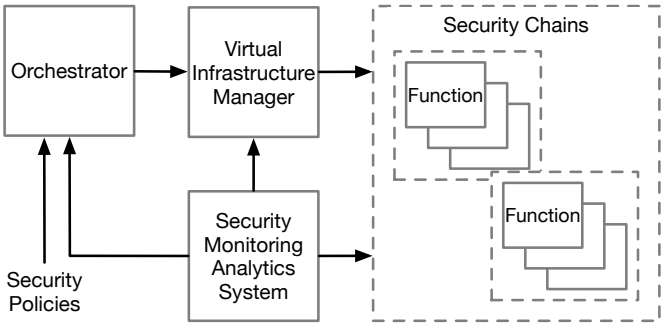


Fig. 1: Architecture

introduce the system architecture, then describe in detail each component of the system.

A. Architecture

ETSI developed monitoring and management use-cases in the context of NFV security [12]. Our system architecture adapts some of the definitions from the ETSI use cases. As depicted in Fig. 1, it consists of three components as follows.

1) Orchestrator: Interacting with other components, the orchestrator *reacts* to various environment states and attack scenarios. This reactive behavior is governed by security policies specified by the network security administrator. The orchestrator translates these high level policies into executable operations, including deploying, modifying, and removing security chains as needed. We require policies to be simple enough for the network security administrator to specify, as well as to be independent of the underlying infrastructure and technology (e.g., independent from whether the security function is deployed in a container or a virtual machine)

2) Virtual Infrastructure Manager: According to NFV-MANO [11], the Virtual Infrastructure Manager (VIM) manages and controls infrastructure resources. In our design, this module controls the resources of a virtual edge-server and manages security chains. More specifically, it is responsible for creating, updating, querying, and deleting security chains. VIM provides a north-bound API, used by the other system components, to manage and query information about security chains. We require this north-bound API to be independent from underlying implementation mechanisms.

3) Security Monitoring Analytic System: The Security Monitoring Analytic System (SMAS) monitors and analyzes data collected across the system. This module queries VIM regarding deployed security services, monitors host's resources, and retrieves the logs of security functions. Analyzing the collected data, SMAS feeds the orchestrator with alerts that may trigger security actions. We require SMAS to have a small footprint in terms of resources utilization.

B. Orchestrator

We express security policies in a language articulated around the notion of *events* that can be associated to security alerts (e.g., high-CPU load). The occurrence of an event is then associated to performing a set of *actions* (e.g., the deployment of specific security services). Further, the current state of the environment must be considered to decide whether an action should be performed. This almost naturally leads us to adapt the *Event-Condition-Action* (ECA) paradigm [26]. Accordingly, if a certain event happens, provided that particular conditions hold, a specific sequence of actions is executed. Compared to the *Condition-Action* (CA) paradigm in which events are implicit and limited in scope [30], [37], in ECA, events are separated from actions and conditions. This explicit separation enables us to define custom events to capture various attacks and environment states.

We adapt $\mathcal{L}_{active}$ [21], an ECA language, for the specification of our security policies. Here, first we list useful types of ECA rules, then we define the components of these rules.

1) ECA Rules: The following propositions are defined.

Active-rule. The occurrence of event e triggers the execution of action a if conditions $c_1, \ldots, c_n$ $(n \geq 1)$ hold:

$$e \textbf{ initiates } a \textbf{ if } c_1, \ldots, c_n$$

Causality. If conditions $c_1, \ldots, c_n$ $(n \geq 1)$ hold, the *complete* execution of action a makes $p_1, \ldots, p_m$ $(m \geq 1)$ to be true:

$$a \textbf{ causes } p_1, \ldots, p_m \textbf{ if } c_1, \ldots, c_n$$

p_i is either a condition (the same as a c_j) or a predefined procedure. In the latter case, the predefined procedure is run (e.g., $timer(t)$ that starts a timer counting t units of time).

Event. Event e occurs after the execution of action a if conditions $c_1, \ldots, c_n$ $(n \geq 1)$ hold:

$$e \textbf{ after } a \textbf{ if } c_1, \ldots, c_n$$

2) Rule Components: The **event** in an ECA rule specifies the signal that invokes this rule. An event may carry parameters providing more information regarding the event occurrence. We consider two types of events: i) security alerts generated by SMAS, and ii) *internal events* that happen as a result of executing an action. An example of a security alert is cpu_high denoting that CPU utilization is higher than a given threshold. An example of internal events is $timeout$ meaning that a certain timer has expired. The orchestrator listens on a selected TCP port to receive external events. Running an action, the orchestrator may *fire* internal events.

The **conditions** of an ECA rule are *predicates* evaluated and if satisfied, the rule actions are performed. Examples include time-related conditions, such as $date(d)$ holding if the current date is d; service related ones, e.g. $chain(x)$ and $function(y)$ holding if a chain x and function y are deployed, respectively; traffic-related, for instance $steered(t, x)$ indicating if traffic flow t, identified by a 5-tuple, is being processed by chain x.

The **actions** of an ECA rule constitute the security service logic performed if the conditions are satisfied. An action is a sequence of operations applied to security services. Actions must be defined carefully to avoid redundancy and ease policy consistency verification. As security services in our system are implemented using virtualized security function chains, we define the following elementary actions:

- $create\_chain(x : t, \{y_1{:}k_1, \ldots, y_n{:}k_n\})$ deploys a chain named x to process traffic flow t with the ordered sequence of functions. $y_i{:}k_i$ denotes a function named y_i of type k_i ($n \geq 0$ where $n = 0$ means the empty sequence of functions).
- $delete\_chain(x)$ deletes deployed chain named x.
- $insert(x, y{:}k)$ inserts a function named y of type k into existing chain named x.
- $run(y, c)$ runs command c in function named y.
- $delete(x, y)$ deletes function named y from chain named x.

Traffic flow t in action $create\_chain(.)$ is defined by tuple <f, i, j>. In this tuple, f is a Berkeley packet filter expression [34] (e.g., "*ip*" to filter IP traffic). A chain and traffic traversing through this chain create a *virtual* network. i and j are the symbolic *ingress* and *egress* of traffic in this network. We will discuss this further in Section III-C1.

3) ECA Rule Examples: Fig. 2 shows security policies that automatically deploy and remove a security chain. Rule 1 instructs that upon receiving the event $high\_cpu$ meaning high CPU usage, the orchestrator deploys the chain named x, containing an IDS function named y, and steers all traffic coming from symbolic ingress 1 through this chain, then forwards traffic to symbolic egress 2. Rule 2 fires event $conf$ after $create\_chain(.)$ action if condition $function(y)$ is valid showing that y has been installed. Rule 3 runs command "conf.sh" to configure function y. Rule 4 instructs that after the chain is created, a timer named tx is set for 10 time units. Rule 5 deletes the chain upon a timeout event for a timer tx, if condition $chain(x)$ is true, meaning that chain x exists.

$$
\begin{aligned}
high\_cpu \ &\textbf{initiates}\ create\_chain(x{:}\_, 1, 2, \{y{:}IDS\}) \\
&\textbf{if}\ \text{not}\ chain(x) && (1)\\
conf \ &\textbf{after}\ create\_chain(x) \\
&\textbf{if}\ function(y) && (2)\\
conf \ &\textbf{initiates}\ run(y, \text{"conf.sh"}) \\
&\textbf{if}\ \text{true} && (3)\\
create\_chain(x) \ &\textbf{causes}\ timer(tx, 10) \\
&\textbf{if}\ \text{true} && (4)\\
timeout(tx) \ &\textbf{initiates}\ delete\_chain(x) \\
&\textbf{if}\ chain(x) && (5)
\end{aligned}
$$

Fig. 2: ECA Security Policy Examples

4) Rule Execution: The run-time behavior of the system depends on how ECA rules are executed. More specifically, (i) how conditions are monitored and evaluated; (ii) what is the relative timing of executing the components of an ECA rule; and (iii) how rules are scheduled when an event triggers multiple rules, multiple events occur simultaneously, or a rule triggers other events that invoke other rules. For (i), a process checks the validity of a defined condition. *Coupling modes* [35] describe different timing strategies to deal with (ii). For (iii), the orchestrator maintains the list of fired events ordered based on their *priorities* and *occurrence time*. For more details, we refer the reader to existing work on active databases [35], [23], [21]. Finally, to execute actions introduced in Section III-B2, the orchestrator translates these declarations to actual VIM API calls discussed in Section III-C3.

C. Virtual Infrastructure Manager

Virtual Infrastructure Manager (VIM) is responsible to manage host resources, deploy, and manage security function chains. VIM provides an API to create/delete a chain, insert/delete a function to/from a chain, and query information about deployed chains. This API is used by the orchestrator to manage security services, and by SMAS to query about deployed functions. We leverage the following general purpose mechanisms and tools in the implementation of VIM:

Docker. Containers have a low resource overhead and are fast to create and destroy. VIM utilizes Docker [36] to manage container-based functions.

Network Service Header (NSH). NSH is a modern service plane protocol for dynamic service function chaining [40]. NSH specifies a sequence of functions through which packets are steered before reaching the destination address. NSH is independent of the underlying transport protocol. Further, it can carry metadata that can be exploited for more sophisticated chain operations. NSH is a widely accepted industry standard.

Open Virtual Switch. VIM implements the networking aspect of service function chaining using Open Virtual Switch (OVS) [6]. OVS operates at the kernel level and achieves fast, and constant-time traffic forwarding with very low overhead. We use NSH rules to forward traffic between functions.

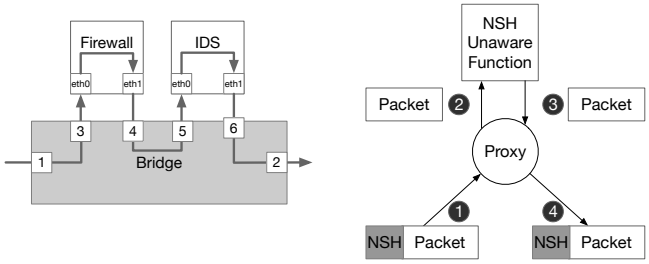
1) Specifications: The following are used in calling the VIM's API.

Chain Specification. VIM uses the specification depicted in Fig. 3 where `chain_name` specifies ch to be the unique name of this chain. As mentioned in Section III-B2, traffic traverses a virtual network connecting functions. `ingress` and `egress` respectively denote from which *point* in this network traffic enters the chain and to which point the traffic is forwarded after the chain process completes. A point in the network can be the Network Interface Cards (NICs) of a virtual edge-server, an explicit OVS port, or a deployed function's ingress or egress NICs. In Section IV, we use this powerful notation to compose chains. `classification_rules` serves as a traffic filter applied on the `ingress`. This field specifies which traffic subset from `ingress` is forwarded to the chain. Two chains cannot have the same `ingress` and `classification_rules`. Field `functions` denote the sequence of functions in the chain.

Function Specification. VIM instantiates a function based on three fields as follows. `function_image` specifies the Docker image. `function_name` is a unique name for the function. Each function in a chain or across chains must have a unique `function_name`. Referring to this field, a function can be shared among multiple chains. Finally, field `nsh_aware` is used for compatibility with legacy functions and states whether the function can parse NSH header.

```
{
  "chain_name": "ch",
  "ingress": "1",
  "egress": "2",
  "classification_rules": "ip",
  "functions": [
    {
      "function_image": "Firewall",
      "function_name": "firewall",
      "nsh_aware": false
    },
    {
      "function_image": "IDS",
      "function_name": "ids",
      "nsh_aware": false
    }
  ]
}
```

Fig. 3: The Specification of a Chain and its Functions



(a) A Deployed Chain (b) Function-Proxy

Fig. 4: Service Function Chaining

2) Service Function Chaining: Functions are deployed based on the function specification using Docker. VIM creates and configures an *OVS bridge* which acts as the networking medium between functions. VIM connects each function to the OVS bridge by creating a `veth-pair`. One side of this `veth-pair` is attached to the OVS bridge, and the other side is connected to the container. Fig. 4a illustrates the deployment of the chain defined in Fig. 3.

Three sets of rules are inserted to steer traffic through the chain. i) *Classification rules* filter incoming packets from `ingress` based on `classification_rules` and attach NSH header to these packets. ii) *Forwarding rules* are NSH-based match/action rules that forward packets between functions. In packet forwarding based on NSH, functions have to participate in forwarding by modifying the NSH header. In the case of NSH-unaware functions, a *function-proxy* parses and performs NSH-based forwarding actions. VIM implements this proxying using a third set of rules as shown in Fig. 4b. iii) *Proxy rules* match and remove the NSH header before forwarding packets to an NSH-unaware function. After receiving from the NSH-unaware function, the appropriate NSH header will be reattached to packets by proxy rules.

3) Northbound API: VIM provides the API shown in Fig. 5. Arguments `chain_sp` and `func_sp` are respectively the specifications of a chain and a function and must follow the specifications presented in Section III-C1. The first 5 methods correspond to actions defined in Section III-B2. The others

```
def create_chain(chain_sp)
def delete_chain(chain_name)
def insert(chain_name, func_sp)
def delete(chain_name, func_name)
def run(func_name,cmd)
def chains()
def chain(chain_name)
def chain_functions(chain_name)
def functions()
def function(func_name)
def steered(bpf,chain_name)
```

Fig. 5: VIM API

TABLE I: Resource Statistics

Bandwidth	CPU	Memory	Storage
Per-NIC util.	Total util.	Pages-ins/outs	Free space
Bytes rec./sent	Per-core util.	Swap-ins/outs	Transfer per sec.
Packets rec./sent	Sys./user modes util.		Read/write per sec.
Packet drops	Context switches		
	Interrupts and IOs		

are query methods about chains, functions, and traffic used by SMAS and the orchestrator.

D. Security Monitoring Analytics System

Security Monitoring Analytics System (SMAS) is responsible for monitoring the logs of deployed functions and resources of a virtual edge-server to collect important metrics, analyzing these monitored data, and generating security alerts to inform the orchestrator. In the current implementation of our system, we focus on monitoring and analyzing the resources of the virtual edge-server to handle misuse attacks.

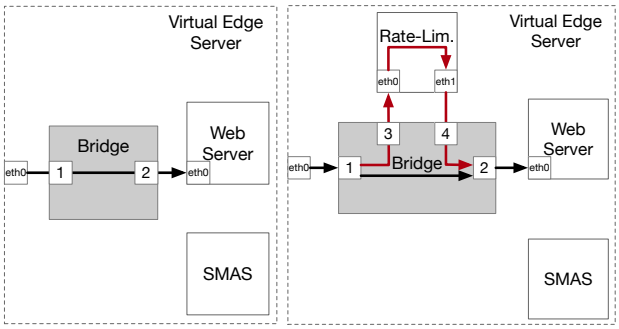
SMAS periodically monitors and collects statistics on *network-bandwidth*, *CPU*, *memory*, and *storage* resources. Our implementation relies on Linux standard tools, such as `/proc/stat` file, `free` command, and `iostat` command for data collection. Typically, when the value of a relevant metric passes some predefined threshold, SMAS generates an alert indicating that this value is either over or under the threshold. For instance, if the power utilization is above a predefined threshold or is under another predefined threshold, SMAS generates $high\_cpu$ or low_cpu, respectively.

The statistics collected for each resource are listed in Table I. To decide which statistic to collect, we carefully select the metrics that do not require high monitoring overhead. We also select metrics that provide immediate and rewarding information. For instance, SMAS does not monitor the average file size, since it is an expensive process; in turn SMAS monitors *page-ins* and *page-outs* whose high rates mean that the memory is short, or the system is spending more resources moving pages than running actual applications.

IV. Use-case Scenarios

A. Rate Limiting Use-case

Rate limiting is a common practice [13], [18], [15] that CDNs use against threats ranging from network layer attacks, e.g. DDoS, to application layer attacks, e.g. brute-force login



(a) Monitoring Stage (b) Rate Limiting Stage

Fig. 6: Rate Limiting Scenario

attempts. Various rate-limiting mechanisms exist, such as limiting traffic-rate per user, geography, or server. In this use-case, traffic is rate-limited per-user. Fig. 6 illustrates this scenario, and Fig. 7 lists the applicable security policies.

Monitoring Stage. Fig. 6a shows the initial system deployment. At the beginning, SMAS performs light resource monitoring of the virtual edge-server. Large traffic volume causes high bandwidth and CPU consumption. SMAS identifies this suspicious behavior as bandwidth and CPU are consumed beyond certain thresholds. SMAS raises an alert, $high\_rate$, to notify the orchestrator regarding this suspicious traffic.

Rate Limiting Stage. Based on Rules 6-8, upon receiving the alert $high\_rate$, if no rate-limiting service exists, the system deploys chain r containing a $Rate\text{-}limit$ to limit the traffic-rate per IP (representing per end-user traffic). A white-list of IP addresses are exempted from rate-limiting. Fig. 6b shows this chain. To enforce Rule 9, a timer starts after the installation of the chain for the predefined period of time d. Upon the expiry of this timer, a timeout event is generated with a parameter tr. Finally, upon receiving the timeout event carrying tr parameter, Rules 10 and 11 are matched. First, executing Rule 10, chain n with no function is deployed. As chain n connects ports 1 and 2, traffic is forwarded to the Web-server. Then, Rule 11 is matched, and chain r is removed.

B. Mitigating HTTPS DDoS Use-case

HTTPS DDoS attacks exploit HTTP and HTTPS and target Web applications running on a server [25], [44]. Such attacks usually generate less traffic and use seamingly legitimate requests, and are, therefore, harder to detect. CDNs commonly utilize *Web Application Firewalls* (WAFs) to mitigate these attacks [14], [17]. Inspection at the application layer is a heavy process that can affect the application response time [9]. In this use-case, our system deploys a security service to mitigate HTTPS DDoS attacks. This service inspects the content of suspicious traffic to mitigate the attack, while legitimate traffic is served directly without inspection. Fig. 8 depicts this use-case scenario, and Fig. 9 lists ECA policies enforced.

$high\_rate$ **initiates** $create\_chain(r$:
<"not src net 129.97.124.0/24", 1, 2>,
$\{f$:$Rate\text{-}limit\}$)
if not $chain(r)$ (6)

lim **after** $create\_chain(r)$
if true (7)

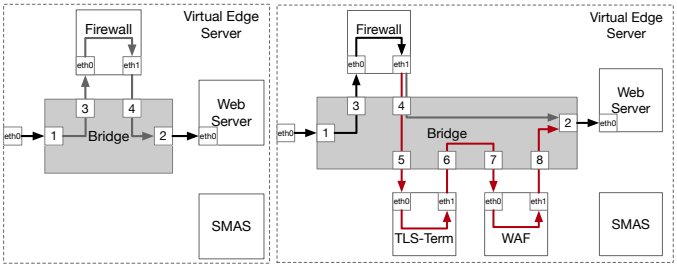
lim **initiates** $run(f$, "$rate\_limit.sh$")
if true (8)

$create\_chain(r)$ **causes** $timer(tr, d)$
if true (9)

$timeout(tr)$ **initiates** $create\_chain(n$:<_, 1, 2>, {})
if true (10)

$timeout(tr)$ **initiates** $delete\_chain(r)$
if $chain(r)$ (11)

Fig. 7: Rate Limiting Policies



(a) L3 Mitigation Stage (b) L4 Mitigation Stage

Fig. 8: Mitigating HTTPS DDoS Scenario

cpu_high **initiates** $create\_chain(u$:<"ip", 1, 2>,
$\{f$:$Firewall\}$)
if not $chain(u)$ (12)

$block$ **after** $create\_chain(u)$
if true (13)

$block$ **initiates** $run(f$, "$block.sh$")
if true (14)

$create\_chain(u)$ **causes** $timer(td, d)$
if true (15)

$timeout(td)$ **initiates** $create\_chain(l$:
"not src net 99.231.0.0/16", f, 2,
$\{t : TLS\text{-}Term, w : WAF\}$)
if not $chain(l)$ and $chain(u)$ (16)

cpu_low **initiates** $create\_chain(n$:<_, 1, 2>, {})
if true (17)

cpu_low **initiates** $delete\_chain(u)$
if $chain(u)$ (18)

cpu_low **initiates** $delete\_chain(l)$
if $chain(l)$ (19)

Fig. 9: HTTPS DDoS Mitigation Policies

L3 Mitigation Stage. An HTTPS DDoS attack exhausts the CPU power of the virtual edge-server. SMAS generates cpu_high alert to notify the orchestrator that CPU is consumed beyond a predefined threshold. Upon reception of this alert to enforce Rule 12, the system instantiates chain u composed of a $Firewall$ named f, as shown in Fig. 8a. Chain u processes IP traffic coming from port 1, going to 2 (the ingress of the Web-server). This chain starts to filter non-HTTPS traffic (Rules 13 and 14); however, since the attack targets the application layer, CPU load is still high.

L4 Mitigation Stage. Upon creating chain u, a timer starts to count (Rule 15). When this timer expires, another chain l comprising a $TLS\text{-}Term$ (a TLS termination) and a WAF is instantiated to perform mitigation at the application layer (Rule 16). Fig. 8b depicts this deployment. Chain l processes a subset of traffic coming out of function f, going to the Web-server. Note that legitimate traffic, i.e. originating from a *white-list* of source IP-addresses in range $99.231.0.0/16$, is still directly steered to the Web-server, while the rest of the traffic, i.e. suspicious traffic, is steered through chain l. $TLS\text{-}Term$ decrypts suspicious traffic, and WAF inspects plain-text traffic to mitigate application layer attacks including HTTPS DDoS. If the CPU utilization drops under a predefined threshold, the traffic is directly forwarded to the Web-server, and both chains u and l are deleted (Rules 17-19).

V. Performance Evaluation

A. Experimental Platform

Testbed. We use a cluster of machines (16GB RAM, 8-cores 3.30GHz Xeon CPUs) connected with 10 Gbps NICs. The servers run Ubuntu 14.04 with Linux kernel version 3.16. We use 1 to 4 servers as load generators, a server as the Device under Test (DuT) to host chains, and a server as the traffic sink. An active daemon of our system runs on DuT.

Traffic generation. We use `iperf` and `Apache benchmark` (`ab`) to generate line-rate TCP and Web traffic, respectively. `iperf` clients and `ab` run on the load generator servers, and `iperf` server runs on the traffic sink server.

Service functions. We use two service functions. Function `fwd` passes traffic from a virtual interface to another. We intentionally use this function in experiments in which we benchmark the overhead of our service function chaining platform independent from the complex functionality of a service function. The other function is `Rate-limit` which limits the rate of the incoming traffic.

B. System in Action

We measure the overhead of deploying chains using our system, and the overhead of our chaining mechanisms in terms of latency and throughput.

1) Chain Deployment Time: This experiment measures the time it takes to deploy a chain using our system. We vary the chain length (the number of functions in a chain) from 1 to 7 and repeat each experiment 5 times. In the process of creating a chain, instantiating functions and connecting them to OVS are the two most time-consuming procedures. As shown in Fig. 10, the chain of length 1 has the lowest deployment time of 1.06 s, and the chain of length 7 has the highest deployment time of 6.13 s. The VM-based platforms (e.g. Bohatei [27]) have a chain creation time in the order of minutes, while it is evident from this experiment that our system is capable of deploying service function chains in less than 7 seconds.

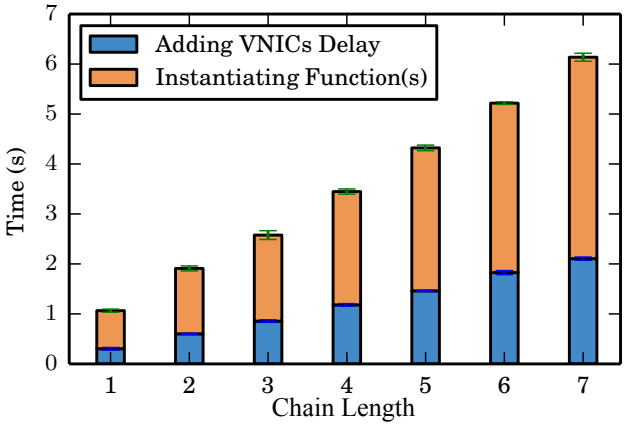


Fig. 10: Chain Deployment Time

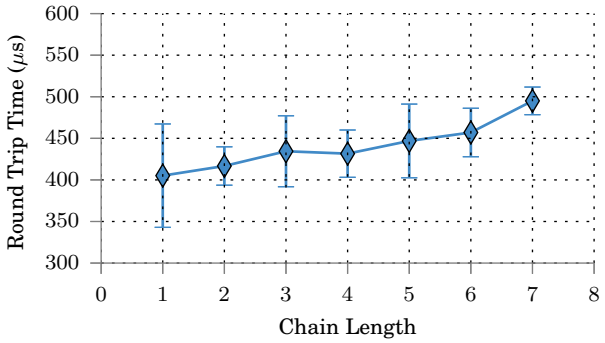


Fig. 11: Traffic Round Trip Time

2) Round Trip Time: In this experiment, we measure the Round Trip Time (RTT) of traffic steered through chains deployed by our system. We use `ping` for the RTT measurements, and repeat each experiment 5 times. As depicted in Fig. 11, we vary the chain length from 1 to 7 `fwd` functions, and report the RTT average and standard-deviation for each chain-length. As expected, the chains of length 1 and 7 have the lowest RTT (405.13 μs) and the highest RTT (495.04 μs), respectively. Although the longer the chain, the higher the RTT, the delay introduced by our routing mechanism is small. As shown, the RTT of the chain of length 7 is only 89.91 μs more than that of the chain of length 1.

3) Resource Utilization and Throughput: In this experiment, we measure the maximum throughput of chains composed of 1 to 7 `fwd` functions using `iperf`. We repeat each experiment 5 times. All functions of a chain are instantiated in a single server. As shown in Fig. 12, the chain-length has a

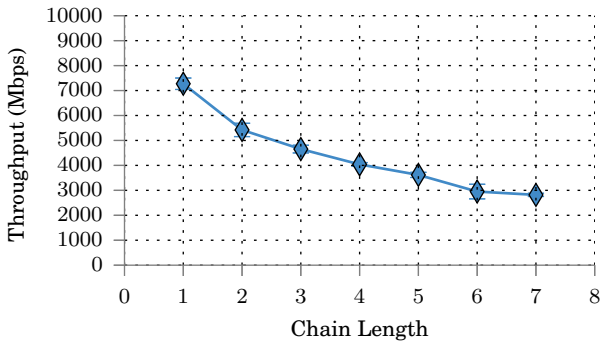


Fig. 12: Throughput vs. Chain Length

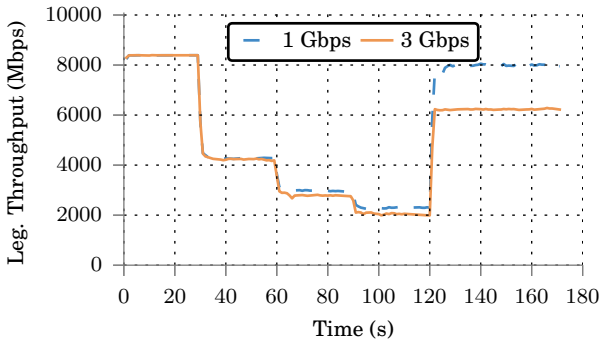


Fig. 13: Recovering Legitimate Traffic Throughput

direct impact on the chain throughput. The chains of length 1 and 7 have respectively the highest throughput (7272 Mbps) and the lowest throughput (2818 Mbps) on average. All cycles of CPU-cores are utilized during this experiment. We observe that `fwd` functions consume a negligible amount of the CPU power, while the process `softirq` consumes the most of CPU power meaning that the packet reception in the Linux kernel of the host becomes the bottleneck. The workflow of the packet reception in the Linux kernel (version 2.5.7 and above) is as follows. The NIC transfers a packet from the *ring* buffer to the main memory via direct memory access and notifies the CPU with *input queue* interrupt request (IRQ). This IRQ is mapped to a CPU core which runs Interrupt Service Routine (ISR) [8] to handle this interrupt. At the end, ISR raises a `softirq` to defer the reception of the packet from the interrupt context to the process context. Packet reception is an expensive process. In a chain, each function generates IRQs by forwarding packets. Making the chain longer increases the number of IRQs, thus doing so decreases the throughput.

C. Responsiveness

In this experiment, we evaluate the effectiveness of our system in performing traffic engineering actions similar to the use-case scenario presented in Section IV-A. We use our system to recover the QoS of legitimate traffic in the case of a flooding attack. To emulate such a scenario, we perform a five-stage experiment summarized in Table II. As shown in Fig. 13, we start by sending only legitimate traffic (8.39 Gbps) in the first stage. In the next three stages, the flooding traffic is gradually increased to drain the network bandwidth and decrease the legitimate traffic throughput. During these stages, the legitimate traffic experiences the throughput of ~ 8.4 Gbps down to ~ 2 Gbps. In the last stage, in response to the generated alert, our system deploys a mitigation chain consisting of a `Rate-limit` function through which the flooding traffic is steered. The flooding traffic is limited to two different rates (1 Gbps and 3 Gbps). In the case of 1 Gbps rate-limit, the legitimate throughput is almost fully recovered (8.02 Gbps). We observe a recovered throughput of 6 Gbps in the case of 3 Gbps rate-limit. In both cases, we achieve immediate recovery (in less than 1 second) after deploying the mitigation chain. These results demonstrate that our system provides fast and effective recovery of the legitimate traffic throughput.

TABLE II: The Stages of Responsiveness Experiment

Stage	Duration (s)	Flooding traffic share
1	0-30	0%
2	30-60	50%
3	60-90	66.6%
4	90-120	75%
5	120-170	Limited to 1 Gpbs / 3 Gbps

D. Static vs. Dynamic Security Service

Our system allows agile deployment of security services and the ability to redirect subsets of traffic on-the-fly. These features make it easy to deploy security chains to process a subset of traffic. In this experiment, we compare a static service with a dynamic one securing a Web-server. These security services perform deep inspection of incoming HTTPS traffic. The traffic passes through a TLS termination (decrypting HTTPS to HTTP) and a WAF (deep inspection of HTTP traffic matching OWASP core-rules-set [16]). In addition to security functions, a Web-server is installed in the DuT.

In this experiment, we measure the time to download 400 Web-pages concurrently. The static security service (`Static`) corresponds to the manual deployment of the security chain through which *all* requests are steered. In the dynamic security service (`Dynamic`), 300 legitimate requests are directly served without any inspection, while 100 suspicious requests are analyzed in the security chain. As shown in Fig. 14, all pages are retrieved within 104.205 ms in the case of `Static`. In the case of `Dynamic`, the legitimate requests are retrieved in less than 43.09 ms, and others are served within 110.309 ms. The dynamic security service serves the legitimate requests $2.4\times$ faster. To further evaluate the overhead introduced by security services, we measure the completion time of requests when all requests are directly served (`Baseline`). In the case of `Baseline`, all Web-pages are downloaded within 10.496 ms. One would think that legitimate requests in the case

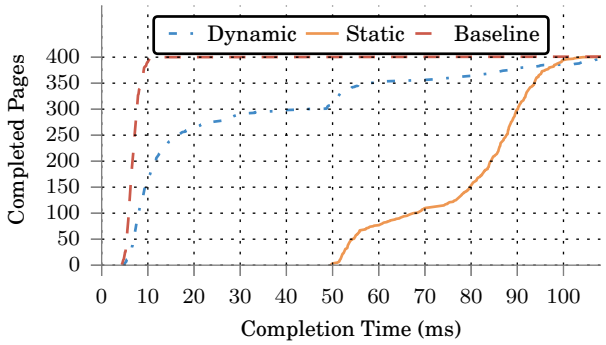


Fig. 14: Completion Time of Retrieving Web-pages

of `Dynamic` should be served with the same latency as in the case of `Baseline`. However, there is an overhead of 32.6 ms explained by the high CPU utilization (mostly consumed by the WAF) when security chains inspect traffic. Consequently, the Web-server is deprived of some of the CPU resources.

VI. Conclusion

In this work, we designed and implemented a policy-based security system that automatically and dynamically deploys security function chains. We illustrated how our system can be flexibly programmed to handle real world use-cases. The evaluation results demonstrated that our system has low overhead in terms of chain deployment time and latency of traffic passing through a chain. This system is able to immediately respond to threats and quickly recover legitimate traffic throughput ($\sim$1 second). Further, our system is capable of on-the-fly traffic redirection. Using our system, legitimate traffic can be exempted from the high overhead imposed by heavy security services. To do so, security policies should dictate redirecting only suspicious traffic to security chains while legitimate traffic is directly served without inspection. Using this capability, we have shown that legitimate requests are served 2.4$\times$ faster than than in the case of static security services.

References

[1] Akamai's [state of the internet] / security q3 2016 report. https://goo.gl/pliMHT.

[2] Cdn hosting vs traditional web hosting. https://goo.gl/ynnku2.

[3] Fastly ddos mitigation. https://goo.gl/wwco7U.

[4] Global ddos threat landscape q3 2016. https://goo.gl/kgT6pM.

[5] Netflix open connect. https://openconnect.netflix.com.

[6] Open vswitch. http://openvswitch.org.

[7] What to look for when choosing a cdn for ddos protection. https://goo.gl/HiaCA9.

[8] Assign interrupts to processor cores on intel ethernet controller. https://goo.gl/nWXjzU, 2009.

[9] Ways to improve performance of your server in modsecurity 2.5. https://goo.gl/EdRzJR, 2009.

[10] Sandvine global internet phenomena report 2h-2013. https://goo.gl/GWqQWV, 2013.

[11] Network functions virtualisation (nfv); management and orchestration. https://goo.gl/wRm9LK, 2014.

[12] Network functions virtualisation (nfv) release 3; security; security management and monitoring specification. https://goo.gl/hQMXNP, 2014.

[13] Cloudflare rate limiting. https://goo.gl/PovNvK, 2017.

[14] Ddos prevention: Ddos protection product | defencepro. https://goo.gl/FBazjJ, 2017.

[15] Defend http rate limiting: Stop application layer ddos attacks at the edge of the internet. https://goo.gl/UajPrT, 2017.

[16] Owasp modsecurity core rule set project. https://goo.gl/ihxX98, 2017.

[17] Web application ddos protection | layer 3-4 and 7 | incapsula. https://goo.gl/Dkdbct, 2017.

[18] What is rate limiting? https://goo.gl/HxWRC9, 2017.

[19] T. Alharbi, A. Aljuhani, and H. Liu. Holistic ddos mitigation using nfv. In *2017 IEEE CCWC*, 2017.

[20] B. Anwer, T. Benson, N. Feamster, and D. Levin. Programming slick network functions. In *ACM SOSR*. ACM, 2015.

[21] C. Baral, J. Lobo, and G. Trajcevski. *Formal characterizations of active databases: Part II*, pages 247–264. Springer Berlin Heidelberg, 1997.

[22] A. Bremler-Barr, Y. Harchol, and D. Hay. Openbox: A software-defined framework for developing, deploying, and managing network functions. In *ACM SIGCOMM*, 2016.

[23] M. J. Carey, R. Jauhari, and M. Livny. On transaction boundaries in active databases: a performance perspective. *IEEE TKDE*, 1991.

[24] J. Chen, T. Wan, and V. Paxson. Forwarding-loop attacks in content delivery networks. In *the 23st Annual Network and Distributed System Security Symposium*, 2016.

[25] Y. G. Dantas, V. Nigam, and I. E. Fonseca. A selective defense for application layer ddos attacks. In *IEEE JISIC*, 2014.

[26] U. Dayal, A. P. Buchmann, and D. R. McCarthy. *Rules are objects too: A knowledge model for an active, object-oriented database system*. Springer, 1988.

[27] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic ddos defense. In *USENIX Conference on Security Symposium*. USENIX Association, 2015.

[28] A. Gerber and R. Doverspike. Traffic types and growth in backbone networks. In *OSA OFC and NFOEC*, 2011.

[29] D. Gillman, Y. Lin, B. Maggs, and R. K. Sitaraman. Protecting websites from attack with secure delivery networks. *Computer*, 2015.

[30] W. Han and C. Lei. A survey on policy languages in network and security management. *Computer Networks*, 2012.

[31] A. H. M. Jakaria, W. Yang, B. Rashidi, C. Fung, and M. A. Rahman. Vfence: A defense against distributed denial of service attacks using network function virtualization. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016.

[32] J. Li, S. Berg, M. Zhang, P. Reiher, and T. Wei. Drawbridge: Software-defined ddos-resistant traffic engineering. In *ACM*. ACM, 2014.

[33] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On dominant characteristics of residential broadband internet traffic. In *ACM IMC*. ACM, 2009.

[34] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX*, 1993.

[35] D. McCarthy and U. Dayal. The architecture of an active database management system. *SIGMOD Rec.*, 18(2):215–224, June 1989.

[36] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, page 2, 2014.

[37] T. Moses et al. Extensible access control markup language (xacml) version 2.0. *Oasis Standard*, 200502, 2005.

[38] I. Poese, B. Frank, B. Ager, G. Smaragdakis, and A. Feldmann. Improving content delivery using provider-aided distance information. In *ACM IMC*. ACM, 2010.

[39] M. Prince. Technical details behind a 400gbps ntp amplification ddos attack. https://goo.gl/5Fn84x.

[40] P. Quinn, U. Elzur, and C. Pignataro. Network Service Header (NSH). Internet-Draft draft-ietf-sfc-nsh-28, IETF, 2017. Work in Progress.

[41] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *NSDI*. USENIX, 2012.

[42] S. Triukose, Z. Al-Qudah, and M. Rabinovich. Content delivery networks: protection or threat? In *ESORICS*. Springer, 2009.

[43] C. Wueest. The continued rise of ddos attacks. https://goo.gl/EuLPr2, 2013.

[44] Y. Xie and S.-Z. Yu. Monitoring the application-layer ddos attacks for popular websites. *IEEE/ACM Trans. Netw.*, 2009.

[45] M. Yu, Y. Zhang, J. Mirkovic, and A. Alwabel. Senss: Software defined security service. In *ONS*, Santa Clara, CA, 2014. USENIX.