Dynamic Security Orchestration System Leveraging Machine Learning

by

Elaheh Jalalpour

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

Statement of Contributions

Chapter 3 and Chapter 4 borrow content from two papers "A Security Orchestration System for CDN Edge Servers" [72] and "Dynamic Security Orchestration for CDN Edge-Servers" [71].

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. Attacks are becoming more sophisticated, and new attacks are being introduced. Protecting edge-servers in the face of these attacks 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 detect and mitigate various attacks. The approach is to detect threats and automatically react by deploying and managing security services. The desired system behavior is governed by high-level security policies dictated by a network operator. Leveraging advanced machine learning techniques, our system can detect new and sophisticated attacks and generate alerts that trigger policies. Policy enforcement can result in the deployment of mitigation services realized using virtualized security function chains created, configured, and removed dynamically. We demonstrate how our system can be programmed using these policies to automatically handle real-world attacks. Our evaluation shows that our system not only detects known sophisticated attacks accurately but is capable of detecting new attacks. Moreover, the results show that our system is low-overhead, immediately responds to threats, and quickly recovers legitimate traffic throughput.

Acknowledgements

I would like to express my sincere gratitude and appreciation to my supervisor, Professor Raouf Boutaba, for his continuous support throughout the course of my Masters studies. I've been so blessed to have a wonderful source of knowledge with extensive experience. Thank you for your enthusiasm, inspiration, and patience.

I would like to extend a special thank to my thesis committee members for their constructive and valuable feedback: Bernard Wong and Samer Al Kiswany.

I am so grateful to Milad Ghaznavi, my colleague, lab-mate, and great friend, for always patiently motivating me to work hard and guiding me when I was lost. I would also thank my fellow coworkers in the Network Lab for their feedback, cooperation and of course friendship, with a special mention to Ali Abedi.

I would also like to thank Stere Preda, Daniel Migault, and Makan Pourzandi, my fellow coworkers at Ericsson, for their valuable input.

I am truly grateful to my parents Fahimeh Mirrokni and Mohammad Jalalpour, and to my brother Amin Jalalpour, for their unconditional love, support, and sacrifices during my studies. I thank Saeed, my dearest, for his tremendous support.

This work benefitted from the use of RIPPLE and SYN Facilities at the University of Waterloo.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Delivering digital content (e.g., video, images, and Web-pages) accounts for most of today's Internet traffic [82, 92, 63]. CDNs play a critical role in delivering digital content to end-users. Open-Connect [15] carries part of Netflix's content accounting for 35.2% of all the traffic across North America, and daily Akamai CDN delivers more than 30 Tbps of traffic [35]. 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 [65, 105, 2]. DDoS attacks exhaust the resources of an edge-server. They range from network flooding (e.g., UDP fragment, SYN flood) [30] to amplification/reflection [94], and HTTP/S flooding [8]. Most CDNs host Web services [3] 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 [65]). Moreover, new attacks are being introduced every day (e.g., forwarding-loop attacks [53]).

Securing edge-servers against these main attack vectors is a complex task. Security services must be immediate in responding 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.

1.1 Challenges and Opportunities

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 new 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 capital expenditure (CAPEX) and operational expenditure (OPEX). Security attacks evolve rapidly [107], 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 CDN providers 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 security 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 [60, 70, 44] 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 thesis 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 the industry.

1.2 Contributions

Our main contributions in this thesis are summarized below:

- We design and implement a *dynamic* and *automatic* security orchestration system for protecting CDN edge-servers. We demonstrate how our system can be flexibly programmed using high-level policies to handle real world use-cases (Chapter 3). We demonstrate that the orchestration 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 (Chapter 4).

- We propose a Hybrid Classification Clustering (HCC) method that not only detects known sophisticated attacks accurately but is capable of detecting new attacks. Further, to improve the detection rate of new attacks and anomalies, we propose an Autoencoder-based Network Anomaly Detection (ANAD) method using a fully-connected autoencoder model (Chapter 3). Performance evaluation results show that our system using HCC method is accurate in detecting known attacks (with 99.9% detection recall) and is capable of detecting new attacks (with 56.4% detection recall). Using ANAD method, the system can achieve 76.7% recall in detecting anomalies (Chapter 4).

1.3 Thesis Organization

The remainder of this thesis is organized as follows. We provide a background of content delivery networks and we discuss the related work trying to secure them in Chapter 2. In Chapter 3, we present the design and implementation of our security orchestration system. An extensive evaluation of our system is presented in Chapter 4. We conclude in Chapter 5 by summarizing the thesis and outlining some possible future work to improve the system.

Chapter 2

Background

2.1 Content Delivery Networks

A CDN aims at enhancing the quality of experience in delivering digital content to end-users while utilizing network resources more efficiently. A CDN caches content in multiple locations nearby end-users, routes a content requests to a location in which the requested content is cached, and transfers this content to end-users [51, 75, 89].

Fig. 2.1 shows the main parties involved in the content delivery procedure [89]. A *CDN provider* manages and operates the CDN infrastructure. The CDN provider supports services, such as the delivery of static content (e.g., videos and images) and dynamic content (e.g., dynamic web pages), and streaming (e.g., real time video streaming). A *content owner* owns digital content. Content owners establish agreements with CDN providers and delegate the delivery of their content to CDNs. An *end-user* consumes digital content by requesting such content through their digital devices: such as TVs, tablets, and smart phones.

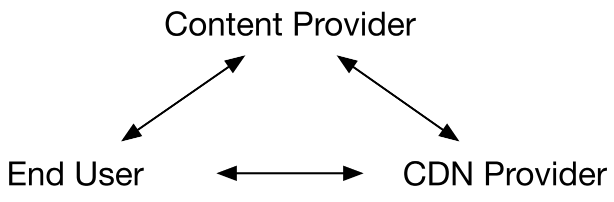


Figure 2.1: Main entities involved in content delivery

2.1.1 Content Delivery Procedure

A CDN is an infrastructure consisting of numerous servers (e.g., hundreds to thousands of servers [88]) which are distributed across the Internet to provide high-capacity storage capability to cache content close to end-users. Fig. 2.2 depicts content delivery through a CDN. A *content owner* places digital content in *origin servers*. The *CDN provider* distributes and replicates the content from origin servers into *edge servers* in the vicinity of end-users [88]. Fig. 2.2 illustrates how content delivery is performed, as follows. When an *end-user* requests a content, a CDN on behalf of a distant content owner serves this request (Step 1). Note that the CDN infrastructure is generally transparent to end-users. Using a *request routing mechanism*, the CDN selects and redirects a legitimate request to one of its edge servers (Step 2). The selected server performs an admission control, and if the request is accepted, the server delivers the content from its *cache* (Step 3) [55]. On cache misses, an edge server retrieves and forwards the content from either another edge server or *origin server* (Step 4).

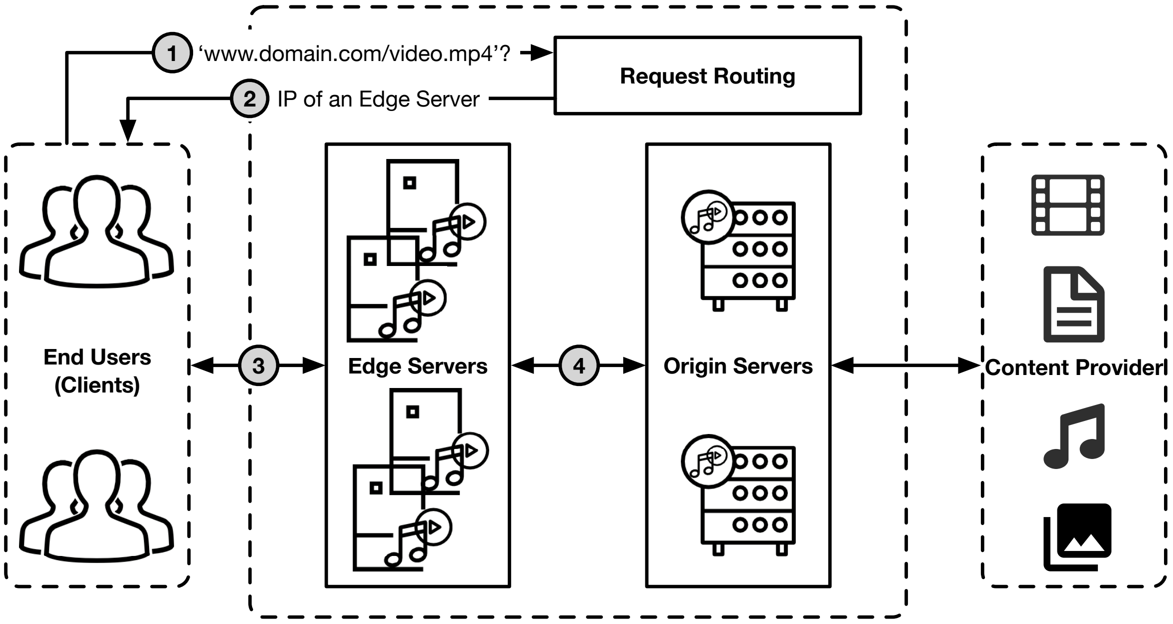


Figure 2.2: Content delivery through a CDN

2.1.2 Attacks against CDN Edge-Servers

In this section, we review security attacks on edge-servers and countermeasures. We start by common application layer attacks, and then we focus on DDoS attacks.

Application Layer

Common Application Layer Security Challenges. Most CDNs serve Web traffic, and cache the Web content at edge servers. Serving Web applications can cause edge servers to be prone to Web and application layer attacks. Common attacks include SQL injection (SQLi), cross-site scripting (XSS), file inclusion, remote command execution, and illegal resource access. Focusing on the damages caused to the victim organizations, the authors of [28] classify threats into three categories: i) *hacking* by SQLi and XSS, causing data breaches; ii) *business abuse* via screen scrapping, spamming; and fake accounts, with a wide range of negative impacts from business malfunction and links to malware to fraud; and iii) *denial of service*, via DoS and DDoS attacks on applications, potentially interrupting business. The authors of *CoDeen* [106], an academic CDN testbed, list several threats inherent to Web deployments including spamming and bandwidth abuse.

Countermeasures. Static and dynamic analyses on the source code of Web applications can reveal some vulnerabilities and security flaws [104, 9]; however, not all possible threats can be prevented. *CoDeen* [106] employs rate limiting and blacklisting. Furthermore, Web Application Firewalls (WAFs), which are installed in or at the front of edge servers, protect edge servers and origin-servers against common Web application attacks. A WAF performs a deep inspection of every Web request and response in order to detect and block common Web application layer attacks [93]. Some commercial WAFs from CDNs protect Web applications against the Open Web Application Security Project (OWASP) top 10 security attacks [26], for instance the Alibaba WAF [54]. Other WAFs, for example the Incapsula WAF [29], are designed to be compliant with Payment Card Industry Data Security Standard or Health Insurance Portability and Accountability Act standards.

Denial of Service

In a Denial of Service (DoS) attack, "an attacker attempts to prevent legitimate users from accessing information or services" [85]. For example, an attacker can flood a network to consume key resources and make them unavailable to legitimate users. An attacker can also send a number of malicious packets to a victim machine to confuse a protocol or application and cause the machine to reboot or freeze. A Distributed Denial of Service (DDoS) attack employs multiple attacking entities to achieve the goal. DDoS attacks cause widespread congestion, jam crucial infrastructure, and cost target businesses loss of considerable revenue from disrupted services.

Recent DDoS Trends. Historically, flooding attacks (e.g., UDP fragment, SYN flood) have been the most common DDoS attacks; however, these attacks recently became less popular [30], while amplification and reflection are becoming more popular. Using amplification/reflection techniques, attackers launch massive DDoS attacks on the scale of hundreds of Gbps [94, 1, 5, 95]. Application layer DDoS attacks are rising in popularity and complexity [8, 30, 31]. Moreover, botnets such as Mirai leveraging the increasing deployment of IoT devices are becoming the major sources of DDoS attacks [5]. *Multi-vector* DDoS attacks employ different attack strategies to assault different layers. These attacks can last for days or weeks, resembling *advanced persistent threats* [103]. The aim of these DDoS attacks is to take down the security applications (e.g., WAFs) or cover other malicious activities, such as stealing data [11].

Countermeasures. Common defense mechanisms include traffic-blocking and rate-limiting. CDNs block traffic based on geographical location, black-listing, white-listing, source of traffic, or traffic behavior, while rate limiting restricts the rate of traffic to a configurable threshold per IP, request type, or other metrics. In this way, the traffic volume that a bot can generate is limited. CDNs usually provide *always-on* and *on-demand* DDoS defenses [30, 4]. Always-on solutions use inline *scrubbers* that constantly monitor traffic and stop suspicious traffic at edge servers. For instance, at edge servers, identifiable malicious DNS amplification or SYN flood packets are filtered [10]. Some CDNs only allow a whitelist of traffic types (for instance, DNS, HTTP/HTTPS, media streaming) and block other traffics [13]. Moreover, the defense distinguishes between the traffic generated by humans and good and bad bots. Doing so is a key factor in identifying application layer DDoS attacks [10]. Another way of filtering the traffic generated by a bad bot is to use the *IP reputation* mechanism, which generates a signature for the bot, and if the signature is matched, the traffic is blocked. On-demand solutions redirect traffic using dynamic BGP changes and DNS redirection to scrubbing centers. BGP changes route the traffic belonging to *routable* addresses (e.g., the IP addresses controlled by a CDN) to scrubbing centers. BGP changes require a sufficient range of IP addresses (e.g., more than 256 addresses) to be eligible for being *routable*. Doing so can shut down even very large-scale layer 3 and 4 DDoS attacks [12]; however, it might slow down legitimate requests. The effectiveness of such mechanisms depends on how long it takes for a DDoS attack to be detected and for the operator to react. Therefore, CDN providers also use monitoring service for detecting network anomalies. On-demand defenses are effective against SNMPbased and SlowLoris attacks; however, they are not aware of layer-7 information and can miss application layer attacks.

DDoS against known edge servers. By knowing the IP-addresses of edge servers, attackers can launch denial of service attacks against these servers [91]. Attackers employ a set of bots to target an edge server and exhaust the server's resources. To hide the identities of bots, their IP-addresses are generally spoofed.

Countermeasures. Lee et al. [77] proposed deterring such flooding attacks at the request routing and edge servers. The basic idea is to serve requests that are redirected by request routing components with a higher priority in order to save the resources of edge servers. Edge servers and the request routing components use the same *cryptographic hash* function. The inputs to this hash-function are the IP-address of the end-user and and a secret key. The request routing component assigns a request to an edge server using the hash function. The edge server also uses the hash-function to validate whether the request is actually assigned by the routing mechanism by checking that the hash matches the edge server itself. If the hash does not match with the edge server, it can be concluded that the request is sent directly without the intervention of a request routing component. Periodically changing the key can improve the robustness of this strategy. Moreover, in a Cloud environment using a shuffling mechanism [74], new replicas are instantiated for edge servers that are under attack, and end-users are *shuffled* among these replicas (i.e., the assignment of end-users to edge servers is changed) in order to isolate attackers.

2.2 Defence Mechanisms

2.2.1 Traditional Security Mechanisms

Scrubbing-centers are over-provisioned cloud data-centers that provide security mechanisms for high traffic loads. Using DNS or BGP mechanisms [6], 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 general, it is more desirable that custom security mechanisms can be deployed. Also, it is important that traffic is processed locally to avoid potential latency and throughput degradation.

2.2.2 Software Based Security Mechanisms

DrawBridge [78] 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) [111] provides interfaces from ISPs that enable victims to detect and mitigate attacks across multiple SDN-operated ISP networks. These studies suggest that some or all the network switches should be replaced with OpenFlow compatible switches. None of these proposals have seen any major deployment in real ISPs where significant changes have proven to be difficult or even impossible. In general it is more desirable to use standard mechanisms that can be deployed in practice without the need for major infrastructure changes. VFence [70] proposes a platform which performs SYN flood mitigation in a scalable manner by using dynamic allocation of virtual functions. To mitigate DDoS attacks, *Bohatei* [60] deploys a protection chain based on the attack types. The protection workflow is as follows: i) *flag* a suspicious flow, ii) *estimate* the attack volume, iii) *place* defense functions across multiple data-centers, and iv) *steer* traffic through the chain. To mitigate SYN Flood attacks, Alharbi et al. [44] present 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.

Chapter 3

Security Orchestration System for CDN Edge Servers

This thesis present a system that 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. In the following, we first introduce the system architecture, then describe in detail each component of the system.

3.1 Architecture

European Telecommunications Standards Institute (ETSI) developed monitoring and management use-cases in the context of NFV security [37]. Our system architecture adapts some of the definitions from the ETSI use cases. As depicted in Fig. 3.1, it consists of three components as follows.

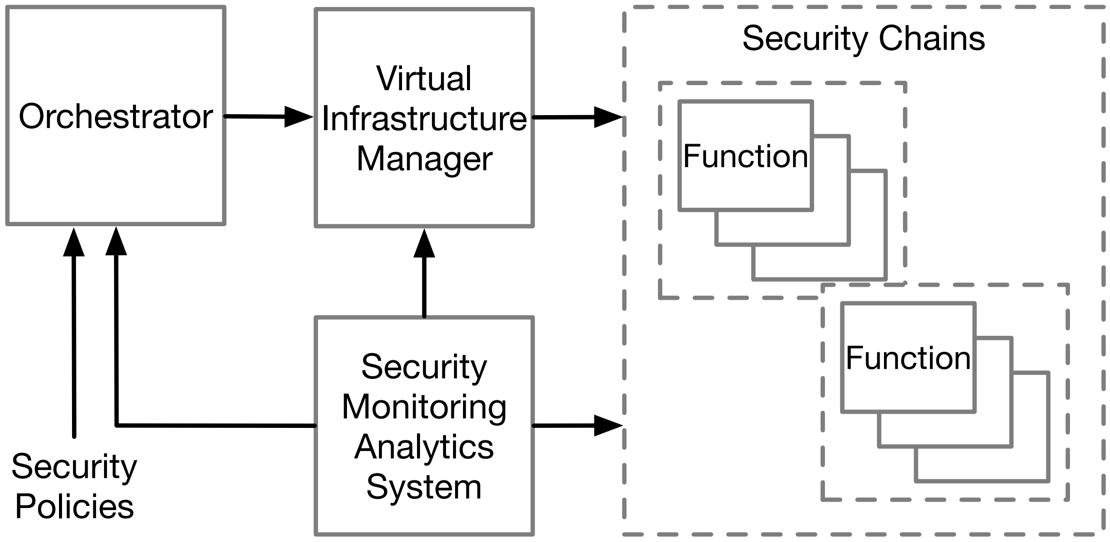

Orchestrator
Virtual Infrastructure Manager
Security Chains
Function
Function
Security Monitoring Analytics System
Security Policies

Figure 3.1: Architecture

3.1.1 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)

3.1.2 Virtual Infrastructure Manager

According to NFV-MANO [36], 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.

3.1.3 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 incoming traffic. 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.

3.2 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 [58]. 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 [68, 87], 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 $\mathcal{L}_{active}$ [46], 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.

3.2.1 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:

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

3.2.2 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 [83] (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 3.3.3.

3.2.3 ECA Rule Examples

Fig. 3.2 shows security policies that automatically deploy and remove a security chain. Rule 3.1 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 3.2 fires event $conf$ after $create\_chain(.)$ action if condition $function(y)$ is valid showing that y has been installed. Rule 3.3 runs command "conf.sh" to configure function y. Rule 3.4 instructs that after the chain is created, a timer named tx is set for 10 time units. Rule 3.5 deletes the chain upon a timeout event for a timer tx, if condition $chain(x)$ is true, meaning that chain x exists.

$$
\begin{aligned}
\mathit{high\_cpu}\ &\textbf{initiates}\ \mathit{create\_chain}(x{:}\_, 1, 2, \{y{:}\mathit{IDS}\}) \\
&\textbf{if}\ \text{not}\ \mathit{chain}(x) \qquad (3.1) \\
\mathit{conf}\ &\textbf{after}\ \mathit{create\_chain}(x) \\
&\textbf{if}\ \mathit{function}(y) \qquad (3.2) \\
\mathit{conf}\ &\textbf{initiates}\ \mathit{run}(y, \text{“conf.sh”}) \\
&\textbf{if}\ \text{true} \qquad (3.3) \\
\mathit{create\_chain}(x)\ &\textbf{causes}\ \mathit{timer}(tx, 10) \\
&\textbf{if}\ \text{true} \qquad (3.4) \\
\mathit{timeout}(tx)\ &\textbf{initiates}\ \mathit{delete\_chain}(x) \\
&\textbf{if}\ \mathit{chain}(x) \qquad (3.5)
\end{aligned}
$$

Figure 3.2: ECA Security Policy Examples

3.2.4 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* [84] 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 [84, 50, 46]. Finally, to execute actions introduced in Section 3.2.2, the orchestrator translates these declarations to actual VIM API calls discussed in Section 3.3.5.

3.3 Virtual Infrastructure Manager

Virtual Infrastructure Manager (VIM) is responsible for managing host resources and deploying and managing security function chains. VIM provides an API for creating/deleting a chain, inserting/deleting a function to/from a chain, and for querying information about deployed chains. This API is used by the orchestrator to manage security services, and by SMAS to query deployed functions.

3.3.1 Service Function Chaining Requirements

Service function chaining involves (i) defining and instantiating the service functions and (ii) steering traffic through a *service path*, the ordered sequence of service functions, and (iii) the ability to carry metadata. Thus, an appropriate solution has to manage service functions running on compute-resources and manage network-resources to route traffic and carry metadata through service paths. To implement requirements (ii) and (iii), we need a flexible protocol to steer traffic and carry metadata, and a network controller that supports this protocol. Implementing requirement (i) is simple. The hard part in realizing service function chaining is to implement requirement (ii) and (iii) and integrates this implementation with a compute-resource manager that realizes requirement (i). In summary, to realize service function chaining, we require a platform that provides the following features:

- orchestrating compute-resources to deploy and manage service functions,
- enabling a flexible protocol to carry metadata and steer traffic through service functions,
- providing a software switch fabric supporting the protocol mentioned above, and
- integrating the switch with a network controller that programs the switch fabrics mentioned above.

Existing Platforms

Service functions are usually packaged in *virtual machines* or *containers* (*LXC* and *Docker containers*), for which there are several stable platforms. Popular solutions to orchestrate compute-resources and deploy VMs and containers are *OpenStack Nova* [23], *OpenStack Zun* [32], *Docker* [86], and *Linux containers* [14].

Routing based on Ethernet and IP is not flexible for service function chaining. At each hop in a service path, an entity (e.g., a service function) has to be aware of the next service function address and modify IP and Ethernet addresses accordingly. Moreover, IP and Ethernet do not support metadata natively, and we need to hack them to carry metadata to inside their headers. Encapsulation protocols, such as VLAN [69], VXLAN [81], MPLS

Platform	SF Management	NSH Support	SP using NSH	Issues
OPNFV	VM and Container	Yes	Yes	Complex deployment and performance
OpenStack	VM and container	Yes	Yes	The lack of documentation, deprecated branches, unstableness
OpenStack Nova	VM	N/A	N/A	-
Kubernetes	Container	No	No	Not realizing service-paths
ODL SFC	No	Yes	Yes	No support for VM or container

Table 3.1: Available Solutions. SF and SP stand for Service Function and Service Path, respectively

[99], NVGRE [62], and STT [57] provide flexible routing using *tags*. In terms of carrying metadata, VLAN supports 12-bit; VXLAN, MPLS, and NVGRE provide 24-bit; and STT supports 64-bit length metadata fields. Less than 64-bit size is not sufficient to carry metadata. To overcome the limitation of metadata size, Geneve [66] provides variable-length metadata; however, switch fabrics strip this header before forwarding traffic to a service function, meaning that the service function does not receive carried metadata.

Network Service Header (NSH) [96] is the protocol that overcomes all limitations mentioned, and we have selected this protocol as our routing protocol. NSH is imposed to realize routing through service paths independent of physical addresses and is able to carry metadata. There are two versions of NSH. *NSH metadata type 1* supports 128-bit metadata, and *NSH metadata type 2* provides variable-length metadata. Although the second version is not yet well-supported, there are significant attempts in the industry to support variable length metadata, and we expect that this version will be widely supported in the near future. Currently, OVS NSH patch [25] provides the support of matching NSH rules and taking NSH actions. *OpenDaylight* (ODL) [21] and *Open Network Operating System* (ONOS) [19] are NSH-aware network controllers.

Kubernetes [27], *ODL service function chaining* [18], and OpenStack [22] are platforms that meet the above requirements to some extent as summarized in Table 3.1. Kubernetes automates the deployment, scaling, and management of Docker containers across a network; however, this system does not support NSH. OPNV is the integration of OpenStack with several other services. ODL service function chaining integrates OVS, OVS NSH patch, and SFC agents to realize service function chaining. However, SFC agents in this initiative are simple service functions developed for demo purposes. For these reasons, we first focused on OpenStack to realize service function chaining. However, we encountered several issues as discussed next, and we decided to implement our own service function chaining system.

OpenStack Issues: OpenStack includes several services orchestrated for creating public and private clouds. Devstack is a set of scripts used to quickly configure and deploy an OpenStack development environment. Service function chaining is not natively supported by OpenStack. We need to manually configure and orchestrate several services to realize service function chaining. These services are Docker, Nova Docker, and Nova service to deploy service functions, and NSH supported OVS, an NSH-aware Openflow controller (ODL or ONOS), and *Neutron* service [24] to steer traffic through service chains. Nova Docker was deprecated a few weeks after the beginning of the project. The replacement service is Zun [32]. There is no official documentation or tutorial to explain the integration of services mentioned with the basic services that a bare OpenStack deployment requires. There are a few blog posts [45, 49] on how to integrate a subset of these services. Unfortunately, all these blog posts use *Mitaka*, a deprecated branch of OpenStack. Therefore, we resorted to other OpenStack branches called Newton [16] and Ocata [17]. After numerous unsuccessful attempts to deploy our desired configuration with these branches, we decided to omit some of the services, including Zun. However, even with a subset of the services enabled, these branches are unstable due to failure of one of the services (either a bare service or the services for service function chaining).

Facing the limitations and issues of existing solutions, we decided to develop our own service function chaining platform using general purpose mechanisms and tools, as part of VIM.

3.3.2 VIM Building Blocks

Docker: Containers have a low resource overhead and are fast to create and destroy. VIM utilizes Docker [86] to manage container-based functions.

Network Service Header (NSH): NSH is a modern service plane protocol for dynamic service function chaining [97]. 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. Using NSH, service functions can exchange information [67]. NSH is a widely accepted industry standard.

Open Virtual Switch: VIM implements the networking aspect of service function chaining using Open Virtual Switch (OVS) [20]. 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.

3.3.3 Specifications

The following are used in calling the VIM's API.

Chain Specification

VIM uses the specification depicted in Fig. 3.3 where `chain_name` specifies *ch* to be the unique name of this chain. As mentioned in Section 3.2.2, 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 3.5, 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.

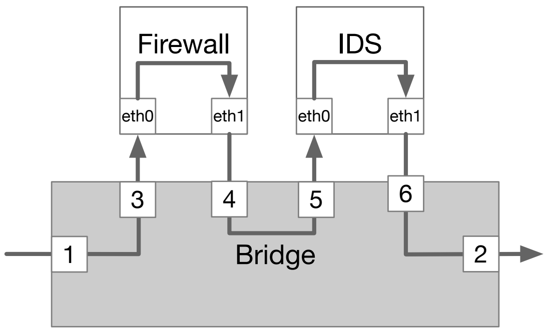
3.3.4 Service Function Chaining

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. 3.4a illustrates the deployment of the chain defined in Fig. 3.3.

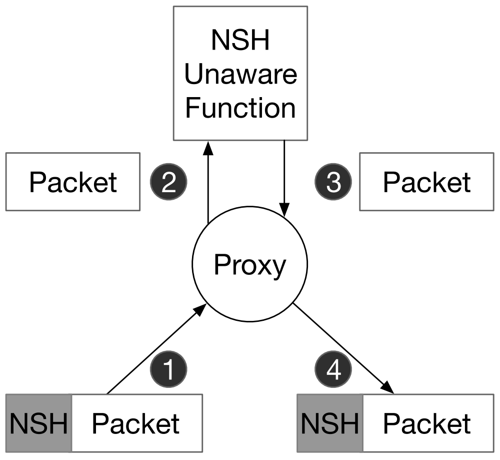
Three sets of rules are inserted to steer traffic through the chain. i) *Classification rules* filter incoming packets from `ingress` based on `classification_rules` and attach NSH header to these packets. ii) *Forwarding rules* are NSH-based match/action rules that forward packets between functions. In packet forwarding based on NSH, functions

```
{
  "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
    }
  ]
}
```

Figure 3.3: The Specification of a Chain and its Functions



(a) A Deployed Chain



(b) Function-Proxy

Figure 3.4: Service Function Chaining

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. 3.4b. iii) *Proxy*

```python
def create_chain(chain_sp)
def delete_chain(chain_name)
def insert(chain_name, func_sp, position)
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)
```

Figure 3.5: VIM API

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.

3.3.5 Northbound API

VIM provides the API shown in Fig. 3.5. Arguments `chain_sp` and `func_sp` are respectively the specifications of a chain and a function and must follow the specifications presented in Section 3.3.3. The first five methods correspond to actions defined in Section 3.2.2. The others are query methods about chains, functions, and traffic used by SMAS and the orchestrator.

3.4 Security Monitoring Analytics System

The Security Monitoring Analytics System (SMAS) is responsible for monitoring the resources of the edge-server and the incoming traffic to the edge-server, collecting and analyzing important metrics, and generating security alerts towards the orchestrator. The first part of this section presents how SMAS monitores the usage of resources, analyzes them, and fires alerts. Next, we explain monitoring and analyzing incoming traffic to the edge-server. SMAS uses two proposed machine learning methods to detect known and new attacks.

Table 3.2: 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		

3.4.1 Resource Monitoring Analytics

The focus here is on handling misuse attacks through monitoring and analyzing the resources of the virtual edge-server. SMAS periodically monitors and collects statistics on *network-bandwidth*, *CPU*, *memory*, and *storage* resources. Our implementation relies on Linux standard tools, such as the `/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 CPU 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 3.2. 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 contrast 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.

3.4.2 Traffic Monitoring Analytics

This section focuses on monitoring and analyzing the incoming traffic of the virtual edge-server to generate alerts. Traditionally, *attack signatures* are used to detect intrusions. Human experts manually craft these signatures based on their knowledge of the intrusions, which requires substantial delay to recognize new attacks and identify their signatures. This limitation motivates the application of machine learning and data mining methods that automatically devise models replacing manually crafted attack signatures. In the literature, misuse and anomaly detection have been extensively used for intrusion detection.

Misuse detection is commonly done using supervised machine learning, which requires training over a labeled dataset's records [98]. Although accurate in detecting known attack types, supervised learning methods are weak in detecting new attacks not seen in their training dataset. Unsupervised anomaly detection methods model the normal behavior of a system and detect deviations from it; they are capable of detecting new attacks. A number of anomaly detection algorithms have been proposed before. However, they often suffer from low accuracy rate and higher false alarms compared to supervised methods [76].

To tackle the aforementioned limitation of supervised machine learning, we propose a Hybrid Classification Clustering (HCC) method that not only achieves high accuracy in the detection of known attacks, but it can recognize new attacks. To address the limitations of traditional anomaly detection methods, we propose Autoencoder-based Network Anomaly Detection method (ANAD). This method improves the accuracy of HCC on detecting new attacks.

SMAS monitors the incoming traffic to the edge-server, analyzes it using our proposed ML methods, and fires attack alerts, such as *app_ddos*, *port_scan*, and *anomaly*. Based on these alerts, appropriate security policies are triggered to mitigate attacks. To do So, SMAS periodically monitors the incoming traffic of the edge-server and extracts features of traffic flows identified using the five tuple (Source and Destination IP addresses, Source and Destination ports, and protocol). Then, SMAS runs our proposed ML algorithms to analyze these features for attack and anomaly detection. In the following, we describe our proposed ML methods, namely HCC and ANAD.

Hybrid Classification Clustering (HCC)

Table 3.3 compares HCC with classification and clustering methods. As shown, HCC provides the benefits of both classification and clustering. Similar to a classification method, HCC detects the type of a known attack. Similar to a clustering method, HCC identifies whether a flow is a new attack (though the type of a new attack is not determined). HCC trains a classifier using labeled training dataset to detect existing attacks in the dataset, which we refer to as *known* attacks. HCC uses a clustering method to discover clusters that contain new attacks, i.e., previously unseen attacks in the classifier's training dataset.

Let $X = (x_1, \ldots, x_n)$ be a set of n data points where $x_i \in \mathcal{X}$ for all $i \in \{1, \ldots, n\}$. The goal of supervised learning is to find a mapping from data points to *labels* from a *training set* of pairs (x_i, y_i), where $y_i \in \mathcal{Y}$ is the label of a data point x_i. An unsupervised learning method aims to infer interesting structures in data points X. For instance, a clustering

Table 3.3: A Comparison Between Machine Learning Methods

	Classification	Clustering	HCC
Known Attack Type	✓	✗	✓
New Attack	✗	✓	✓
New Attack Type	✗	✗	✗
Labeled Training Dataset	✓	✗	✓

algorithm categorizes X into *clusters*, each of which contains data points that are believed to have *similar* structures.

Overview: As shown in Algorithm 1, HCC receives as incoming traffic features f and the number of clusters C (line 1). Note that C is always greater than the number of classes. For each class there is a corresponding cluster. Additional clusters correspond to new attacks. In addition to known class labels, the algorithm predicts the `new_attack` label. First, the algorithm runs a pre-trained classifier and a clustering method (lines 2 and 3). Next, HCC recognizes the clusters that most probably contain the data points of a class and returns the rest of the clusters as new attacks (line 4). Combining all results, HCC predicts new labels for the data points (line 5). Still to clarify is how to find new attack clusters and how to find a new prediction.

Algorithm 1 Hybrid Classification Clustering

```
1: procedure HCC($f, C$)
2:     $y_1 \leftarrow$ classification($f$)
3:     $y_2 \leftarrow$ clustering($f, C$)
4:     $N \leftarrow$ NOVELS($y_1, y_2$)
5:     $y_3 \leftarrow$ PREDICT($y_1, y_2, N$)
6:     return $y_3$
7: end procedure
```

Finding New Attack Clusters: Algorithm 2 receives the predictions and finds clusters that contain new attacks. To do so, it compares the classification and clustering predictions and relates clusters to classes. A cluster is considered *known*, if its intersection with at least one class is larger than other clusters'. The algorithm finds known clusters and considers the remaining clusters as new attacks.

The algorithm receives the results of the classifier and clustering methods in the form of two lists with indices showing the data points and values showing the predictions. Next, it creates two dictionaries d_1 and d_2 (line 2). The former is a mapping of the classes to their corresponding data points, and the latter is a dictionary from the clusters to their corresponding data points. The algorithm initializes a list K that is iteratively extended with known clusters (lines 3-11). For each class l_1, the algorithm finds the size of the intersection of the members of l_1 with all the clusters' members (lines 6-9). The clusters with the biggest intersection with each class are identified and added to K (line 10). Each class can be mapped to one or multiple clusters. The algorithm returns the clusters that are not in K (line 12).

Algorithm 2 Finding New Attack Clusters

```
1: procedure NOVELS($y_1, y_2$)
2:     $d_1, d_2 \leftarrow \text{map}(y_1), \text{map}(y_2)$
3:     $K \leftarrow \emptyset$
4:     for $l_1$ in $d_1$ do
5:         $c \leftarrow \text{array}(|\text{keys}(d_2)|)$
6:         for $l_2$ in $d_2$ do
7:             $s \leftarrow |\text{values}(d_1, l_1) \cap \text{values}(d_2, l_2)|$
8:             set value of $c$ at $l_2$ to $s$
9:         end for
10:        $K \leftarrow K \cup \text{argmaxes}(c)$
11:    end for
12:    return $\text{keys}(d_2) - K$
13: end procedure
```

Finding A New Prediction: Dictionary d_2 maps clusters to their corresponding data points (line 2). List L is initialized and iteratively updated by the data points in the new attack clusters (line 4-6). List y_3 that stores the final results is first initialized by the classification results, then updated by changing the values of data points in L to `new_attack` (lines 7-9).

Algorithm 3 Finding a New Prediction

```
1:  procedure PREDICT($y_1, y_2, N$)
2:      $d_2 \leftarrow$ map($y_2$)
3:      $L \leftarrow \emptyset$
4:      for c in N do
5:          $L \leftarrow L \cup$ values($d_2, c$)
6:      end for
7:      $y_3 \leftarrow y_1$
8:      set values of $y_3$ at $L$ to new_attack
9:      return $y_3$
10: end procedure
```

3.4.3 Autoencoder-based Network Anomaly Detection (ANAD):

Anomaly detection approaches are able to discover new intrusions by modeling the normal behavior of the system and detecting any deviation from it [52, 61, 100]. Several factors, such as the difficulty of having a boundary around the normal behavior, intelligent adversaries who adapt themselves to new detection methods, and insufficient training/testing data make anomaly detection a complex task. Anomaly detection methods are categorized as *supervised*, *semi-supervised*, and *unsupervised*. The first category is classifies traffic into two classes, normal and anomalous. As discussed before, the supervised methods are not adequate in detecting new attacks. Semi-supervised approaches are trained based on normal traffic only. Providing real traffic traces which only contain normal data is a complex task. Unsupervised learning approaches detect the anomalies without any labeled training data with the assumption that normal traffic instances are much more frequent than anomaly instances [52, 47]. However, most unsupervised anomaly detection methods suffer from low accuracy.

Deep Learning based anomaly detection recently received significant attention in different fields, such as fraud detection, medical diagnosis, and network intrusion detection [109, 90, 73]. In [101], a generative adversarial network simultaneously trains a generator to produce anomalous data which are close to healthy data and trains a discriminator to detect the anomalies. Unsupervised deep learning models are also employed for learning a representation of the data in a lower dimension than the input's. Later, this representation is used for supervised anomaly detection [109, 59, 73]. In [59], a one-class SVM uses features extracted by a deep belief network for classification. A stacked non-symmetric deep autoencoder has been used in [73] to extract features for random forest classifier.

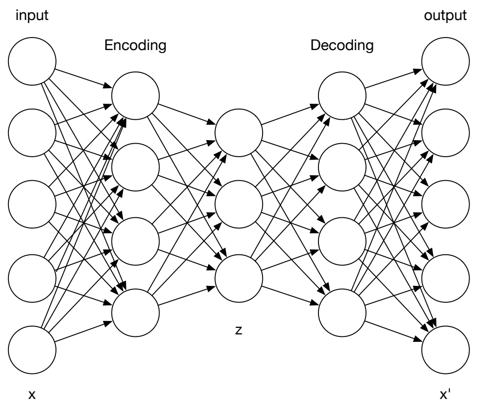


Figure 3.6: AutoEncoder

Unlike other methods, we use autoencoders to design an unsupervised network anomaly detection method. Our unsupervised method is not only capable of detecting new attacks, but achieves higher accuracy in comparison with commonly used unsupervised anomaly detection methods.

Autoencoder: Autoencoders are deep neural networks trained to learn efficient data coding. A sample autoencoder is represented in Fig. 3.6. An autoencoder contains an input layer, one or more hidden layers, and an output layer. The input and output layers of an autoencoder have the same dimension, while the number of neurons may vary in hidden layers. The first part of this model, the *encoder*, transforms the high dimensional input x into a lower dimensional representation z. Equation (3.6) presents the encoder function of a single encoder layer, where σ is the activation function, W is the weight matrix, and b is the bias vector of the hidden layer. The autoencoder decodes data in what is called the *latent space* and learns to extract the most important features in the decoding phase. The second part, *decoder*, uses this lower representation to rebuild the input in x'. Equation (3.7) presents the decoder function; W' is the weight matrix, and b' is the bias vector of the hidden layer. The loss function of an autoencoder is based on the *reconstruction error*, the difference between the input x and prediction x'.

$$z = \sigma(Wx + b) \tag{3.6}$$

$$x' = \sigma(W'z + b') \tag{3.7}$$

ANAD: Our method trains a fully-connected autoencoder with unlabeled training dataset. Note that the training dataset consists of mostly normal data points. The reconstruction errors of anomalies are higher than those of the normal data points, because the autoencoder tries to learn the encoding and decoding of the normal data points which constitute most of the data. To compute the reconstruction error of a data point, we use its *Residual Sum of Square (RSS)* as presented in Equation (3.8), where x is the input, x' is the prediction, and n is the number of input features.

$$\sum_{i=1}^{n} (x_i - x'_i)^2 \tag{3.8}$$

ANAD analyzes incoming flows as shown in Algorithm 4. This method receives the input data points x and contamination parameter c stating the proportion of anomalies in the given data points (line 1). ANAD runs the autoencoder and stores the predictions in x' (line 2). For each data point, this method computes and stores its reconstruction error using RSS (lines 3-6). The algorithm sorts the data points descendingly based on their reconstruction errors and returns the top c percent as anomalies (lines 7-8). For a constant number of alerts, a network operator can set the contamination parameter c based on the flow incoming rate.

Algorithm 4 Autoencoder-based Network Anomaly Detection

```
1: procedure ANAD($x, c$)
2:     $x' \leftarrow$ autoencoder($x$)
3:     $e \leftarrow \emptyset$
4:     for $x_1, x_2$ in $x, x'$ do
5:         $e \leftarrow e \cup \mathrm{RSS}(x_1, x_2)$
6:     end for
7:     sort $x$ based on $e$ descendingly
8:     return top $c\%$ of $x$
9: end procedure
```

3.5 Use-case Scenarios

3.5.1 Rate Limiting Use-case

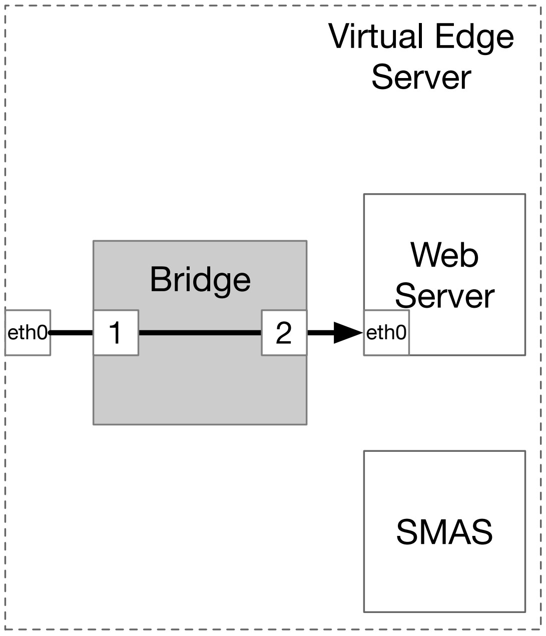
Rate limiting is a common practice [38, 43, 40] that CDNs use against threats ranging from network layer attacks, e.g. DDoS, to application layer attacks, e.g. brute-force login attempts. Various rate-limiting mechanisms exist, such as limiting traffic-rate per user, geography, or server. In this use-case, traffic is rate-limited per-user. Fig. 3.7 illustrates this scenario, and Fig. 3.11 lists the applicable security policies.

Monitoring Stage. Fig. 3.7a 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.

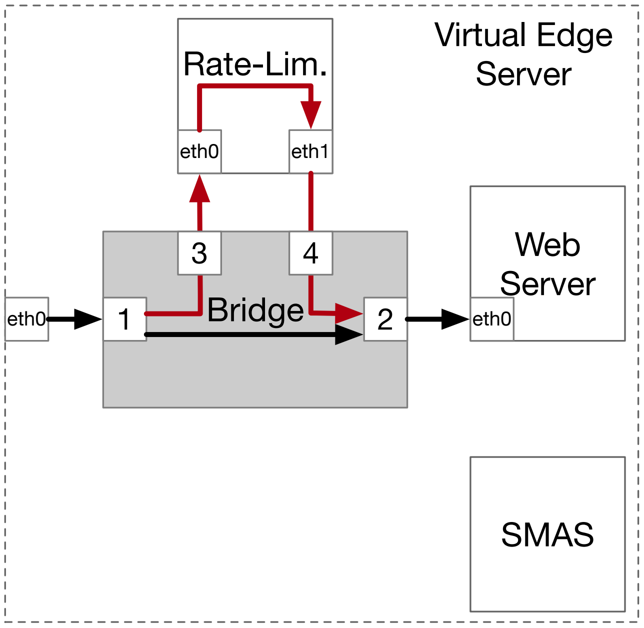
Rate Limiting Stage. Based on Rules 3.23-3.25, upon receiving the alert *high_rate*, if no rate-limiting service exists, the system deploys chain r containing a *Rate-limit* function to limit the traffic-rate per IP (representing per end-user traffic). A white-list of IP addresses are exempted from rate-limiting. Fig. 3.7b shows this chain. To enforce Rule 3.12, 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 3.13 and 3.14 are matched. First, executing Rule 3.13, chain n with no function is deployed. As chain n connects ports 1 and 2, traffic is forwarded to the Web-server. Then, Rule 3.14 is matched, and chain r is removed.

3.5.2 Mitigating HTTPS DDoS Use-case

HTTPS DDoS attacks exploit HTTP and HTTPS and target Web applications running on a server [56, 108]. 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 [39, 42]. Inspection at the application layer is a heavy process that can affect the application response time [34]. 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. 3.9 depicts this use-case scenario, and Fig. 3.10 lists ECA policies enforced.



(a) Monitoring Stage



(b) Rate Limiting Stage

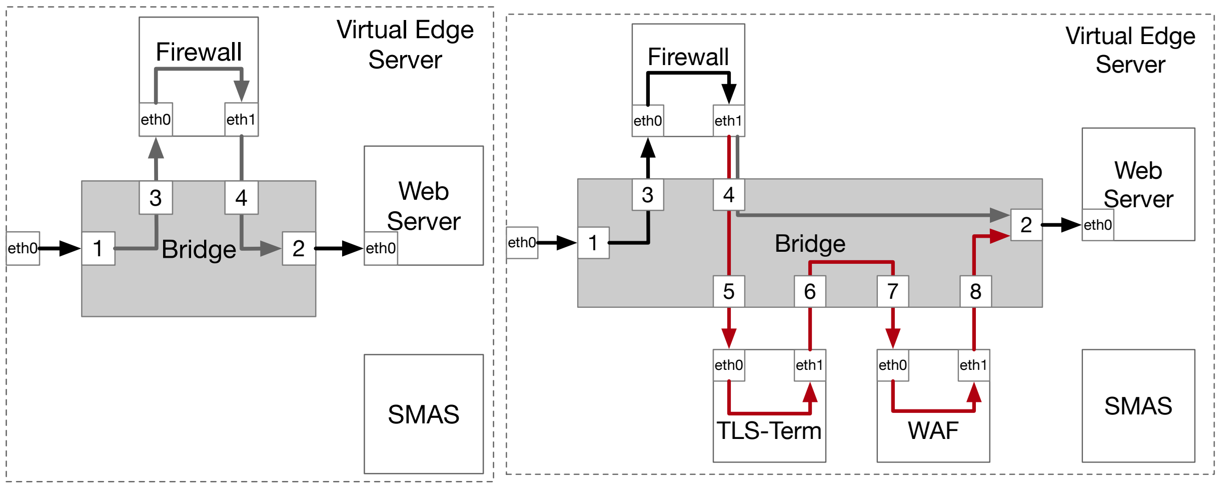
Figure 3.7: Rate Limiting Scenario

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 3.15, the system instantiates chain u composed of a $Firewall$ named f, as shown in Fig. 3.9a. 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 3.16 and 3.17); 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 3.18). When this timer expires, another chain l comprising a TLS-$Term$ (a TLS termination) and a WAF is instantiated to perform mitigation at the application layer (Rule 3.19). Fig. 3.9b 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-$Term$ 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

$$\begin{aligned}
\textit{high\_rate}\ &\textbf{initiates}\ \textit{create\_chain}(r{:} \\
&\quad <\text{“not src net } 129.97.124.0/24\text{”}, 1, 2>, \\
&\quad \{f{:}\textit{Rate-limit}\}) \\
&\textbf{if}\ \text{not}\ \textit{chain}(r) \qquad (3.9) \\
\textit{lim}\ &\textbf{after}\ \textit{create\_chain}(r) \\
&\textbf{if}\ \text{true} \qquad (3.10) \\
\textit{lim}\ &\textbf{initiates}\ \textit{run}(f, \text{“}\textit{rate\_limit.sh}\text{”}) \\
&\textbf{if}\ \text{true} \qquad (3.11) \\
\textit{create\_chain}(r)\ &\textbf{causes}\ \textit{timer}(tr, d) \\
&\textbf{if}\ \text{true} \qquad (3.12) \\
\textit{timeout}(tr)\ &\textbf{initiates}\ \textit{create\_chain}(n{:}<\_, 1, 2>, \{\}) \\
&\textbf{if}\ \text{true} \qquad (3.13) \\
\textit{timeout}(tr)\ &\textbf{initiates}\ \textit{delete\_chain}(r) \\
&\textbf{if}\ \textit{chain}(r) \qquad (3.14)
\end{aligned}$$

Figure 3.8: Rate Limiting Policies



(a) L3 Mitigation Stage

(b) L4 Mitigation Stage

Figure 3.9: Mitigating HTTPS DDoS Scenario

$$\begin{aligned}
\mathit{cpu\_high}\ &\textbf{initiates}\ \mathit{create\_chain}(u{:}{<}\text{“ip”}, 1, 2{>},\\
&\{f{:}\mathit{Firewall}\})\\
&\textbf{if}\ \text{not}\ \mathit{chain}(u) && (3.15)\\
\mathit{block}\ &\textbf{after}\ \mathit{create\_chain}(u)\\
&\textbf{if}\ \text{true} && (3.16)\\
\mathit{block}\ &\textbf{initiates}\ \mathit{run}(f, \text{“}\mathit{block.sh}\text{”})\\
&\textbf{if}\ \text{true} && (3.17)\\
\mathit{create\_chain}(u)\ &\textbf{causes}\ \mathit{timer}(\mathit{td}, d)\\
&\textbf{if}\ \text{true} && (3.18)\\
\mathit{timeout}(\mathit{td})\ &\textbf{initiates}\ \mathit{create\_chain}(l{:}\\
&\text{“not src net 99.231.0.0/16”}, f, 2,\\
&\{t : \mathit{TLS\text{-}Term}, w : \mathit{WAF}\})\\
&\textbf{if}\ \text{not}\ \mathit{chain}(l)\ \text{and}\ \mathit{chain}(u) && (3.19)\\
\mathit{cpu\_low}\ &\textbf{initiates}\ \mathit{create\_chain}(n{:}{<}\_, 1, 2{>}, \{\})\\
&\textbf{if}\ \text{true} && (3.20)\\
\mathit{cpu\_low}\ &\textbf{initiates}\ \mathit{delete\_chain}(u)\\
&\textbf{if}\ \mathit{chain}(u) && (3.21)\\
\mathit{cpu\_low}\ &\textbf{initiates}\ \mathit{delete\_chain}(l)\\
&\textbf{if}\ \mathit{chain}(l) && (3.22)
\end{aligned}$$

Figure 3.10: HTTPS DDoS Mitigation Policies

$$\begin{aligned}
&\textit{anomaly}\ \textbf{initiates}\ \textit{create\_chain}(r{:}\\
&\quad <\text{“”}, 1, 2>, \{r{:}\textit{Rate-limit}\})\\
&\quad \textbf{if}\ \text{not}\ \textit{chain}(r), \text{not date("Christmas")} && (3.23)\\
&\textit{rate\_limit}\ \textbf{after}\ \textit{create\_chain}(r)\\
&\quad \textbf{if}\ \text{true} && (3.24)\\
&\textit{rate\_limit}\ \textbf{initiates}\ \textit{run}(r,\\
&\quad \text{“rate\_limit.sh } \textit{anomaly}\text{.suspicious\_ips 1Gbps”})\\
&\quad \textbf{if}\ \text{true} && (3.25)
\end{aligned}$$

Figure 3.11: Rate Limiting Policy

3.20-3.22).

Rate-limit New Attacks: A flash-crowd happens when websites suddenly become popular due to a big event taking place. Flash-crowds can be easily mistaken for a DDoS attack, when a resource misuse alert is generated. It is important for network operators to over-provision resources for handling the requests of legitimate users during peak workload periods (e.g., during Black Friday). However, during regular workload periods, the protection system must deploy a mitigation service, such as rate-limiting when anomalies are detected. Several Rate-limiting mechanisms exist, e.g, limiting traffic-rate per user or server [110, 38, 40].

In the use-case of Fig. 3.11, SMAS using ANAD algorithm detects anomalies and generates corresponding alerts. First, receiving *anomaly* alert, the policy deploys a per user rate-limiting service. The security chain contains a *Rate-limit* function, if it does not already exist, and if the date is not in the Christmas week (Rule 3.23). Second, a limit of 1 Gbps is forced for the accumulated traffic of suspicious IPs detected by ANAD (Rule 3.24 and Rule 3.25).

Chapter 4

Evaluation

4.1 Detection Performance

4.1.1 Experimental Platform

We use a cluster of machines (256 GB RAM, 32-cores 2.00 GHz Xeon CPUs), equipped with NVIDIA Tesla K10 GPU (320 GBps memory bandwidth, 3072 CUDA cores 745 MHz). The server runs Ubuntu 16.04 with Linux kernel version 4.4.0.

4.1.2 Dataset

We leverage a labeled dataset *CIC-2017* [64] to train and test our models. This dataset consists of normal and multiple types of attack traffic. In addition to packet capture (pcap) files, network flows and their corresponding features have been extracted using CICFlowMeter [7]. To simulate an edge-server traffic, we use flows towards a victim server.

4.1.3 Training and Testing

We use 70% and 30% of the labeled data points as the training dataset and testing dataset, respectively. These data points are labeled either normal or an attack (e.g., DDoS and port scan). There are 81 features for each flow from which we extract 76 features. IP addresses and port numbers are 32-bit and 16-bit numerical values, respectively. These

Table 4.1: Experimental Dataset of the Attack Detection Performance

Attack	# of Flows	Label
Benign	182491	0
DoS Hulk	230124	1
PortScan	158804	2

numerical features are commonly used to train machine learning algorithms; however, flows from different classes might have close numerical port numbers, and the machine learning algorithm interprets the close numbers as similarity between data points. The same argument is valid for IP addresses; therefore, the source and destination IPs, the source and destination ports, and the flow identification are removed from our feature set.

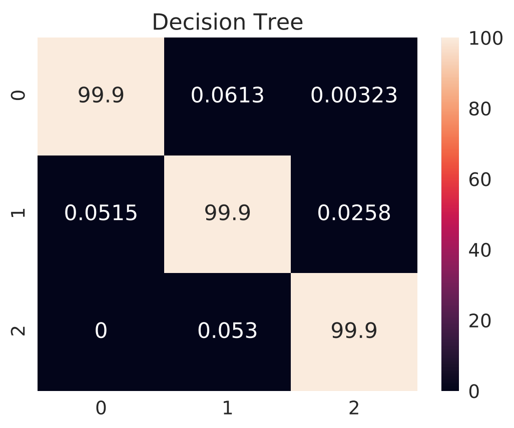
Throughout this section, the performance of machine learning methods is reported using the normalized confusion matrix, where an element ij represents what percentage of class i is classified under class j. In this way, the elements on the diagonal show the *recall* or *true positive rate* values. The sum of the non-diagonal elements of a row shows the *false negative rate* for the corresponding class.

4.1.4 Attack Detection Performance

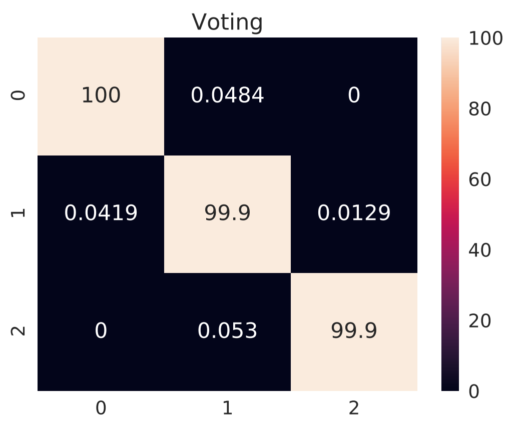
To evaluate the performance of HCC, we need to equip this algorithm with a classifier and a clustering method. To do so, we run and compare a number of classification methods and select a classifier that achieves the highest performance. HCC performs the same search for clustering methods. Then, to show the effectiveness of HCC in mitigating new attacks, we compare our hybrid method with the best selected classifier. For training and testing purposes, we use three classes of flows listed in Table 4.1.

Classifiers and Clustering Methods Performance

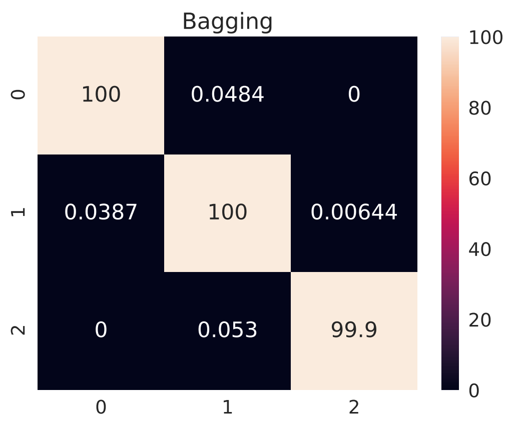
We trained a decision tree, a random forest, and a bagging classifier. We also trained an *ensemble voting*, which uses a voting mechanism between the above three classifiers. A bagging classifier uses random subsets of a dataset to train multiple decision trees. A random forest takes random features as well as random subsets to train multiple decision trees. Both of these ensemble methods [79] take the average of the trained decision trees' predictions. Fig. 4.1 shows the confusion matrices of the classifiers. Although these models have close recall values, the bagging classifier has a recall of 100% for both normal and DoS Hulk classes and 99% for the port scan class which makes this classifier the best among those tested.



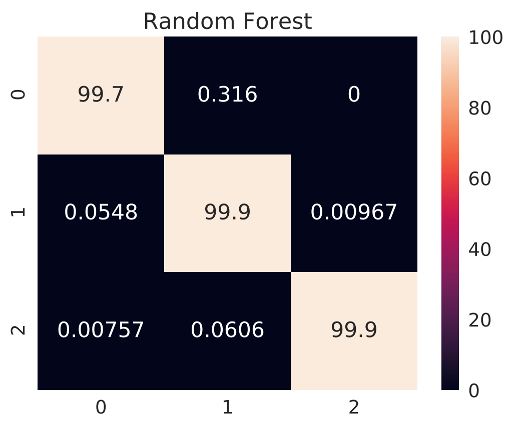
(a) Decision Tree



(b) Voting



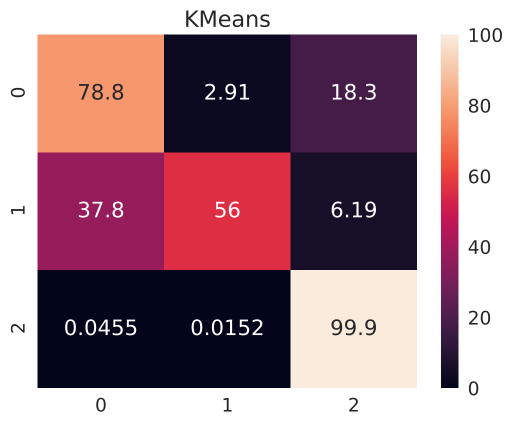
(c) Bagging



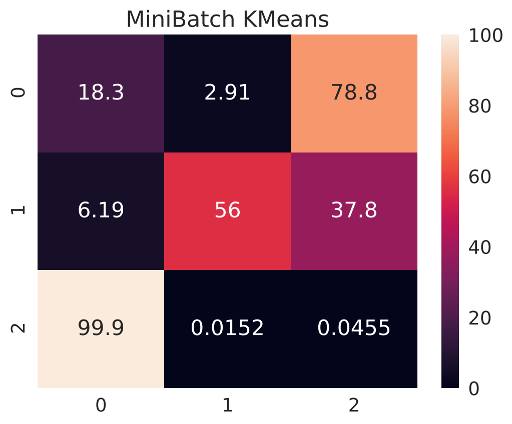
(d) Random Forest

Figure 4.1: Supervised Algorithm Confusion Matrices

We used *k-means* and *mini-batch k-means* to divide flows into 3 clusters. Both algorithms start by randomly initializing the cluster centroids and iteratively adjust the centroids. In each iteration, the k-means assigns all the data points to cluster centroids, whereas the mini-batch k-means does that on only a batch of data points. Then, the cluster centroids are updated based on new data points assigned to them. The performance of the



(a) k-means



(b) mini-batch k-means

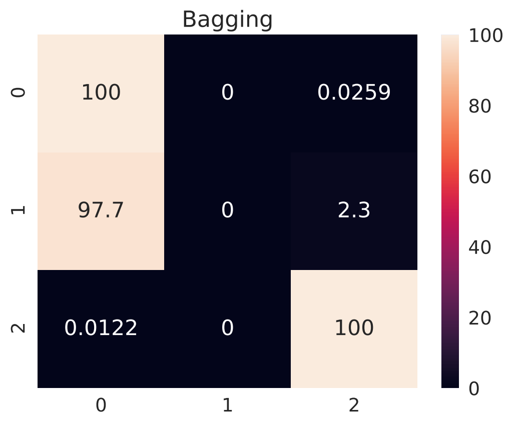
Figure 4.2: Clustering Algorithms Confusion Matrices

clustering methods is shown in Fig. 4.2. A cluster with the most data points of a class c is mapped to c and the recall of c is computed based on the data points in mapped cluster. For example, in Fig. 4.2b, the recall for normal class is 78.8% which is the percentage of normal data points in cluster 2. The different mappings for the two algorithms is due to the random initialization of the cluster centroids. Fig. 4.2 shows that these two algorithms have the same recall values suggesting both of them as good candidates.

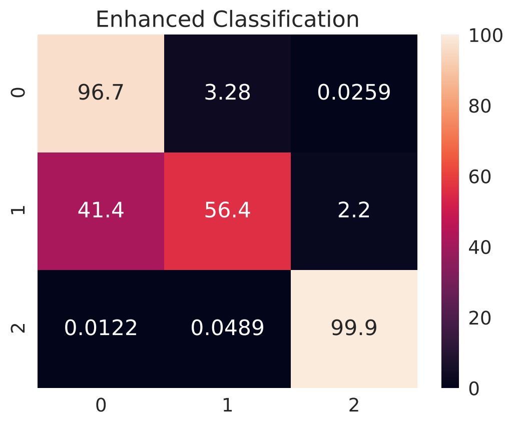
HCC Performance

We equip HCC with the best classification method, the bagging classifier, and one of the clustering candidates, k-means. HCC and its competitor, bagging, are trained over two classes (normal and port scan), then they are tested for all three classes shown in Table 4.1. The goal is to evaluate the performance of HCC in detecting the data points of the unseen class (DoS Hulk). From the new attack data points, the bagging classifier mis-classifies 97.7% and 2.3% under the normal and port scan classes, respectively. Mis-classification as the normal class can cause undesirable outcomes; no mitigation service is deployed, and DoS attack can exhaust the resources of the victim and take down its services. Mis-classification as a wrong attack results in the deployment of inappropriate mitigation services that not only does not mitigate the threat, but contributes to the attack and consumes more resources of the victim.

The performance of HCC is reported in Fig. 4.3b. HCC correctly detects 96.7% and



(a) Bagging



(b) Hybrid

Figure 4.3: Hybrid Method Performance in Detecting Known and New Attacks

Table 4.2: Experimental dataset of the anomaly detection experiments

Class	# of Flows	Label
Benign	438693	0
Anomaly	10293	1

99.9% of the normal and port scan data points, respectively. For the new attack data points, HCC is able to detect 56.4%, while mis-classifies 41.4% and 2.2% as normal and port scan classes. The false alarm rate is 3.3% due to the inaccuracy of the unsupervised clustering method, k-means. We believe that a more advanced clustering algorithm can improve HCC's performance.

4.1.5 Anomaly Detection Performance

We evaluate ANAD by comparing its accuracy with that of *LOF* [48] and *isolation forest* [80], two commonly used anomaly detection methods. These anomaly detection methods, including ANAD, receive several input parameters which affect their detection performance. We tune the values of these parameters to optimize their recalls for anomalies. To do so, we employ exclusive grid search that examines all the combinations of given values for all parameters. Finally, we use the data points of the two classes provided in Table 4.2.

Parameters	Isolation Forest			LOF		
	examined	default	chosen	examined	default	chosen
n_jobs	1, 1	1	-1	1, 1	1	-1
contamination	.1, .05, .022	.1	.1	.1, .05, .022	0.1	0.1
n_estimators	50, 100, 300, 500	100	300	-	-	-
max_samples	auto[a], .001, .01	auto	0.01	-	-	-
max_features	.02, .05, 1.0	1.0	.02	-	-	-
n_neighbors	-	-	-	20, 25, 30	20	30

[a]min(256, number of samples)

Table 4.3: Anomaly Detection Methods Parameters

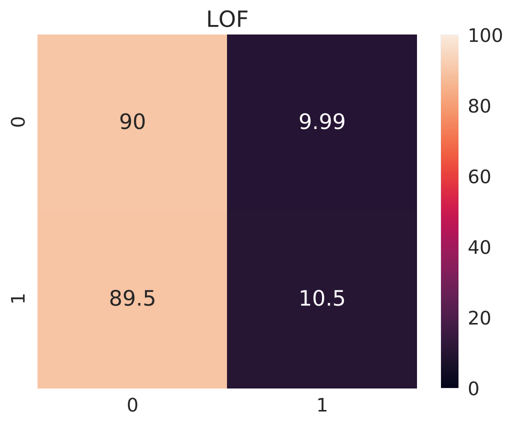
Anomaly Detection Methods Performance

For each data point, LOF computes the local density which is the metric depicting how isolated this data point is compared with its k neighbors. Isolation forest is an ensemble learning method that combines the results of multiple decision trees each of which produces an anomaly score. The isolation forest takes the average over these anomaly scores. Both methods are given *contamination*, an input parameter that identifies the proportion of anomalies. Optimized using the grid search, the default, examined, and chosen values of the input parameters are shown in Table 4.3. *n_jobs* denotes the number of jobs to run in parallel. If given -1, this parameter is set to the number of cores. *contamination* represents the percentage of outliers and *n_ estimators* specifies the number of decision trees trained by the isolation forest. Isolation forest receives the number of samples, and the number of features to draw for training each decision tree, as *max_ samples* and *max_features*, respectively. LOF receives *n_ neighbors* to set k, the numbers of neighbours considered of each datapoint. This value can be absolute or a percentage of the dataset.

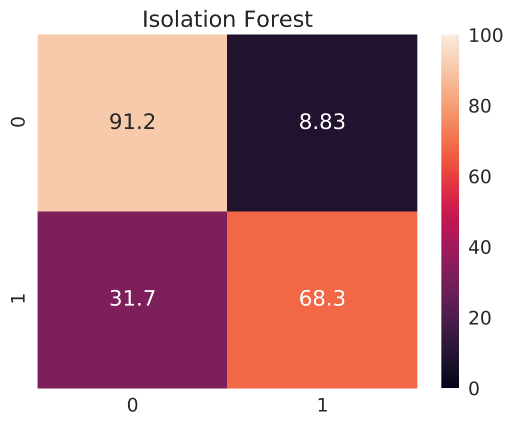
The performance of these two methods is shown in Fig. 4.4. LOF's recall for the anomalies is 10.5%, and that of isolation forest is 68.3%. The recall of the normal class is almost 90% for both methods. The remaining 10% causes low precision results for anomalies, i.e., LOF and isolation forest produce 98% and 88% false alarm rates, respectively. These poor results motivate the use of deep learning based approaches for anomaly detection.

ANAD Performance

ANAD employs an autoencoder with input and output layers of size 76 and two hidden layers. The hyper parameters, examined values, and chosen ones are reported in Ta-



(a) LOF



(b) Isolation Forest

Figure 4.4: Traditional Anomaly Detection Algorithms Confusion Matrices

ble 4.4.*n_ neurons* denotes the number of neurons in the hidden layers. *activation_function* specifies the function of the neurons. *drop_ out* regulates a rate of randomly selected neurons to be set to zero during each training iteration. This parameter helps the model not to overfit [102]. *training_ epoch* denotes the number of times the model should be trained on the whole training dataset. The number of data points that the model processes before each update of the weights and biases is represented by *batch_ size*. *kernel_ initializer* indicates the distribution from which the weights are initialized.

Fig. 4.5 reports ANAD performance. ANAD outperforms LOF significantly. In comparison with isolation forest, the recall values are improved by 0.4% and 8.4%. These results confirm that the autoencoder-based anomaly detection achieves higher performance in the detection recalls of normal and anomaly data points compared to the two commonly used anomaly detection methods.

4.2 Mitigation Performance

4.2.1 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.

Table 4.4: ANAD's Hyper Parameters

Hyper-params	examined	chosen
contamination	.1, .05, .022	.1
n_neurons	5, 35	5
activation_function	relu, tanh, linear	linear
epochs	100, 150	150
batch_size	100, 1000	100
dropout_rate	0.0, 0.2	0.0
weight_constraint	1, 5	5
kernel_initialization	uniform, normal	normal

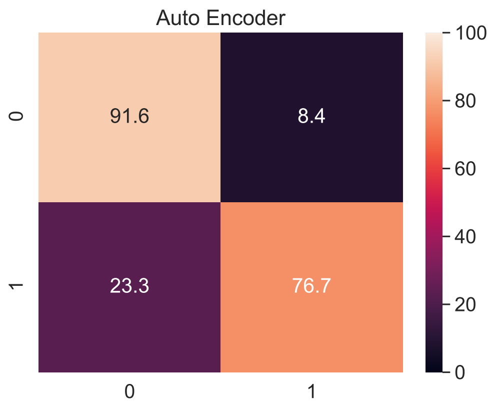


Figure 4.5: ANAD's Performance

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.

4.2.2 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.

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

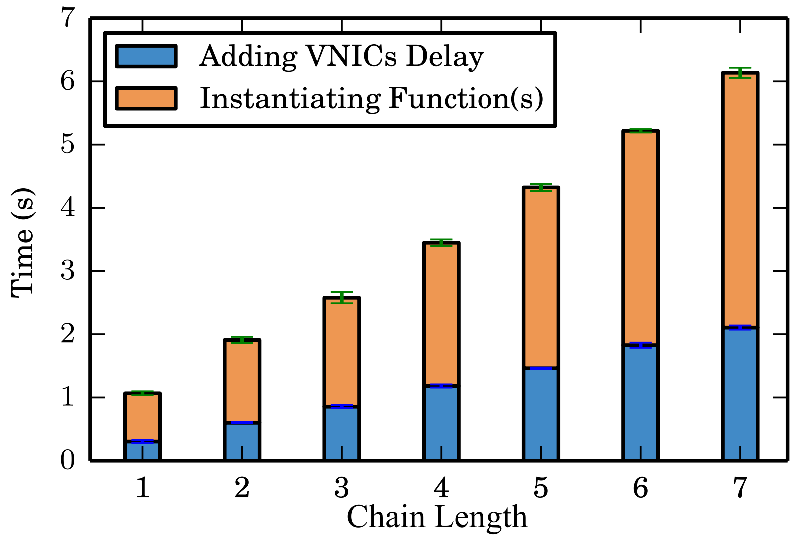


Figure 4.6: Chain Deployment Time

to OVS are the two most time-consuming procedures. As shown in Fig. 4.6, 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 [60]) 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.

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. 4.7, 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.

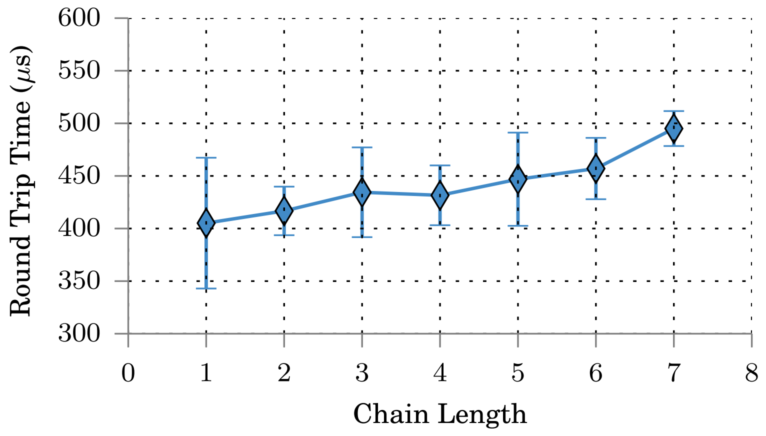


Figure 4.7: Traffic Round Trip Time

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. 4.8, the chain-length has a 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) [33] 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.

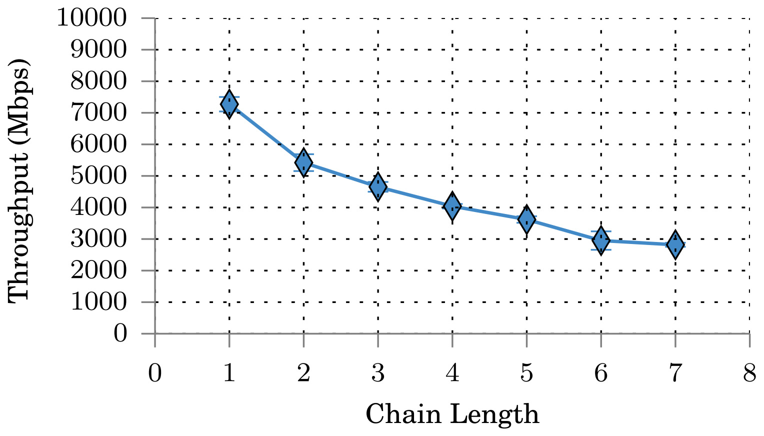


Figure 4.8: Throughput vs. Chain Length

4.2.3 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 [41]). 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. 4.9, 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× 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 of `Dynamic` should be served with the same latency as in the case of `Baseline`. However,

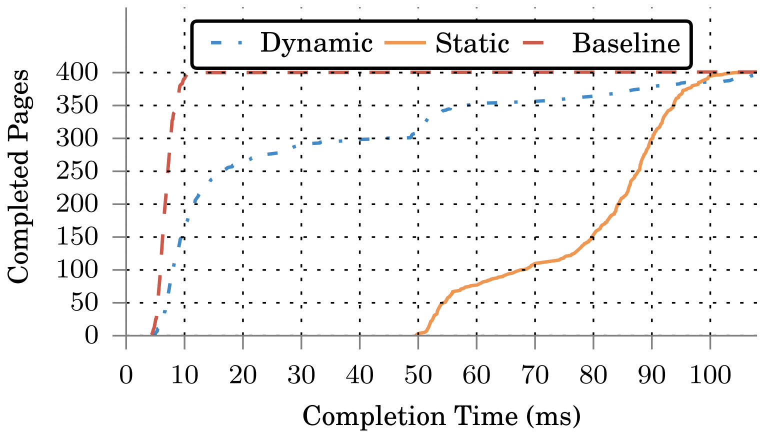


Figure 4.9: Completion Time of Retrieving Web-pages

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.

4.2.4 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 3.5.1. 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 4.5. As shown in Fig. 4.10, 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 a throughput decrease of $\sim$ 8.4 Gbps down to $\sim$ 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

Table 4.5: 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

throughput.

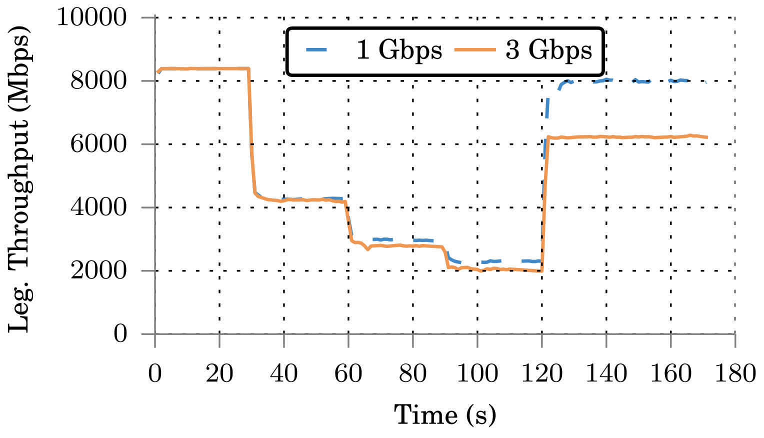


Figure 4.10: Recovering Legitimate Traffic Throughput

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The dynamicity of CDN edge-server environments requires that the systems deployed to protect them are also dynamic. It is also important that such protection system have low overhead not to adversely affect legitimate user traffic. In this thesis, we have 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. Our system is capable of accurately generating alerts for known attacks as well as generating fairly accurate alerts for unseen attacks using a proposed hybrid classification clustering method and anomaly detection approaches. Upon, detecting attacks, the system generates alerts that might trigger policies. With our hybrid method, the recall for known attacks ranges from 96.7% to 99.9%, and the recall for the new attack classes increases from 0% to 56% compared to a bagging classifier. Using an autoencoder based anomaly detection method, our system can detect 76.7% of anomalies which improves the isolation forest anomaly recall by 12%. The evaluation results demonstrate that our system has low overhead in terms of chain deployment time and latency of traffic passing through a chain. Our mitigation system is able to immediately respond to threats and quickly recover legitimate traffic throughput (~1 second). Using our system, legitimate traffic can be exempted from the high overhead imposed by heavy security services. To do so, security policies dictate redirecting only suspicious traffic to security chains while legitimate traffic is directly served without inspection. Accordingly, we have shown that legitimate requests are served 2.4× faster than in the case of static security services.

.

5.2 Future Work

Here, we present the following possible extensions to our work:

Orchestration: In the current implementation of the orchestrator, conflicting and redundant rules might be applied. For example, a rule that is triggered to build a chain for a subset of a traffic can override an already deployed chain for the traffic. There should be methods to ensure policy consistency through formal verification. Additionally, we have studied how to reduce the signaling overhead in the Orchestration process. As well as the ECA defined actions based on which the VIM is commanded to instantiate chains of various service functions, we tested delegating part of the service function chain management. An enabler for this purpose is the NSH protocol, which can carry metadata between different service functions. Thus, a decision from the Orchestrator may be propagated among the various service functions without full involvement of the Orchestrator in every decision. This type of delegation is challenging and remains a work in progress.

Attack methods: As part of the traffic monitoring analytics, we have proposed a hybrid learning method. This method uses a traditional clustering mechanism, namely k-means. Recently in the field of deep learning, variational and adversarial autoencoders are proposed for unsupervised clustering. In these models, the latent space is forced to have a normal mixed Gaussian distribution by which different clusters are separated. As future work, k-means can be replaced by these models.

References

[1] 400gbps: Winter of whopping weekend ddos attacks. https://goo.gl/XwQuL4.

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

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

[4] Cloud ddos protection service. https://goo.gl/hDGE9M.

[5] Ddos security trends report - state of the internet. https://goo.gl/ZSDcSu.

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

[7] Flowmeter. http://www.unb.ca/cic/datasets/flowmeter.html.

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

[9] Ibm appscan source. https://goo.gl/xa3yeM.

[10] Imperva incapsula ddos protection. https://goo.gl/6z9hwm.

[11] The imperva incapsula network ops ddos playbook. https://goo.gl/Z4R5Cx.

[12] Infrastructure ddos protection | bgp routing | incapsula. https://goo.gl/qMyhXy.

[13] Is your website available? https://goo.gl/h1Z5r1.

[14] Linux containers. https://linuxcontainers.org. Accessed: 2017-10-05.

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

[16] Newton release of openstack. https://www.openstack.org/software/newton/. Accessed: 2017-10-05.

[17] Ocata release of openstack. https://releases.openstack.org/ocata/. Accessed: 2017-10-05.

[18] Odl service function chaining. https://wiki.opendaylight.org/view/Service_Function_Chaining:Main. Accessed: 2017-10-05.

[19] Open network operating system (onos). http://onosproject.org. Accessed: 2017-10-05.

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

[21] Opendaylight. https://www.opendaylight.org. Accessed: 2017-10-05.

[22] Openstack. https://www.openstack.org.

[23] Openstack compute (nova). https://docs.openstack.org/nova/latest/. Accessed: 2017-10-05.

[24] Openstack neutron. https://wiki.openstack.org/wiki/Neutron. Accessed: 2017-10-05.

[25] Ovs nsh patches. https://github.com/yyang13/ovs_nsh_patches. Accessed: 2017-10-05.

[26] Owasp top 10-2013. https://goo.gl/Ne1M5e. Accessed: 2017-02-28.

[27] Production-grade container orchestration. https://kubernetes.io. Accessed: 2017-10-05.

[28] Web application firewall: More than web security. https://goo.gl/SZvMbp.

[29] Web application firewall (waf) | application security | incapsula. https://goo.gl/Dv5LQf.

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

[31] Wordpress default leaves millions of sites exploitable for ddos attacks. https://goo.gl/jVwoP7.

[32] Zun. https://wiki.openstack.org/wiki/Zun. Accessed: 2017-10-05.

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

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

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

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

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

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

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

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

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

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

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

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

[45] Deepak Nadig Anantha. Sdn service function chaining with onos and devstack. `https://deepaknadig.com/blog/sdn-sfc-onos-devstack/`. Accessed: 2017-10-05.

[46] Chitta Baral, Jorge Lobo, and Goce Trajcevski. *Formal characterizations of active databases: Part II*, pages 247–264. Springer Berlin Heidelberg, 1997.

[47] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita. Network anomaly detection: Methods, systems and tools. *IEEE Communications Surveys Tutorials*, 16(1):303–336, First 2014.

[48] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. Lof: Identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, May 2000.

[49] B. Cafarelli. Service function chaining demo with devstack. `http://blog.cafarelli.fr/2016/11/service-function-chaining-demo-with-devstack/`. Accessed: 2017-10-05.

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

[51] G. Carofiglio, G. Morabito, L. Muscariello, I. Solis, and M. Varvello. From content delivery today to information centric networking. *Computer Networks*, 57(16):3116 – 3127, 2013. Information Centric Networking.

[52] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[53] Jianjun Chen, Tao Wan, and Vern Paxson. Forwarding-loop attacks in content delivery networks. In *the 23st Annual Network and Distributed System Security Symposium*, 2016.

[54] Bram Cohen. Alibaba cloud mobile security service is an online mobile application security service that protects applications from potential risks, threats and vulnerabilities. `https://goo.gl/wnrXx6`.

[55] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

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

[57] B. Davie and J. Gross. A stateless transport tunneling protocol for network virtualization (stt). Rfc, RFC Editor, April 2016.

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

[59] Sarah M. Erfani, Sutharshan Rajasegarar, Shanika Karunasekera, and Christopher Leckie. High-dimensional and large-scale anomaly detection using a linear one-class svm with deep learning. *Pattern Recognition*, 58:121 – 134, 2016.

[60] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *USENIX Conference on Security Symposium*. USENIX Association, 2015.

[61] P. GarcÃŋa-Teodoro, J. DÃŋaz-Verdejo, G. MaciÃą-FernÃąndez, and E. VÃązquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers and Security*, 28(1):18–28, 2009.

[62] P. Garg and Y. Wang. Nvgre: Network virtualization using generic routing encapsulation. RFC 7637, RFC Editor, September 2015.

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

[64] A. Gharib, I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani. An evaluation framework for intrusion detection dataset. In *2016 International Conference on Information Science and Security (ICISS)*, pages 1–6, Dec 2016.

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

[66] J. Gross, I. Ganga, and T. Sridhar. Geneve: Generic network virtualization encapsulation. Rfc, RFC Editor, September 2017.

[67] Joel M. Halpern and Carlos Pignataro. Service Function Chaining (SFC) Architecture. RFC 7665, October 2015.

[68] Weili Han and Chang Lei. A survey on policy languages in network and security management. *Computer Networks*, 2012.

[69] S. HomChaudhuri and M. Foschiano. Cisco systems' private vlans: Scalable security in a multi-client environment. RFC 5517, RFC Editor, February 2010.

[70] 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.

[71] E. Jalalpour, M. Ghaznavi, D. Migault, S. Preda, M. Pourzandi, and R. Boutaba. Dynamic security orchestration for cdn edge-servers. In *2018 IEEE Conference on Network Softwarization (NetSoft)*, June 2018.

[72] E. Jalalpour, M. Ghaznavi, D. Migault, S. Preda, M. Pourzandi, and R. Boutaba. A security orchestration system for cdn edge servers. In *2018 IEEE Conference on Network Softwarization (NetSoft)*, June 2018.

[73] Ahmad Javaid, Quamar Niyaz, Weiqing Sun, and Mansoor Alam. A deep learning approach for network intrusion detection system. In *Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (Formerly BIONETICS)*, BICT'15, pages 21–26, ICST, Brussels, Belgium, Belgium, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[74] Q. Jia, H. Wang, D. Fleck, F. Li, A. Stavrou, and W. Powell. Catch me if you can: A cloud-enabled ddos defense. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 264–275, June 2014.

[75] I. Lazar and W. Terrill. Exploring content delivery networking. *IT Professional*, 3(4):47–49, Jul 2001.

[76] Aleksandar Lazarevic, Levent Ertöz, Vipin Kumar, Aysel Ozgur, and Jaideep Srivastava. A Comparative Study of Anomaly Detection Schemes in Network Intrusion Detection. In *Proceedings of the Third SIAM International Conference on Data Mining*, 2003.

[77] Kang-Won Lee, S. Chari, A. Shaikh, S. Sahu, and Pau-Chen Cheng. Protecting content distribution networks from denial of service attacks. In *IEEE International Conference on Communications, 2005. ICC 2005. 2005*, volume 2, pages 830–836 Vol. 2, May 2005.

[78] Jun Li, Skyler Berg, Mingwei Zhang, Peter Reiher, and Tao Wei. Drawbridge: Software-defined ddos-resistant traffic engineering. In *ACM*. ACM, 2014.

[79] Wei-Chao Lin, Shih-Wen Ke, and Chih-Fong Tsai. Cann: An intrusion detection system based on combining cluster centers and nearest neighbors. *Knowledge-Based Systems*, 78:13 – 21, 2015.

[80] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation-based anomaly detection. *ACM Trans. Knowl. Discov. Data*, 6(1):3:1–3:39, March 2012.

[81] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (vxlan): A framework for

overlaying virtualized layer 2 networks over layer 3 networks. RFC 7348, RFC Editor, August 2014. `http://www.rfc-editor.org/rfc/rfc7348.txt`.

[82] Gregor Maier, Anja Feldmann, Vern Paxson, and Mark Allman. On dominant characteristics of residential broadband internet traffic. In *ACM IMC*. ACM, 2009.

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

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

[85] Mindi McDowell. Understanding denial-of-service attacks. `https://www.us-cert.gov/ncas/tips/ST04-015`, 2013.

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

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

[88] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, August 2010.

[89] M. PATHAN. A taxonomy of cdns. *Content Delivery Netowrks, LNEE*, 9, 2008.

[90] E. L. Paula, M. Ladeira, R. N. Carvalho, and T. MarzagÃčo. Deep learning anomaly detection as support fraud investigation in brazilian exports and anti-money laundering. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 954–960, Dec 2016.

[91] Vern Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *SIGCOMM Comput. Commun. Rev.*, pages 38–47, 2001.

[92] Ingmar Poese, Benjamin Frank, Bernhard Ager, Georgios Smaragdakis, and Anja Feldmann. Improving content delivery using provider-aided distance information. In *ACM IMC*. ACM, 2010.

[93] Stefan Prandl, Mihai Lazarescu, and Duc-Son Pham. *A Study of Web Application Firewall Solutions*, pages 501–510. Springer International Publishing, Cham, 2015.

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

[95] Matthew Prince. The ddos that almost broke the internet. `https://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet`, 2013.

[96] P. Quinn, U. Elzur, and C. Pignataro. Network service header (nsh). Rfc, RFC Editor, October 2017.

[97] Paul Quinn, Uri Elzur, and Carlos Pignataro. Network Service Header (NSH). Internet-Draft draft-ietf-sfc-nsh-28, IETF, 2017. Work in Progress.

[98] Jamal Raiyn et al. A survey of cyber attack detection strategies. *International Journal of Security and Its Applications*, 8(1):247–256, 2014.

[99] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. RFC 3031, RFC Editor, January 2001. `http://www.rfc-editor.org/rfc/rfc3031.txt`.

[100] F. Sabahi and A. Movaghar. Intrusion detection: A survey. In *2008 Third International Conference on Systems and Networks Communications*, pages 23–26, Oct 2008.

[101] Thomas Schlegl, Philipp Seeböck, Sebastian M. Waldstein, Ursula Schmidt-Erfurth, and Georg Langs. Unsupervised anomaly detection with generative adversarial networks to guide marker discovery. In Marc Niethammer, Martin Styner, Stephen Aylward, Hongtu Zhu, Ipek Oguz, Pew-Thian Yap, and Dinggang Shen, editors, *Information Processing in Medical Imaging*, pages 146–157, Cham, 2017. Springer International Publishing.

[102] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[103] Colin Tankard. Advanced persistent threats and how to monitor and deter them. *Network Security*, 2011(8):16 – 19, 2011.

[104] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In Vittorio Cortellessa and Dániel Varró, editors, *Fundamental Approaches to Software Engineering*, pages 210–225, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[105] Sipat Triukose, Zakaria Al-Qudah, and Michael Rabinovich. Content delivery networks: protection or threat? In *ESORIS*. Springer, 2009.

[106] Limin Wang, KyoungSoo Park, Ruoming Pang, Vivek S Pai, and Larry L Peterson. Reliability and security in the codeen content distribution network. In *USENIX Annual Technical Conference, General Track*, pages 171–184, 2004.

[107] Candid Wueest. The continued rise of ddos attacks. `https://goo.gl/EuLPr2`, 2013.

[108] Yi Xie and Shun-Zheng Yu. Monitoring the application-layer ddos attacks for popular websites. *IEEE/ACM Trans. Netw.*, 2009.

[109] Dan Xu, Elisa Ricci, Yan Yan, Jingkuan Song, and Nicu Sebe. Learning deep representations of appearance and motion for anomalous event detection. *CoRR*, abs/1510.01553, 2015.

[110] Xiaowei Yang, David Wetherall, and Thomas Anderson. Tva: A dos-limiting network architecture. *IEEE/ACM Trans. Netw.*, 16(6):1267–1280, December 2008.

[111] Minlan Yu, Ying Zhang, Jelena Mirkovic, and Abdulla Alwabel. Senss: Software defined security service. In *ONS*, Santa Clara, CA, 2014. USENIX.