

Intrusion Detection in OpenStack using Snort and custom implementation

submitted by

Arka Das

Class Roll No: 002210503046

Exam Roll: MCA244045

Master of Computer Application

under the supervision of

Prof. Chandan Mazumdar

in the

Department of Computer Science and Engineering

Faculty of Engineering and Technology

Jadavpur University

2024

Faculty of Engineering and Technology
Jadavpur University

CERTIFICATE OF RECOMMENDATION

To Whom It May Concern

I hereby recommend that the project report entitled “Intrusion Detection in OpenStack using Snort and custom implementation” has been carried out by Arka Das (Reg. No.: 163638 of 2022-23, Roll No.: 002210503046, Exam Roll: MCA244045), under my guidance and supervision and be accepted in partial fulfilment of the requirement for the degree of Master of COMPUTER APPLICATION in DEPARTMENT of COMPUTER SCIENCE and ENGINEERING, JADAVPUR UNIVERSITY during the academic year 2023-2024.

Prof. Chandan Mazumdar
Project Supervisor
Dept. of Computer Science and
Engineering
Jadavpur University

Prof. (Dr.) Nandini Mukhopadhyay
Head of the Department
Dept. of Computer Science & Engineering
Jadavpur University

Prof. Dipak Laha
Dean, Faculty Council
of Engineering & Technology
Jadavpur University, Kolkata-032

Faculty of Engineering and Technology
Jadavpur University

CERTIFICATE OF APPROVAL

This is to certify that the project report entitled **“Intrusion Detection in OpenStack using Snort and custom implementation”** is a bonafide record of work carried out by Arka Das, in fulfillment of the requirements for the award of the degree of Master of Computer Application in the Department of Computer Science and Engineering, Jadavpur University. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the project report only for the purpose for which it has been submitted.

Signature of Examiner 1

Date :

Signature of Examiner 2

Date :

Faculty of Engineering and Technology
Jadavpur University

**DECLARATION OF COMPLIANCE WITH
ACADEMIC ETHICS**

I hereby declare that this project report entitled " Intrusion Detection in OpenStack using Snort and custom implementation " is an outcome of my genuine and independent work, as part of my Master of Computer Application program.

All information in this document has been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, all materials and results that are not original to this work have been fully cited and referenced.

Arka Das

Registration No.: 163638 of 2022-23

Examination Roll No.: MCA244045

Thesis Title :- Intrusion Detection in OpenStack using Snort and custom implementation

Signature with Date

ACKNOWLEDGEMENT

With my most sincere gratitude, I would like to thank Prof. Chandan Mazumdar, Department of Computer Science & Engineering, my supervisor, for his overwhelming support throughout the duration of the project. His motivation always gave me the required inputs and momentum to continue with my work, without which the project work would not have taken its current shape. His valuable suggestions and numerous discussions have always inspired new ways of thinking. I feel deeply honored that I got this opportunity to work under him.

I would like to express my sincere thanks to all my teachers for providing a sound knowledge base and cooperation.

I am particularly grateful to Dr. Srijita Basu, Dr. Anirban Sengupta, and the research scholars of CDC JU lab for their seamless cooperation and support throughout this project. Their assistance was instrumental in achieving its successful completion.

I would like to thank all the faculty members of the Department of Computer Science & Engineering of Jadavpur University for their continuous support.

Last, but not least, I would like to thank my batch mates for staying by my side when I needed them.

Arka Das

Class Roll No.: 002210503046

Examination Roll No.: MCA244045

Department of Computer Science & Engineering
Jadavpur University

Abstract

The growing adoption of cloud storage by businesses has amplified security concerns within cloud applications. As reliance on cloud computing surges, robust security measures become ever more critical. OpenStack, a widely used open-source cloud platform, offers Infrastructure as a Service (IaaS), attracting a large user base. To safeguard its virtualized environment, OpenStack necessitates effective intrusion detection systems (IDS). Furthermore, recent years have witnessed a surge in interest surrounding the implementation and testing of Virtual Network Functions (VNFs) built upon the OpenStack platform. VNFs are a cornerstone of Network Functions Virtualization (NFV), a technology that aims to revolutionize network infrastructure and service delivery by transitioning network functions from hardware to software, fostering greater efficiency.

The project is focused on deploying Virtual Network Functions (VNFs) on OpenStack, a popular open-source cloud platform. To achieve this, the project undertakes a deep dive into the functionalities of various OpenStack components, including Nova (compute service), Neutron (networking), and Cinder (block storage). This comprehensive understanding of each component's role is crucial for successful VNF deployment within the OpenStack environment.

Along with deployment of NFV this project talks about securing OpenStack cloud applications, using intrusion detection software. First, it examines how an Intrusion Detection System (IDS) can be implemented within an NFV framework, discussing the required architecture and potential challenges. The project then focuses on integrating Snort, a popular open-source NIDS, as a VNF within OpenStack. It highlights Snort's advantages, such as its extensive rule set and scalability, and details the deployment approach for this integration. Furthermore, the project explores a custom, user-specific NIDS designed for low-resource systems where Snort might be too demanding. This custom NIDS underwent development, testing, and performance evaluation through graphical visualizations to compare its effectiveness against Snort.

Contents

1. Introduction	9
i. Introduction to OpenStack.....	10
ii. OpenStack architecture.....	11
iii. OpenStack components	12
iv. Motivation of the project	17
v. Structure of the project.....	18
2. Introduction to Intrusion Detection System	19
i. Need of IDS.....	20
ii. Types of IDS	20
iii. Intrusion detection techniques.....	22
iv. Common IDS tools	24
3. Network Function Virtualization	25
i. NFV ETSI architecture	26
ii. NFV Infrastructure (NFVI)	27
iii. NFV Management and Orchestration (NFV MANO)	27
4. Installation and deployment	29
i. OpenStack installation with Devstack	29
ii. Configuration of cloud image and instances	33
iii. Implementation and deployment of VNF.....	36
iv. Testing VNF deployment	41
5. Configuration of Snort in VNF	42
i. Snort configuration file	42
ii. Snort rule structure	43
iii. Snort set up	44
iv. Snort operation and monitoring	46
6. Implementation of custom IDS	48
i. Custom IDS packet processing workflow	49
ii. Performance comparison with Snort	50
7. Conclusion	52
References	

List of figures

1. Figure 1: OpenStack logical architecture.....	11
2. Figure 2: OpenStack map.....	12
3. Figure 3: General IDS architecture	19
4. Figure 4: NFV vs traditional network	25
5. Figure 5: NFV ETSI architecture	26
6. Figure 6: OpenStack dashboard log in page	32
7. Figure 7: OpenStack dashboard	32
8. Figure 8: Cloud image uploading	34
9. Figure 9: Created OpenStack instances	36
10. Figure 10: VIM register output.....	37
11. Figure 11: VNFD creation output.....	38
12. Figure 12: VNF creation output	38
13. Figure 13: VIM management	38
14. Figure 14: Network topology	39
15. Figure 15: Snort rule structure	43
16. Figure 16: Snort version	44
17. Figure 17: Snort local rules.....	45
18. Figure 18, 19, 20: Snort generated alerts	47
19. Figure 21, 22, 23: Snort vs custom implementation results.....	50

1. Introduction

Self-hosted cloud solutions such as OpenStack [1] have been extensively developed as cloud computing becomes increasingly popular. They provide multiple services to meet cloud computing's needs in computing, storage, and networking. OpenStack is a free, open-source platform that lets you manage this pool and create virtual machines (VMs) and other resources on demand.

In recent years networks have been subjected to increased traffic demands as both businesses and consumers exponentially rely on network connectivity. Modern or traditional network functions rely on dedicated hardware appliances which have several major drawbacks like, cost, scalability, and flexibility. That is when VNFs [2] came into picture. VNF stands for Virtual Network Function. It's basically a software program that performs network-related tasks, but it runs on virtual machines instead of dedicated hardware. VNFs are a core concept in Network Functions Virtualization (NFV [2]), which aims to move network functions from hardware to software for greater efficiency. As VNFs are software based they serve as a wide variety of network components such as firewall, IDS [6], IPS, load balancers etc.

However, NFVs and cloud VMs often lack proper network security features to protect themselves from external cyber-attacks. Moreover, according to the 2022 Verizon data breach investigations report [8], 18% of attacks involved insiders.

Inside attackers can exploit cloud networking vulnerabilities against other tenants by port scanning, traffic sniffing, and traffic spoofing, and so on. To protect cloud tenants from both external and internal attacks, cloud administrators can apply network intrusion detection systems (NIDSes) such as Snort [7] and Suricata [9] to detect malicious traffic. Using these Intrusion detection systems cloud administrators can detect both internal and external attacks and can block the attempt by updating the rules accordingly.

In this project, we proposed a way to implement an Intrusion Detection System (IDS) as a VNF to monitor network traffic for malicious activity and take appropriate actions. In our approach the VNF serves as a gateway between the client network and server network which enables us to properly analyze both incoming and outgoing packets for abnormal activity thus in turn secures the cloud environment by alerting the cloud administrator or by blocking the attack.

1.1 Introduction to OpenStack:

OpenStack has emerged as a leading open-source cloud computing platform, enabling the creation and management of on-demand cloud infrastructure. OpenStack acts as the expert, orchestrating resources like storage, computing, networking etc to deliver self-service cloud services. At its core, OpenStack empowers users to provision virtual machines, storage, and networking resources through a user-friendly web interface or programmatic APIs.

Technically, OpenStack leverages a modular architecture composed of independent software programs, each offering specific functionalities [10]. These core services, like Nova for compute management and Cinder for block storage, work together seamlessly to provide the fundamental building blocks for Infrastructure as a Service (IaaS). A key strength of OpenStack lies in its distributed architecture. Each OpenStack service runs on separate servers, facilitating horizontal scaling. This allows for the addition of more machines to handle increased workloads, ensuring high availability and scalability to meet growing demands. Furthermore, OpenStack embraces open APIs, a rich set of programmatic interfaces that empower developers to automate cloud resource management and integrate OpenStack with custom applications and tools. This fosters automation and streamlines workflows within cloud environments. Finally, OpenStack boasts flexibility in hypervisor support. Hypervisors are software programs responsible for creating and managing virtual machines.

A brief historical perspective sheds light on OpenStack's impressive trajectory. The story begins in 2010 with a collaborative effort between Rackspace Hosting, a cloud service provider, and NASA, the US space agency. Rackspace envisioned open sourcing its cloud infrastructure code, while NASA was actively developing a platform for managing its compute resources (code-named Nova). This fruitful collaboration led to the birth of OpenStack, aiming to establish a ubiquitous open-source cloud platform that would democratize access to cloud computing technologies. The initial OpenStack release in 2010 comprised just a few core services. [11]

However, it garnered rapid adoption, with new services and features continually being added in subsequent releases. By 2012, the OpenStack Foundation, a non-profit entity, was established to oversee the project's ongoing development and foster a thriving community of developers and contributors. [12]

Today, OpenStack thrives as a community-driven project, constantly evolving to meet the ever-changing demands of cloud computing. It remains a popular choice for building and managing both private and public cloud deployments, offering a cost-effective and flexible alternative to proprietary cloud platforms. The OpenStack ecosystem extends beyond the core services, encompassing numerous additional projects that broaden the platform's capabilities. In conclusion, OpenStack presents a powerful and versatile open-source platform for cloud computing. It empowers users to build, manage, and scale their cloud infrastructure with flexibility and control, making it a cornerstone technology in today's dynamic cloud computing landscape.

1.2 OpenStack architecture:

OpenStack has a modular architecture with various code names for its component projects. It is a powerful and highly configurable integration engine with a set of core projects. What makes OpenStack quite special is that it is a collection of multiple software and not a big piece of monolithic software. It consists of several independent parts called OpenStack services. Each service is designed to work together to provide a complete infrastructure as a service. And the integration between services is facilitated through public Application Programming Interfaces (API) that each service offers.

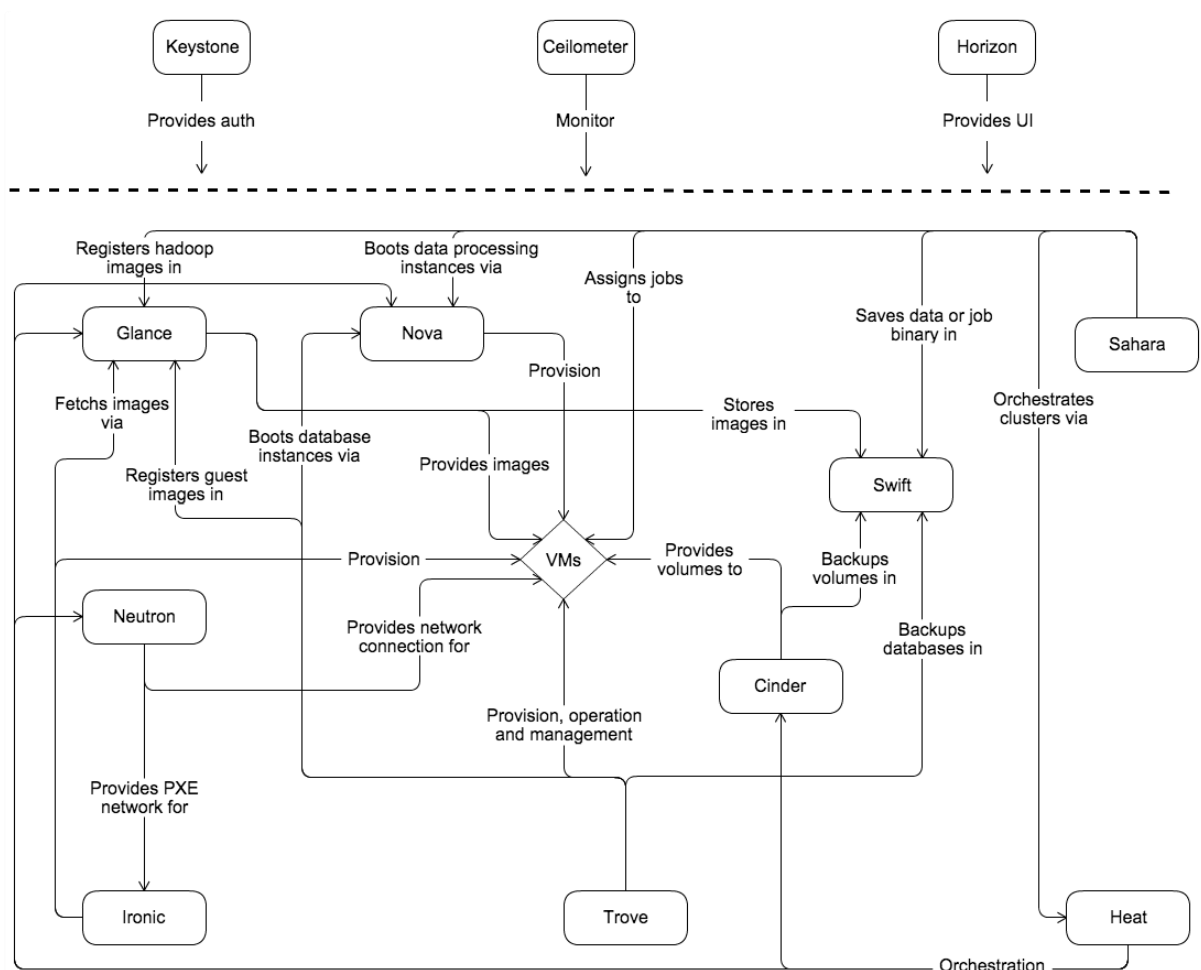


Figure 1: OpenStack logical architecture. (taken from docs.openstack.org)

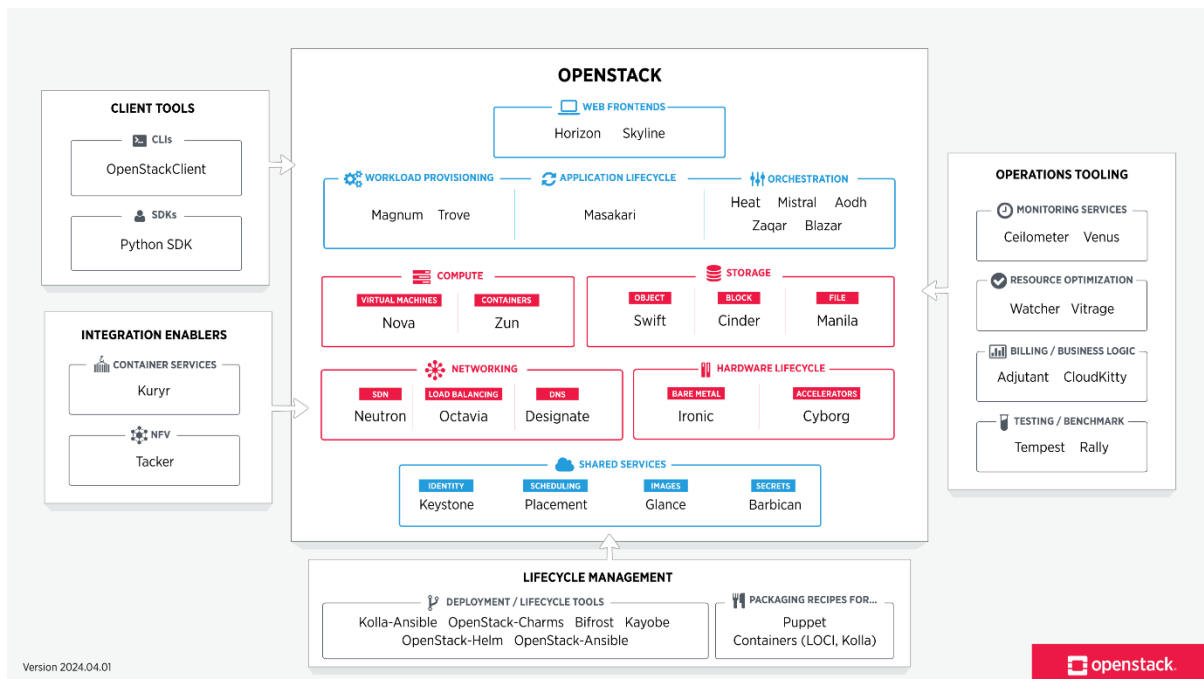


Figure 2: OpenStack map. (taken from <https://www.openstack.org/software>)

1.3 OpenStack components:

OpenStack Compute (nova):

Nova [3] is one of the first two original and complicated projects of OpenStack. Nova is designed to manage and automate pools of computing resources and can work with available virtualization technologies, as well as bare metal and high-performance computing configurations.

OpenStack Compute serves as the core of the OpenStack cloud by providing virtual machines on demand. Compute schedules virtual machines to run on a set of nodes by defining drivers that interact with underlying virtualization mechanisms, and by exposing the functionality to the other OpenStack components. Compute supports the libvirt [15] driver libvirtd [16] that uses KVM [17] as the hypervisor. Compute interacts with the Identity service to authenticate instance and database access, with the Image service to access images and launch instances, and with the dashboard service to provide user and administrative interface.

OpenStack nova is again divided into several components which handles some specific tasks.

openstack-nova-api: Handles requests and provides access to the Compute services, such as booting an instance.

openstack-novacompute: Runs on each node to create and terminate virtual instances. The compute service interacts with the hypervisor to launch new instances and ensures that the instance state is maintained in the Compute database.

openstack-novascheduler: Dispatches requests for new virtual machines to the correct node based on configured weights and filters.

Nova: Command-line client to access the Compute API.

OpenStack Networking (neutron)

Neutron [4] is the project under the OpenStack tenant.

OpenStack Networking handles creation and management of a virtual networking infrastructure in the OpenStack cloud. Infrastructure elements include networks, subnets, and routers. We can also deploy advanced services such as firewalls or virtual private networks (VPN). OpenStack Networking provides cloud administrators with flexibility to decide which individual services to run on which physical systems. Because OpenStack Networking is software-defined, it can react in real-time to changing network needs, such as creation and assignment of new IP addresses. It can provide advanced services like load balancing, VPN, firewall.

Neutron components

Network agent: Service that runs on each OpenStack node to perform local networking configuration for the node virtual machines and for networking services such as Open vSwitch.

neutron-dhcp-agent: Agent that provides DHCP services to tenant networks.

neutron-server: Python daemon that manages user requests and exposes the Networking API. The default server configuration uses a plug-in with a specific set of networking mechanisms to implement the Networking API. Certain plug-ins, such as the openvswitch and linuxbridge plug-ins, use native Linux networking mechanisms, while other plug-ins interface with external devices or SDN controllers.

Neutron: Command-line client to access the API

OpenStack Block Storage (cinder):

OpenStack Block Storage (Cinder [5]) provides persistent block storage management for virtual hard drives. Block Storage enables the user to create and delete block devices, and to manage attachment of block devices to servers. The actual attachment and detachment of devices is handled through integration with the Compute service. Volumes may be attached to a single instance at a time but maybe detached or re-attached to a different instance while

retaining all data much like a USB drive. Cinder virtualizes the management of block storage devices and provides end-users with a self-service API to request and consume those resources without requiring any knowledge of where their storage is deployed.

Components of Cinder:

openstack-cinder-api: Acts as a central hub for block storage requests. It receives user requests, verifies their identity, and translates them into messages specifying the desired storage action. These messages are then sent to a message broker for further processing by other Cinder services.

openstack-cinder-scheduler: This service acts as the director, assigning block storage tasks to the appropriate queue and determining the optimal server for provisioning volumes. It analyzes requests from the message queue and selects the most suitable Cinder Volume service based on specific criteria. It then communicates with the chosen service to handle the request.

openstack-cinder-volume: This service functions as the storage manager, interacting directly with block storage devices. It receives requests from the scheduler and performs actions like creating, modifying, or deleting volumes. It utilizes various drivers (e.g., NFS, Red Hat Storage, Dell EqualLogic) to communicate with different types of block storage devices.

OpenStack Identity (keystone)

OpenStack Identity provides user authentication and authorization to all OpenStack components. Identity supports multiple authentication mechanisms, including username and password credentials, token-based systems, and AWS-style log-ins. It is an essential service for building a straightforward cloud architecture. Keystone [18] helps in the creation, administration, and maintenance of user accounts within the Identity service. User accounts include associated information such as usernames and passwords. Role management is one of the important functions of Keystone, each role is assigned a specific set of permissions, thereby dictating user capabilities within the system.

OpenStack Image (glance)

OpenStack Image service (Glance [19]) acts as a registry for virtual disk images. Users can add new images or take a snapshot of an existing server for immediate storage. You can use the snapshots for backup or as templates for new servers. Registered images can be stored in the Object Storage service or in other locations, such as simple file systems or external Web servers.

The following image disk formats are supported [20]: **aki/ami/ari** (Amazon kernel, ramdisk, or machine image), **iso** (archive format for optical discs, such as CDs), **qcow2** (Qemu/KVM,

supports Copy on Write), **raw** (unstructured format), **vhd** (Hyper-V, common for virtual machine monitors from vendors such as VMware, Xen, Microsoft, and VirtualBox), **vdi** (Qemu/VirtualBox), **vmdk** (VMware)

Glance components:

openstack-glance-api: Interacts with storage back ends to handle requests for image retrieval and storage. The API uses openstack-glance-registry to retrieve image information. You must not access the registry service directly.

openstack-glance-registry: Manages all metadata for each image.

Glance: Command-line client to access the Image API.

OpenStack Object Storage (swift):

Object Storage (Swift [21]) provides an HTTP-accessible storage system for large amounts of data, including static entities such as videos, images, email messages, files, or VM images. Objects are stored as binaries on the underlying file system along with metadata stored in the extended attributes of each file. The Object Storage distributed architecture supports horizontal scaling as well as failover redundancy with software-based data replication. Because the service supports asynchronous and eventual consistency replication, you can use it in a multiple data-center deployment.

OpenStack Orchestration (heat):

OpenStack Orchestration (Heat [22]) provides templates to create and manage cloud resources such as storage, networking, instances, or applications. Templates are used to create stacks, which are collections of resources. According to Red Hat OpenStack Platform 9 Architecture Guide 20 [23] for example, you can create templates for instances, floating IPs, volumes, security groups, or users. Orchestration offers access to all OpenStack core services with a single modular template. Heat leverages IaC templates, written in YAML or HOT (Heat Orchestration Template) format, to define the desired cloud infrastructure. These templates specify the resources needed (e.g., VMs, storage volumes) and their configurations, allowing for a repeatable and automated approach to infrastructure provisioning. Templates can be recursively defined and reusable, such as nested stacks. The cloud infrastructure can then be defined and reused in a modular way.

OpenStack Dashboard (horizon):

Horizon [24] is the framework that provides the web interface for an easy management of instances and other OpenStack configuration. It is an ideal platform for end-users doing self-service, while the dashboard is the web interface built on top of that platform. OpenStack Dashboard provides a graphical user interface for users and administrators to perform

operations such as creating and launching instances, managing networking, and setting access control. The Dashboard service provides the Project, Admin, and Settings default dashboards. The modular design enables the dashboard to interface with other products such as billing, monitoring, and additional management tools. Horizon gets the end-users' instructions and reaches the specific OpenStack service via its API and gives the instruction to that service with the help of the restore interface.

OpenStack NFV Orchestration Service (Tacker):

Tacker [25] is a formally recognized OpenStack project dedicated to the development of a Generic VNF Manager (VNFM) and an NFV Orchestrator (NFVO). These components function collaboratively to deploy and manage Network Services and Virtual Network Functions (VNFs) within an NFV infrastructure platform, such as OpenStack. The project adheres to the design principles established by the ETSI MANO Architectural Framework [2]. Consequently, Tacker offers a comprehensive suite of functionalities that enable the end-to-end orchestration of Network Services through the utilization of VNFs.

European Telecommunications Standards Institute (ETSI) defines the Network Function Virtualization (NFV) architecture [8] as comprising of three main components or working domains [2], these three components are:

NFV Orchestrator (NFVO) - is responsible for on-boarding NS and VNF, and providing global resource management, validation, and authorization of NFV infrastructure (NFVI) resource requests.

VNF Manager (VNFM) - is responsible for the lifecycle management of VNF.

Virtualised Infrastructure Manager (VIM) - controls and manages compute, storage, and network resources.

Tacker furnishes a functional block that encompasses both a Network Functions Virtualization Orchestrator (NFVO) and a Generic VNF Manager (VNFM). The underlying infrastructure, known as the Virtual Infrastructure Manager (VIM), can be realized using an OpenStack cluster. In a specific scenario involving the deployment of a VNF instance, a VNF Package must first be prepared. This VNF Package encapsulates several key components, including the VNFD (Virtual Network Function Descriptor), software images for Virtual Machines (VMs), and additional artifact resources such as scripts and configuration files. The NFVO is invoked to initiate the deployment of a VNF instance using the designated VNF Package. Subsequently, the NFVO transmits a request to the VNFM, prompting the deployment of the actual VNF on the NFV infrastructure. It is essential to note that all interfaces leveraged throughout this process are clearly defined within the ETSI NFV specifications.

1.4 Motivation of the project:

The motivation for this project came from the ever-increasing need for security solutions within cloud environments, particularly those built upon the OpenStack platform. With this need in mind the project explores two complementary approaches aimed at strengthening cloud security posture.

The first approach uses the inherent flexibility of Network Function Virtualization (NFV) to integrate an Intrusion Detection System (IDS) within the OpenStack. This integrated solution actively monitors network traffic for malicious activity, significantly strengthening the overall security status of the cloud environment. The project specifically focused on the successful integration of Snort, a popular open-source NIDS, demonstrating its effectiveness in this context.

While Snort offers a valuable suite of functionalities, it can be resource-intensive, potentially posing a challenge for deployment on systems with limited resources. To address this concern, the project also focused on the development of a custom NIDS specifically designed for low-resource environments. This custom solution enables organizations to benefit from intrusion detection capabilities without compromising system performance, even on resource-constrained platforms.

By exploring these two distinct approaches, we enable organizations to make decisions regarding their cloud security posture. Organizations can choose to leverage the comprehensive protection offered by integrating an IDS within the NFV framework or opt for the lightweight and efficient custom NIDS on resource-limited systems.

1.5 Structure of the project:

With the introduction and motivation of this project in mind we can now take a deep dive into each of the portions of this project. This project has been properly structured into chapters; each one dedicated to one aspect of the project. A brief description of each chapter is provided here for a quick overview.

Chapter 2: Intrusion Detection Systems (IDS): This chapter lays a foundation for understanding IDS, its critical role in modern network security, the various detection techniques it employs, and a survey of popular IDS solutions.

Chapter 3: Network Function Virtualization (NFV): This chapter delves into the concept of NFV, explaining the ETSI architecture and its key components: Network Function Virtual Infrastructure (NFVI), NFV Management and Orchestration (NFV MANO), and Virtual Infrastructure Manager (VIM).

Chapter 4: OpenStack and VNF Deployment: This chapter details the installation process for OpenStack, including image and instance creation within the platform. It then progresses to explain the deployment of Virtual Network Functions (VNFs) within the OpenStack environment.

Chapter 5: Snort Integration in VNF: This chapter focuses on integrating Snort, a popular NIDS, into a VNF. It details the necessary Snort components, proper setup procedures, user-defined rule creation, and how to activate Snort's sniffing capabilities with example-based alert generation.

Chapter 6: Custom Lightweight IDS Implementation: This chapter explores the implementation of a custom, resource-efficient IDS solution. It details the implementation process, the system's packet processing workflow, and concludes by comparing its real-time alert generation performance against Snort.

2. Introduction to IDS (Intrusion Detection System)

Internet security has emerged as a significant challenge for recent organizations, particularly within cloud environments. To safeguard sensitive data, a well-established set of security technologies has been traditionally deployed, including web firewalls, encryption, authentication mechanisms, and Virtual Private Networks (VPNs). These measures serve to secure network infrastructure and communication channels traversing the internet. Intrusion Detection Systems (IDS) [6] represent a relatively recent addition to this security toolkit. Essentially, IDS technology represents a significant leap that enhances overall network security and reinforces an organization's data protection efforts. An IDS functions by continuously monitoring network traffic for malicious activity and promptly alerting network administrators of potential threats. This enables them to take appropriate actions to mitigate such attacks and safeguard sensitive data.

An intrusion, by definition, refers to any unauthorized attempt to access or exploit information resources. Intruders, or attackers as they are commonly known, are real-world entities that actively seek unauthorized access to information systems with the intent to cause harm or engage in other malicious activities. [29]

According to the National Institute of Standard and Technology (NIST), intrusion detection is the process of monitoring events occurring in a computer system or a network and analyze these events for a sign of intrusions. The monitoring processes can be accomplished with software or hardware to secure the system from malicious activity. This also protect integrated policies like firewall port configuration, data encryption, secure sockets layer (SSL) authentication, etc. that are being violated. [30]

Intrusion Detection Systems (IDS) serve as a crucial layer of security that complements firewalls. While firewalls act as the first line of defence, safeguarding against external threats attempting to infiltrate the network, IDS technology plays a complementary role. IDS focus on identifying and alerting about suspicious activities within the network. They don't actively prevent intrusions, but rather detect them in real-time or even before they reach the target system. This ensures that even if an attacker bypasses the firewall's defence's, their activities are still identified promptly, allowing for swift implementation of mitigation strategies to minimize potential damage.

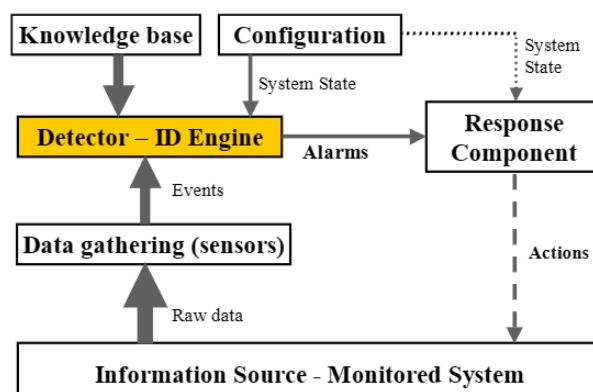


Figure 3: General IDS architecture. [34]

2.1 Need of IDS in cloud environments:

Cloud environments introduce a distinct set of security challenges that necessitate the implementation of Intrusion Detection Systems (IDS). These challenges can be categorized as follows:

- **Shared Responsibility Model:** Cloud providers assume responsibility for securing the underlying infrastructure; however, the responsibility of securing your data and workloads within the cloud environment falls upon you.
- **Expanded Attack Surface:** Cloud environments are inherently complex, encompassing a complex web of services, APIs, and microservices. This expanded attack surface presents a greater number of potential vulnerabilities and misconfigurations that attackers can exploit.
- **Dynamic Nature of Cloud Environments:** Cloud environments are renowned for their on-demand scalability, enabling the rapid provisioning of new resources. Cloud-based IDS solutions are specifically designed to adapt and scale automatically in response to these dynamic changes, ensuring the continuity of monitoring efforts.
- **Evolving Threat Landscape:** The threat landscape is constantly evolving, with the potential for entirely novel threats to emerge, even in the presence of robust security measures. Anomaly-based detection capabilities within IDS can prove invaluable in identifying such suspicious activities that deviate from established patterns of normal behaviour.
- **Internal Attackers:** Cloud environments are vulnerable to security breaches not only from external actors but also from internal threats. In fact, studies indicate that over 18% of security incidents involve insiders. Cloud-based Intrusion Detection Systems (IDS) play a critical role in mitigating this risk by actively detecting and generating alerts for malicious activities executed by internal users.

2.2 Types of Intrusion Detection System (IDS):

There are two types of Intrusion Detection systems. These are network-based Intrusion Detection System and host-based Intrusion Detection System.

Network Based Intrusion Detection System (NIDS):

Network-based Intrusion Detection Systems (NIDS) represent a class of intelligent devices strategically distributed within networks. These systems passively monitor and analyze traffic flowing through the network segments on which they reside. NIDS solutions can be implemented as either hardware or software-based deployments. Depending on the specific vendor, NIDS systems may be designed to connect to various network mediums, including Ethernet, FDDI, and others.

NIDS are typically installed at designated points within the network infrastructure. Common placements include routers or as part of a Virtual Network Function (VNF). This strategic positioning allows the NIDS to observe traffic entering and exiting a particular network segment. Additionally, NIDS can be configured to monitor specific host computers within a network segment or deployed to comprehensively monitor all traffic traversing the entire network.

The primary function of a NIDS involves thoroughly examining data packets for unusual patterns and behaviours that might signal an ongoing attack. NIDS systems are adept at detecting and alerting network administrators to a variety of threats, including Denial-of-Service (DoS) attacks, port scans, malware infections, and unauthorized access attempts. NIDS typically operate in either a passive or inline mode, employing various detection methods to identify network intrusions.

In passive mode, the NIDS functions as a dedicated observer, carefully monitoring outgoing network traffic without interfering with its flow. Conversely, inline mode empowers the NIDS to actively modify network traffic for the purpose of intrusion detection or even to directly block malicious activities. However, it is important to note that inline mode carries a potential risk of disrupting legitimate network traffic. Consequently, this mode is generally not recommended for deployment.

Upon detecting a potential network threat, the NIDS generates a detailed alert. This alert typically encompasses information such as the specific type of attack detected, the source and destination IP addresses involved, and the precise time of the attack. In some instances, the NIDS may also be programmed to take autonomous actions to mitigate the threat, such as blocking the offending source IP address or modifying network traffic in real-time.

Host Based Intrusion Detection System (HIDS):

In contrast to Network-based Intrusion Detection Systems (NIDS) that monitor network traffic, Host-Based Intrusion Detection Systems (HIDS) are deployed directly on individual computers or servers, functioning as dedicated guardians of those specific hosts. HIDS systems meticulously monitor the activity occurring on the host system they reside upon. This monitoring focuses on the status of critical system files, enabling the HIDS to detect any attempts by unauthorized users to create, modify, or delete these files.

Host-Based Intrusion Detection Systems function by actively collecting data from the servers, computers, and other host systems they are installed upon. This collected data is then meticulously analyzed for anomalies or any activity that deviates from established baselines and might indicate a potential security threat. The data sources including security-centric data, such as authentication logs that precisely record login events. HIDS solutions also commonly analyze other types of data, such as application and operating system logs. While these latter data types may not possess a direct correlation to security, the identification of unusual patterns within these seemingly innocuous data sets can prove invaluable in uncovering underlying security vulnerabilities.

2.3 Intrusion Detection Techniques:

Intrusion detection systems (IDS) employ a combination of techniques to identify malicious activity on a network or system. Here are the two main categories:

Signature-based intrusion detection systems (SIDS):

Signature-based Intrusion Detection Systems (SIDS) represent a class of systems that uses pattern-matching techniques to identify known attack patterns. This approach, also known as knowledge-based detection, relies on a comprehensive database of intrusion signatures. The core functionality of a SIDS involves meticulously comparing the current set of network activities against the established signatures within the database. If a match is identified, an alarm is triggered, alerting security personnel of a potential threat.

In essence, SIDS function by precisely matching the specific byte sequences of new packets against a library of existing byte sequences stored within the signature database. For instance, a SIDS rule might be structured as "source -> destination with (following packet details)". If the source IP address, destination IP address, and other specified packet details precisely match a signature within the database, an alert is promptly raised.

Advantages of Signature-based Intrusion Detection Systems:

- **High Accuracy for Known Threats:** SIDS excel at accurately detecting previously known attacks. This efficiency comes from the existence of a clear reference signature database that facilitates a direct comparison with current network activity.
- **Fast and Reliable Detection:** The ability to efficiently match pre-defined patterns empowers SIDS to deliver swift and dependable identification of known threats.
- **Relatively Low Maintenance:** While maintaining the signature database necessitates regular updates to incorporate the latest threats, the core operational principles of SIDS are relatively straightforward.

Disadvantages of Signature-based Intrusion Detection Systems:

- **Vulnerability to Zero-Day Attacks:** SIDS are inherently vulnerable to novel and unknown threats, often referred to as zero-day attacks. This vulnerability arises from the absence of corresponding signatures within the database for these new threats.
- **Signature Update Dependency:** The effectiveness of SIDS is heavily contingent upon keeping the signature database meticulously updated with the latest information on evolving threats. Outdated signature databases significantly compromise a SIDS' ability to detect contemporary threats.
- **Limited Detection of Multi-Packet Attacks:** Traditional SIDS solutions struggle to identify attacks that deliberately span multiple packets. As modern malware gets more sophisticated, it may fragment attack signatures across multiple packets to evade detection. This scenario requires the SIDS to recall the contents of previous packets for comprehensive analysis.

Anomaly-based intrusion detection systems (AIDS):

Anomaly-based Intrusion Detection Systems (AIDS) represent a distinct category of intrusion detection systems that leverage machine learning, statistical analysis, or knowledge-based techniques to establish a baseline model of normal computer system behaviour. This established model serves as a reference point for identifying deviations from typical activity. Any significant discrepancy between the observed system behaviour and the constructed model is flagged as an anomaly, potentially indicative of a malicious intrusion attempt.

The core principle underlying Anomaly-based IDS rests upon the assumption that malicious behaviour deviates from the patterns showed by typical user behaviour. The system continuously monitors system activity and classifies deviations from established baselines as intrusions.

The development of an Anomaly-based IDS typically encompasses two distinct phases: the training phase and the testing phase. During the training phase, the system uses a dataset of normal traffic patterns to construct a comprehensive model of normal behaviour. Subsequently, in the testing phase, a new and unseen dataset is introduced to evaluate the system's ability to generalize its knowledge and effectively detect previously unencountered intrusion attempts.

Advantages of Anomaly-based Intrusion Detection Systems (AIDS):

- **Detection of Zero-Day Attacks:** Anomaly-based IDS offers a significant advantage in identifying previously unknown threats, often referred to as zero-day attacks since it bypasses the need for a signature database of known attack patterns.
- **Signature Generation for Signature-based IDS:** The ability to detect novel attacks extends beyond immediate threat identification. Anomaly-based IDS can be used to generate intrusion signatures for SIDS. These newly created signatures can then be updated into the SIDS signature database, enabling SIDS to identify these threats in the future.

Disadvantages of Anomaly-based Intrusion Detection Systems:

- **High False Positive Rate:** Anomaly-based IDS can be susceptible to generating a high number of false positives. This occurs because the system may misinterpret new or unusual, yet legitimate, user activities as potential intrusions.
- **Increased Resource Requirements:** Anomaly-based IDS solutions often requires a significant amount processing power. This increased resource consumption arises because it requires processing vast amounts of data to identify unusual patterns.
- **Configuration Challenges:** The configuration and fine-tuning of Anomaly-based IDS is complex. This complexity comes from identifying the subjective normal behaviour associated a particular system. Furthermore, establishing a baseline model of normal behaviour for highly dynamic computer systems presents a significant challenge.

Additionally, the initial training phase required to establish the baseline model can complicate the initial setup process.

2.4 List of IDS tools:

Snort: Snort [7] is a free open source network intrusion detection system (IDS) and intrusion prevention system (IPS) created in 1998 by Martin Roesch, founder and former CTO of Sourcefire. Snort is now developed by Cisco, which purchased Sourcefire in 2013. In 2009, Snort entered InfoWorld's Open Source Hall of Fame as one of the "greatest [pieces of] open source software of all time".

The system analyzes traffic in real-time, through three different modes: packet sniffer, packet logger, and intrusion detection. It can also employ intrusion prevention techniques. Snort can detect OS fingerprinting, port scanning, SMB probes, and other cyber-attacks using both signature and anomaly detection methods.

Suricata: Suricata [9] is an open-source intrusion detection system (IDS) and intrusion prevention system (IPS). It was developed by the Open Information Security Foundation (OISF). A beta version was released in December 2009, with the first standard release following in July 2010.

Like Snort, Suricata uses signature and anomaly-based detection methods to identify intrusions. It combines intrusion prevention, network security monitoring, and PCAP processing to efficiently manage incoming attacks. Suricata can also examine TLS/SSL certificates, HTTP requests, and DNS transactions.

OSSEC: OSSEC [31] (Open Source HIDS SECurity) is a free, open-source host-based intrusion detection system (HIDS). It performs log analysis, integrity checking, Windows registry monitoring, rootkit detection, time-based alerting, and active response. It provides intrusion detection for most operating systems, including Linux, OpenBSD, FreeBSD, OS X, Solaris and Windows. OSSEC has a centralized, cross-platform architecture allowing multiple systems to be easily monitored and managed. OSSEC has a log analysis engine that can correlate and analyze logs from multiple devices and formats.

Sguil: Sguil [32] (pronounced *sgweel* or *squeal*) is a collection of free software components for Network Security Monitoring (NSM) and event driven analysis of IDS alerts. The sguil client is written in Tcl/Tk and can be run on any operating system that supports these. Sguil integrates alert data from Snort, session data from SANCP, and full content data from a second instance of Snort running in packet logger mode.

Security Onion: Security Onion [33] is an Ubuntu-based Linux distribution for IDS and network security monitoring. It provides a great interactive user interface or dashboard for easier monitoring. The platform combines several open-source technologies, such as Snort, Suricata, Zeek, and Sguil to provide comprehensive intrusion detection, network security monitoring, and log management.

3. Network Function Virtualisation (NFV)

Network Function Virtualization, or NFV [2], is a way to reduce cost and accelerate service deployment for network operators by decoupling functions like a firewall or encryption from dedicated hardware and moving them to virtual servers. Instead of installing expensive proprietary hardware, service providers can purchase inexpensive switches, storage, and servers to run virtual machines that perform network functions. This collapses multiple functions into a single physical server, reducing costs. Essentially, network functions virtualization replaces the functionality provided by individual hardware networking components. This means that virtual machines run software that accomplishes the same networking functions as the traditional hardware. Load balancing, routing and firewall security are all performed by software instead of hardware components.

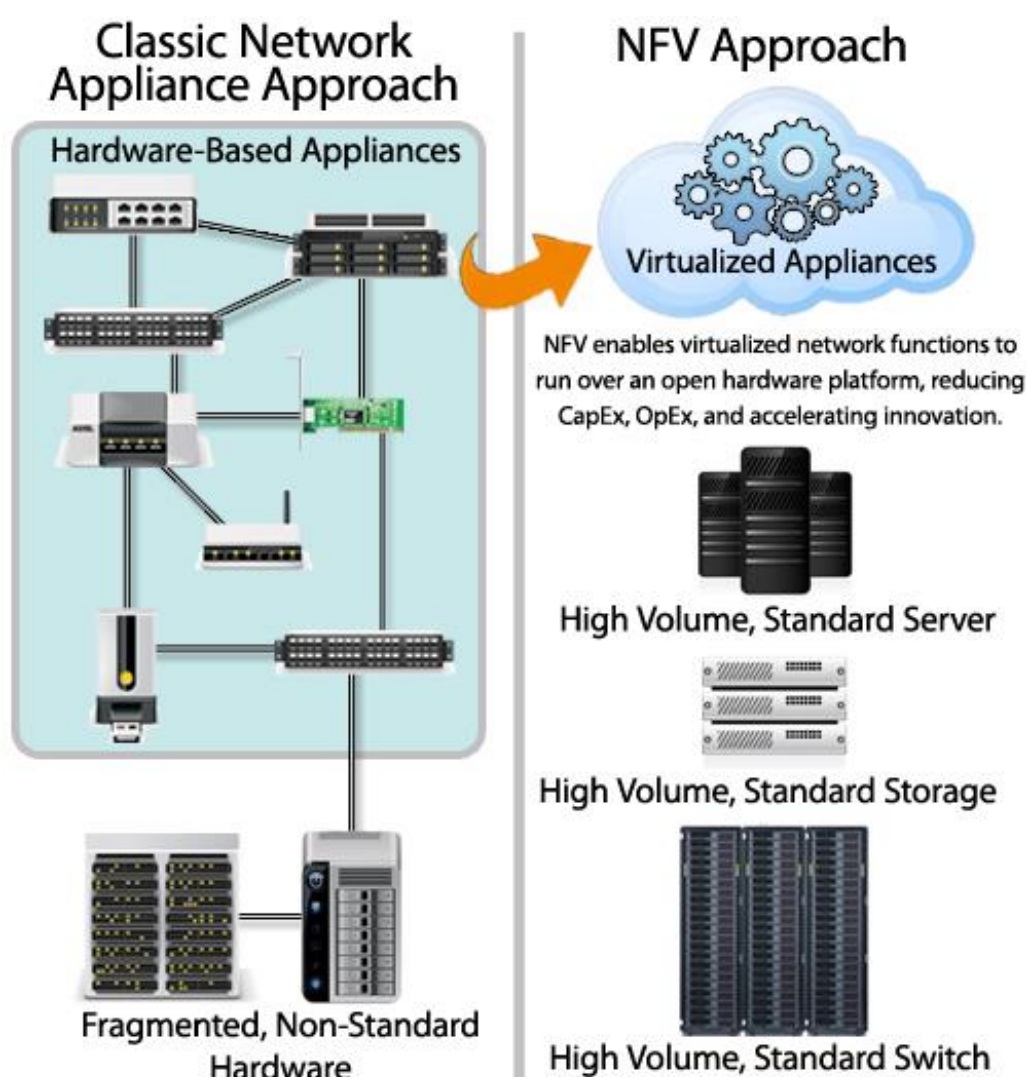


Figure 4: NFV vs traditional network. [37]

The provisioning of virtual services through software offers several advantages. Firstly, it facilitates the flexible deployment of network functions. This translates to a swifter deployment process compared to traditional methods. Secondly, the decoupling of software

functionalities from hardware affords greater flexibility in dynamically scaling VNF performance. This granular control over scaling enables more precise adjustments to meet fluctuating network demands.

3.1 NFV ETSI Architecture:

The European Telecommunications Standards Institute (ETSI) is an independent standardization group that develops standards for information and communications technologies (ICT) in Europe.

According to ETSI in general, a network functions virtualization (NFV) platform [26] has the following components:

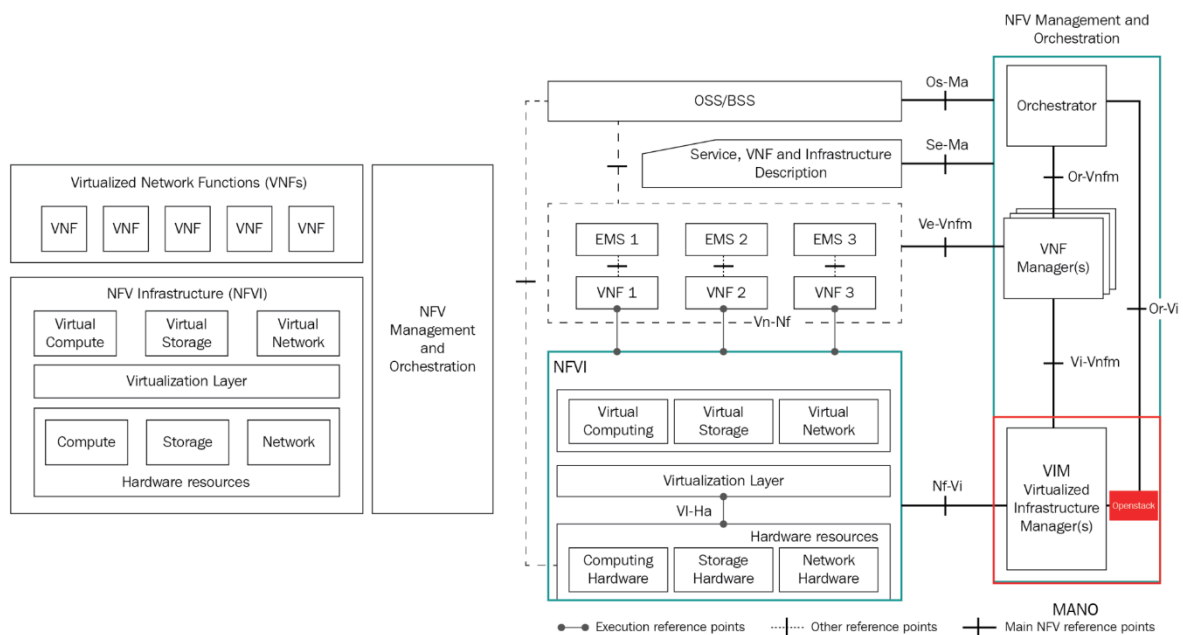


Figure 5: OpenStack and the ETSI NFV Architecture Framework (ETSI) [27]

Virtualized Network Functions (VNFs): These are the actual software implementations of the network functions like routers, firewalls, load balancers. These mimics the working of network hardware resources through software which can be deployed over NFVI. A virtual resource such as a virtual machine (VM) can quickly and easily deploy a virtual network function (VNF), which represents a software version of a network function. VNFs can be linked together like building blocks in a process known as service chaining. An NFV architecture facilitates the on-demand deployment of VNFs. This approach eliminates the delays inherent in the deployment of traditional network hardware. Additionally, it negates the necessity for on-site technical expertise when deploying VNFs remotely and in-turn eases the process of upscaling resources.

3.2 NFV Infrastructure (NFVi):

NFVi is the foundation of NFV, providing the physical resources like compute, storage, networking, and virtualization layer for building virtual networks. It offers a multi-tenant environment managed by a VIM (Virtual Infrastructure Manager). Notably, the network component within NFVi handles data forwarding between virtual machines, enabling VNF deployment without hardware knowledge. Additionally, Enhanced Platform Awareness (EPA) improves packet forwarding performance by exposing hardware acceleration directly to VNFs. NFVi has three major components:

Hardware Resources: These represent the physical compute, storage, and network resources. It encompasses compute, storage (disks, NAS, SAN), and network elements (routers, switches) that provide processing, storage, and connectivity for VNFs through the virtualization layer.

Virtualization Layer: The virtualization layer lies on the top of the hardware resources layer which abstracts and decouples the VNF software from the underlying hardware resources to ensure a hardware independent life cycle for VNFs. The middle virtualization layer ensures that VNFs are separated from hardware resources so that the software can be installed on various types of actual hardware resources.

Virtualized Resources: The virtualization layer functions as an abstraction layer, effectively decoupling the underlying physical hardware resources, including compute, storage, and networking, from the VNFs. This abstraction process makes these resources available to VNFs in the form of virtualized counterparts. It is important to note that while these virtual resources are presented to VNFs as distinct entities, they ultimately operate on top of the same shared physical infrastructure.

3.3 NFV Management and Orchestration (NFV MANO):

The management and orchestration layer, also known as MANO, assumes a critical role in managing the entire lifecycle of network functions. It decouples service management from the underlying hardware through a virtualized approach. This includes functionalities like service definition, automation, error handling, performance monitoring, and lifecycle management of network functions offered to customers. Notably, MANO achieves this decoupling from the underlying physical infrastructure. Achieving this decoupling requires the introduction of additional management layers, most prominently embodied by the Virtual Network Function Manager (VNFM). The VNFM assumes responsibility for the life-cycle management of virtual machines and VNFs. Another key component within the MANO framework is the orchestrator, also known as the NFVO. The NFVO acts as the central interface, interacting with various databases and systems, including Operations/Business Support Systems (OSS/BSS) at the upper layer and the VNFM at the lower layer.

NFV MANO itself comprises of three functional blocks, namely:

NFV Orchestrator (NFVO):

NFVO, a key component of NFV MANO, manages the entire lifecycle of network services and orchestrates resources for VNFs.

The NFVO acts as a central hub, interacting with various systems and potentially bypassing VIMs for direct resource management. It governs how VNF instances share resources and ensures efficient utilization.

The NFV Orchestrator responsibilities include:

- The orchestration of NFVI resources across multiple VIMs, fulfilling the Resource Orchestration functions.
- The lifecycle management of Network Services, fulfilling the Network Service Orchestration functions.
- VNF onboarding and lifecycle management
- Global resource management
- Authorization and validation of NFVI resource requests

VNF Manager (VNFM):

The Virtual Network Function Manager (VNFM) assumes primary responsibility for the management of the VNF lifecycle. This encompasses all stages, including instantiation, update, scaling, and termination. The VNFM is assigned to oversee multiple VNF instances concurrently. It is essential to clarify that while a single VNFM can manage numerous VNF instances, each VNFM is inherently linked to a single VNF descriptor (VNFD). The majority of VNFM solutions offer comprehensive support for a full range of management activities, including instantiation, updating, querying, scaling, and termination of VNF instances. The core functionalities provided by a VNFM include:

- Maintaining the Integrity of VNF Instances Throughout the Lifecycle.
- Lifecycle Management of Individual VNF Instances.
- Facilitating Coordination and Adaptation Between VIM and EM where the VNFM acts as the central coordinator, facilitating the exchange of configuration data between the Virtual Infrastructure Manager (VIM) and the Element Management System (EM).

Virtualized Infrastructure Manager (VIM):

VIM manages compute, storage, and network resources within the NFVI layer. It maintains an inventory of virtual resources mapped to physical ones, enabling optimal allocation, upgrade, release, and reclamation. VIM also manages VNF forwarding graphs (virtual links, networks, etc.) and enforces security group policies. Additionally, it maintains a repository of NFVI hardware and software resources for optimized resource utilization.

4. Installation and Deployment

4.1 OpenStack installation with Devstack:

OpenStack installation for this project has been done in a Virtual machine using VMWare workstation player. There are certain system requirements for installing OpenStack as it uses a lot of resources. The machine should have at least 16 gb of memory and 100 gb of disk storage with a minimum of 4-6 cores CPU and requires Linux distribution based on Debian that uses an Ubuntu LTS kernel. We are using Devstack [35] as Devstack is a series of scripts used to quickly bring up a complete OpenStack environment. The latest version of OpenStack is downloaded from the git master branch. It used to set up a faster and quicker way to set up the development environment and as the basis for most of the OpenStack project functional testing.

Here we have implemented NFV using a single all-in-one node which acts as the controller, compute, and storage node.

Installation steps:

1. Install Ubuntu 22.04 LTS or any other latest LTS version of Linux distribution.
2. **sudo apt-get update** - it fetches the latest version of the package list from your distro's software repository, and any third-party repositories you may have configured. In other words, it'll figure out what the latest version of each package and dependency is but will not actually download or install any of those updates.
3. **sudo apt-get install build-essential** - The build-essential package belongs to Debian. It is not a piece of software. It contains a list of packages that are required to create a Debian package (deb). These packages are libc, gcc, g++, make, dpkg-dev etc. The build-essential package contains those required packages as dependencies, so when we install build-essential, we install all those packages in one single command.
4. **sudo apt-get install net-tools** - this package includes the important tools for controlling the network subsystem of the Linux kernel. This includes arp, ifconfig, netstat, rarp, nameif and route. Additionally, this package contains utilities relating to network hardware types (plipconfig, slattach, mii-tool) and advanced aspects of IP configuration (iptunnel, ipmaddr).
5. **sudo apt-get install python3-pip** - Pip is a package management system that simplifies installation and management of software packages written in Python such as those found in the Python Package Index (PyPI).
6. **pip install -U os-testr A testr** - wrapper to provide functionality for OpenStack projects.
7. **sudo apt-get install autoconf automake gdb git libffi-dev zlib1g-dev libssl-dev -y** - Autoconf is a GPLv2-licensed package of macros that produces a shell script "configure" to generate a Makefile and config header for a project. Automake is a tool for automatically generating `Makefile.in's from files called `Makefile.am'. Tool for generating GNU Standardscompliant Makefiles. GDB is a source-level debugger,

capable of breaking programs at any specific line, displaying variable values, and determining where errors occurred. Currently, gdb supports C, C++, D, Objective-C, Fortran, Java, OpenCL C, Pascal, assembly, Modula-2, and Ada. Git is popular version control system designed to handle very large projects with speed and efficiency; it is used for many high-profile open-source projects, most notably the Linux kernel. fast, scalable, distributed revision control system. libffi-dev package contains the headers and static library files necessary for building programs which use libffi. zlib is a library implementing the deflate compression method found in gzip and PKZIP. This package includes the development support files. libssl-dev package is part of the OpenSSL project's implementation of the SSL and TLS cryptographic protocols for secure communication over the Internet.

8. **sudo useradd -s /bin/bash -d /opt/stack -m stack** -This command creates a new user named stack to the system. The user will access the root opt directory.
9. **sudo chmod +x /opt/stack** - It provides the execute permission to the directory.
10. **echo "stack ALL=(ALL) NOPASSWD: ALL" | sudo tee /etc/sudoers.d/stack** - No root password will be needed for the user stack.
11. **sudo su – stack** - This gives stack user as sudo user privilege also all the environment variables will be sourced to the user, stack
12. **git clone https://opendev.org/openstack/devstack -b stable/yoga** - This will clone the devstack folder from the link and it will clone the stable yoga branch.
13. **cd devstack** - Change the current directory to the devstack folder.
14. **nano local.conf** - Create local.conf file with all the credentials and the services that are needed to be installed. Using this configuration file, we can specify admin passwords for openstack and can choose which component or services we need. The local.conf file used here is given below:

```
[[local|localrc]]
HOST_IP=127.0.0.1
ADMIN_PASSWORD=devstack
MYSQL_PASSWORD=devstack
RABBIT_PASSWORD=devstack
SERVICE_PASSWORD=$ADMIN_PASSWORD
SERVICE_TOKEN=devstack
#####
# Customize the following section based on your installation
#####
# Pip
PIP_USE_MIRRORS=False
USE_GET_PIP=1
#OFFLINE=False
#RECLONE=True
# Logging
LOGFILE=$DEST/logs/stack.sh.log
VERBOSE=True
ENABLE_DEBUG_LOG_LEVEL=True
ENABLE_VERBOSE_LOG_LEVEL=True
# Disable security groups
LIBVIRT_FIREWALL_DRIVER=nova.virt.firewall.NoopFirewallDriver

# Enable heat, networking-sfc, barbican and mistral
```

```

enable_plugin heat https://opendev.org/openstack/heat
stable/yoga
enable_plugin networking-sfc
https://opendev.org/openstack/networking-sfc stable/yoga
enable_plugin barbican https://opendev.org/openstack/barbican
stable/yoga
enable_plugin mistral https://opendev.org/openstack/mistral
stable/yoga
# Ceilometer
#CEILOMETER_PIPELINE_INTERVAL=300
CEILOMETER_EVENT_ALARM=True
enable_plugin ceilometer
https://opendev.org/openstack/ceilometer stable/yoga
enable_plugin aodh https://opendev.org/openstack/aodh
stable/yoga
# Blazar
enable_plugin blazar https://github.com/openstack/blazar.git
stable/yoga
# Tacker
enable_plugin tacker https://opendev.org/openstack/tacker
stable/yoga
[[post-config|/etc/neutron/dhcp_agent.ini]]
[DEFAULT]
enable_isolated_metadata = True
Implementation and deployment of VNF
./stack

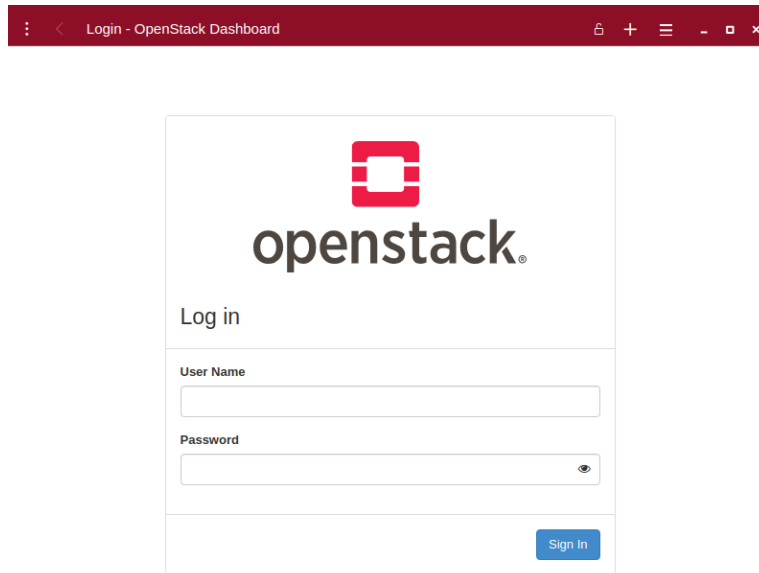
```

15. **./stack.sh** - This executes stack.sh file. ``stack.sh`` is a shell script and an opinionated OpenStack developer installation. It installs and configures various combinations of OpenStack services like Ceilometer, Cinder, Glance, Heat, Horizon, Keystone, Nova. local.conf file is sources by stack.sh

This last command takes 45-50 mins depending on the machine hardware and network bandwidth to run and may end in error due to unavailability of internet. If it ends in error, we need to run ./unstack.sh which stops all the process that started by stack.sh and then run ./stack.sh again.

At this point OpenStack has been successfully installed in the system. The horizon dashboard can be accessed now by going the following url:

<http://<Host Ip address>/dashboard>



The image shows the OpenStack Dashboard login page. At the top is a dark red header bar with the text "Login - OpenStack Dashboard" and navigation icons. Below the header is a white login box containing the OpenStack logo (a red square with a white 'O') and the text "openstack.". Underneath is the "Log in" heading. There are two input fields: "User Name" and "Password". The "Password" field has a toggle icon (an eye) to the right. At the bottom right of the login box is a blue "Sign In" button.

Figure 6: Dashboard login page.

Log in using the admin password given in the local.conf file and then you are in.

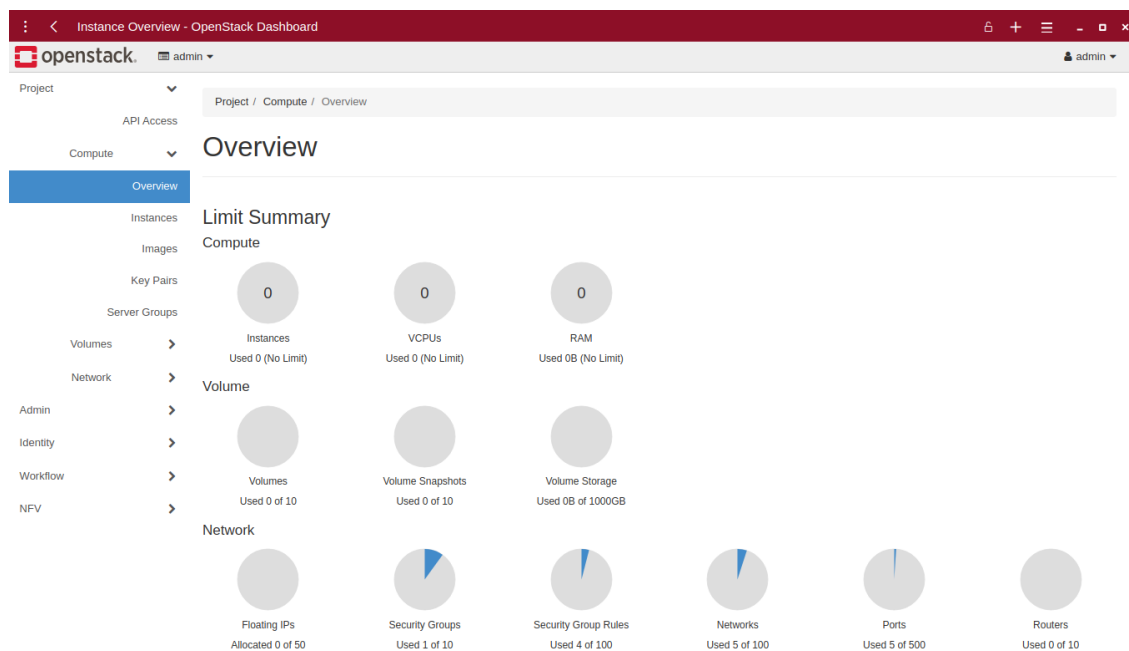


Figure 7: OpenStack dashboard.

4.2 Configuration of cloud image and instances:

OpenStack provides the capability to deploy instances, which are virtual machines that execute within the OpenStack cloud environment. These instances can be customized with user-specified operating systems and system configurations. This project, leveraged OpenStack to create instances utilizing the Ubuntu Server 18.04 LTS (Bionic Beaver) operating system, followed by a process of tailoring the image to align with our specific requirements.

1. Download cloud image: Download the latest Ubuntu Server image from <https://cloud-images.ubuntu.com/bionic/>
2. **sudo apt-get install libguestfs-tools** – This installs virt-customize tool which allows to modify the cloud image, like installing packages, setting admin password and a lot more.
3. **sudo virt-customize -a <cloud img file name>.img --run-command 'apt get update -y'** – it will upgrade the cloud image software packages to latest version.
4. **sudo virt-customize -a <cloud img file name>.img --install python3** – This command installs latest version of python in the cloud image.
5. **sudo virt-customize -a <cloud img file name>.img --install python3-scapy** – Install scapy, a powerful tool that offers a comprehensive suite of functionalities, including the ability to forge or decode packets, transmit them onto the network, capture incoming packets, and correlate requests with their corresponding replies. Scapy's capabilities extend beyond these core functions, encompassing tasks such as network scanning, traceroute execution, network probing, unit testing, attack simulation, and network discovery.
6. **sudo virt-customize -a <cloud img file name>.img --install snort** – This command installs **snort**, the Network Intrusion Detection System (NIDS) software chosen for this project, onto the cloud image. The installation process deployed here is limited to the base package installation. Further configuration tasks will be undertaken after the instance has been booted using this image.
7. **sudo virt-customize -a <cloud img file name>.img --install vim** – Install latest version of vim text editor.
8. **sudo virt-customize -a <cloud img file name>.img --install python3-pip** – Install latest version of pip. Pip is required to install other python libraries in our image.
9. **sudo virt-customize -a <cloud img file name>.img --install python3-blessed** – Blessed is a lightweight library for python to customize the terminal. It is used here to create plots in terminal directly.

These are all the packages we need for this project to work. Now instances can be created with this modified cloud image.

Uploading cloud image:

1. From the panel in the left navigate to Project -> Compute -> Images
2. Select “Create Image”
3. In the pop-up window, add image name here ubuntu_custom, browse and select the image file.
4. Select “QCOW2 – QEMU Emulator” from the Format dropdown list.
5. Finally click “Create image” and the image will start uploading. Once uploaded it will appear on the list in Images.

The screenshot shows the 'Create Image' dialog box. The 'Image Details' tab is selected. The 'Image Name' field contains 'snort_image'. The 'Image Description' field is empty. The 'Image Source' is set to 'File' with a 'Choose file' button and the filename 'snort_blessed.img'. The 'Format' dropdown is set to 'QCOW2 - QEMU Emulator'. The 'Image Requirements' section includes 'Kernel' (Choose an image), 'Ramdisk' (Choose an image), and 'Architecture' (empty). The 'Image Sharing' section includes 'Visibility' (Private, Shared, Community, Public) and 'Protected' (Yes, No). At the bottom are buttons for 'Cancel', 'Back', 'Next', and 'Create Image'.

Figure 8: Image uploading.

Creating instance:

Following the creation of the custom image, this section details the process of launching virtual machine instances utilizing the newly created image.

1. Access the project dashboard and navigate to the **Project** section.
2. Within the **Project** section, locate and select the **Compute** menu option.
3. From the **Compute** menu, select the **Instances** sub-menu.
4. Click the button labelled "Launch Instance" to initiate the instance creation process.

5. A subsequent series of dialog boxes will be presented to configure the instance settings:

- **Instance Name:** Specify a unique and descriptive name for the instance (e.g., Client_1, Client_2, Server).
- **Image:** Select the newly created image from the available image list.
- **Flavor:** Choose a flavor that aligns with the desired resource allocation, including the number of virtual CPUs (vCPUs), RAM capacity, and disk space allocation (e.g., m1.tiny: 1 vCPU, 192MB RAM, 1GB disk space).
- **Network:** Assign the appropriate network for the instance's network connectivity. Here, it is recommended to utilize separate networks (Net0, Net1, Net2) for each instance.

Project-Specific Instance Details

As per the project requirements, three separate instances will be created with the following configurations:

- **Instance Name:**
 - Client_1
 - Client_2
 - Server
- **Flavor:** m1.tiny (1 vCPU, 192MB RAM, 1GB disk space) (applied to all three instances)
- **Network:**
 - Client_1: Net0 (10.10.0.0/24)
 - Client_2: Net1 (10.10.1.0/24)
 - Server: Net2 (10.10.2.0/24)

Note: While the specific network assignments (Net0, Net1, Net2) are used in this example, alternative network configurations can be employed. However, it is crucial to ensure that each of the three instances is assigned to a distinct network.

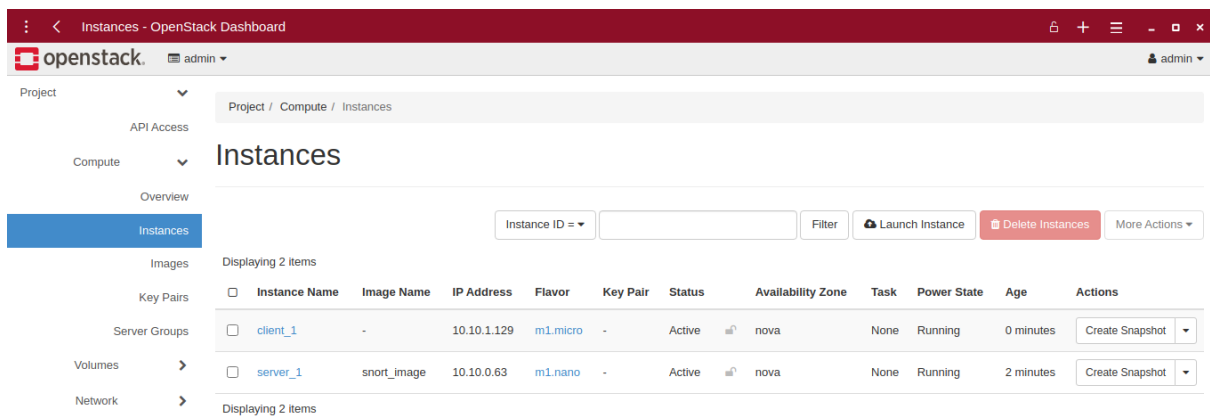


Figure 9: Instances created.

4.3 Implementation and deployment of VNF:

The deployment of Network Function Virtualization (NFV) [2] elements within this project leverages Tacker [25], an open-source platform specifically designed for NFV orchestration. Tacker plays a pivotal role in the management and orchestration of VNFs through the implementation of a Generic VNF Manager (VNFM) and an NFV Orchestrator. It is essential to establish a Virtual Infrastructure Manager (VIM) prior to the creation of VNFs. The VIM acts as a resource abstraction layer, providing a unified interface for managing the underlying infrastructure resources, such as compute, storage, and networking.

1. Open terminal and log in as stack user with **sudo su – stack**.
2. **cd tacke**r/tools - Change the directory to tool folder in tacke
3. **./gen_vim_config.sh** - gen_vim_config.sh generates the vim_config.yaml file in the same path and initializes the environment variables.
4. **cp vim_config.sh ~/devstack** - Copy the vim_config.sh file to the devstack directory.
5. **cd devstack** - Changes the directory to devstack
6. **openstack vim register --config-file ./vim_config.yaml --is-default --description 'vim for nfv_user in nfv' openstack-nfv-vim** - This step executes the registration process for the default OpenStack VIM within the Tacker environment. Following successful VIM registration, its presence will be reflected within the Tacker dashboard. The deployment of VNFs needs the initial a VNF descriptor (VNFD) file. This file serves as a blueprint, encapsulating the detailed configuration information required to deploy and manage the VNF instance.

Field	Value
auth_cred	{ "username": "admin", "user_domain_name": "default", "cert_verify": "True", "project_id": null, "project_name": "admin", "project_domain_name": "default", "auth_url": "http://127.0.0.1/identity/v3", "key_type": "barbican_key", "secret_uuid": "****", "password": "****" }
auth_url	http://127.0.0.1/identity/v3
created_at	2024-05-26 12:12:12.996733
description	vim for nfvd_user in nfvd
extra	
id	3ef9887f-a58c-4b3d-a045-5aea561e559c
is_default	True
name	openstack-nfv-vim
placement_attr	{ "regions": ["RegionOne"] }
project_id	0c52f2a0dbba46b0b66707543819f56b
status	ACTIVE
type	openstack
updated_at	None
vim_project	{ "name": "admin", "project_domain_name": "default" }

Figure 10: Register VIM output.

7. **cd tackler/samples/tosca-templates/vnfd** - Change the directory to the folder containing vnfd files.
8. **cp tosca_vnfd_userdata.yaml ~/devstack** - We have taken tosca_vnfd_userdata.yaml file and copied it to devstack folder.
9. **nano tosca-vnfd-userdata.yaml** - The existing VNFD file (tosca-vnfd-userdata.yaml) can be directly opened and its contents edited to align with the specific requirements of the VNF deployment. Alternatively, a pre-configured VNFD file can be downloaded from an external source to replace the default file. The VNFD file adheres to a specific YAML syntax and encapsulates the configuration parameters essential for deploying and managing the VNF instance for example VNF image name, system specifications, network links etc.

10. **tacker vnfd-create --vnfd-file tosca-vnfd-userdata.yaml ud** - TOSCA VNFD templates can be onboarded to Tacker VNFD Catalog using this command. (tacker vnfd-create --vnfd-file <yaml file path> <VNFD-NAME>).

```

+-----+-----+
| Field | Value |
+-----+-----+
| created_at | 2024-05-26 12:13:10.137305 |
| description | Packet-counting |
| id | 510c5b1f-f8eb-44a0-a530-23591c7d1e41 |
| name | ud |
| service_types | vnfd |
| template_source | onboarded |
| tenant_id | 0c52f2a0dbba46b0b66707543819f56b |
| updated_at | |
+-----+-----+

```

Figure 11: Create VNFD output.

11. **tacker vnf-create --vnfd-name ud udvnf** - This command creates the VNF.

```

+-----+-----+
| Field | Value |
+-----+-----+
| created_at | 2024-05-26 12:13:30.287362 |
| description | Packet-counting |
| error_reason | |
| id | d1e348b9-ef76-423d-b14f-eb8a5414fc3a |
| instance_id | 978b625b-0db0-4794-8818-cd9feaf88153 |
| mgmt_ip_address | |
| name | udvnf |
| placement_attr | {"vim_name": "openstack-nfv-vim"} |
| status | PENDING_CREATE |
| tenant_id | 0c52f2a0dbba46b0b66707543819f56b |
| updated_at | |
| vim_id | 3ef9887f-a58c-4b3d-a045-5aea561e559c |
| vnfd_id | 510c5b1f-f8eb-44a0-a530-23591c7d1e41 |
+-----+-----+

```

Figure 12: Create VNF output.

Now the VNF will be listed in the instances tab. It will also be listed under the VNF -> VNF Management.

The screenshot shows the OpenStack VIM Management interface. The left sidebar has a menu with 'VIM Management' selected. The main content area displays 'VIM Management' with a table of VIM instances. A single instance is listed with the name 'openstack-nfv-vim' and status 'ACTIVE'.

Name	Description	VIM Id	Default	Auth URL	Regions	User	Project	Status	VIM Type
openstack-nfv-vim	vim for nfv_user in nfv	3ef9887f-a58c-4b3d-a045-5aea561e559c	True	http://127.0.0.1/identity/v3	RegionOne	admin	admin	ACTIVE	openstack

Figure 13: Newly created VIM appearing under VIM management.

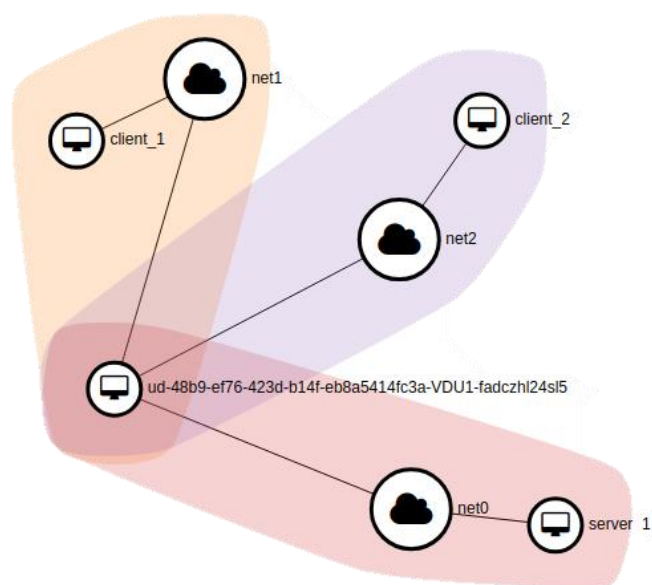


Figure 14: Final network topology.

The contents of **tosca-vnfd-userdata.yaml** used here is given below:

```
tosca_definitions_version: tosca_simple_profile_for_nfv_1_0_0

description: Packet-counting

metadata:
  template_name: sample-vnfd-userdata

topology_template:
  node_templates:
    VDU1:
      type: tosca.nodes.nfv.VDU.Tacker
      capabilities:
        nfv_compute:
          properties:
            num_cpus: 1
            mem_size: 512 MB
            disk_size: 16 GB
      properties:
        image: snort_image
        config: |
          param0: key1
          param1: key2
        mgmt_driver: noop
        user_data_format: RAW
        user_data: |
          #!/bin/sh
          sh /tmp/packet.sh

    CP1:
      type: tosca.nodes.nfv.CP.Tacker
      properties:
        management: true
        ip_address: 10.10.0.1
```

```
    order: 0
    anti_spoofing_protection: false
requirements:
  - virtualLink:
      node: VL1
  - virtualBinding:
      node: VDU1

VL1:
  type: tosca.nodes.nfv.VL
  properties:
    network_name: net0
    vendor: Tacker

CP2:
  type: tosca.nodes.nfv.CP.Tacker
  properties:
    management: true
    ip_address: 10.10.1.1
    order: 0
    anti_spoofing_protection: false
requirements:
  - virtualLink:
      node: VL2
  - virtualBinding:
      node: VDU1

VL2:
  type: tosca.nodes.nfv.VL
  properties:
    network_name: net1
    vendor: Tacker

CP3:
  type: tosca.nodes.nfv.CP.Tacker
  properties:
    management: true
    ip_address: 10.10.2.1
    order: 0
    anti_spoofing_protection: false
requirements:
  - virtualLink:
      node: VL3
  - virtualBinding:
      node: VDU1

VL3:
  type: tosca.nodes.nfv.VL
  properties:
    network_name: net2
    vendor: Tacker
```

4.4 Test VNF deployment:

Following successful deployment and confirmation that both the VIM instance and all target instances are operational; a quick testing can be performed to verify network connectivity between instances residing on distinct networks. After logging in to the VIM instance the following steps are performed.

1. The first step is activating the network interfaces associated with the VIM instance. This needs to be done for all the three networks where VNF is connected to. This can be done by following commands - **ifconfig ens3 up**, **ifconfig ens4 up** and **ifconfig ens5 up**
2. Following network interface activation, it is required to configure each interface with a unique IP address and subnet mask. This enables the VIM instance to communicate with the target instances on their respective networks –
ifconfig ens3 10.10.0.0/24 netmask 255.255.255.0
Repeat this for two other networks.
ifconfig ens4 10.10.2.0/24 netmask 255.255.255.0 and
ifconfig ens5 10.10.3.0/24 netmask 255.255.255.0
3. Log in to each target instance and attempt to ping the IP addresses of other instances on different networks. If the configuration steps were successful, the instances should now be able to ping each other.

5. Configuration of Snort in VNF

This project incorporates Snort [7], an open-source, and light-weight Network Intrusion Detection System (NIDS) software, into the deployed VIM instance as a VNF.

The VNF has a strategic role as an intermediary connection point for all interconnected networks and instances. Consequently, any network packets traversing between client machines and the server, in both directions, will be routed through the VNF. This strategic placement makes the VNF the optimal location to deploy the NIDS software. By positioning Snort within the VNF, it is empowered to detect intrusion attempts and potential vulnerabilities originating from external client networks attempting to access the internal server network.

5.1 Snort configuration file (snort.conf):

Snort contains a configuration file in the installation folder. In our case or for default installation it will be in `"/etc/snort"` folder. The Snort configuration file, typically named `snort.conf`, is the heart of a Snort deployment. It defines various parameters that govern how Snort operates and analyzes network traffic. There are several components to this file.

Global Settings: This section defines overall Snort behaviour, including network interfaces to monitor, logging options, and performance settings.

Preprocessor Rules: Preprocessors prepare captured packets for analysis by Snort. This section defines the preprocessor rules to be loaded.

Detection Engine Rules: This is the core section where you define the actual intrusion detection rules that Snort will use to identify suspicious activity.

Output Plugins: This section configures how Snort will report detected events. You can specify destinations like log files, network alerts, or security information and event management (SIEM) systems.

Within the context of Snort's functionality, a critical component lies in its Detection Engine Rules. These rules define Snort's behaviour by specifying the signatures or patterns it employs to identify intrusion attempts. The default Snort configuration file, `"snort.conf"`, contains a vast collection of these predefined detection rules. It is important to acknowledge that, for smaller systems, a significant portion of these default rules may be excessive and can be disabled to optimize performance.

Beyond the predefined rules, Snort offers users the flexibility to create and incorporate custom intrusion rules or signatures tailored to their specific requirements. These custom rules follow to a well-defined syntax and structure. The process of adding new rules involves modifying the `"local.rules"` file, typically located in the `"/etc/snort/rules/"` directory. It's noteworthy that the `"snort.conf"` file already includes a reference to this `"local.rules"` file within its rules section. Consequently, any custom rules introduced into the `"local.rules"` file become active immediately.

5.2 Snort rule structure:

Snort's detection engine relies on a well-defined rule syntax that offers flexibility while ensuring consistent behaviour. These rules allow for a spectrum of actions, including logging detected events, generating alerts, or actively blocking malicious traffic. Additionally, Snort supports the identification of various network protocols and facilitates pattern matching within network traffic to pinpoint potential intrusions.

The structure of a typical Snort rule looks like this:

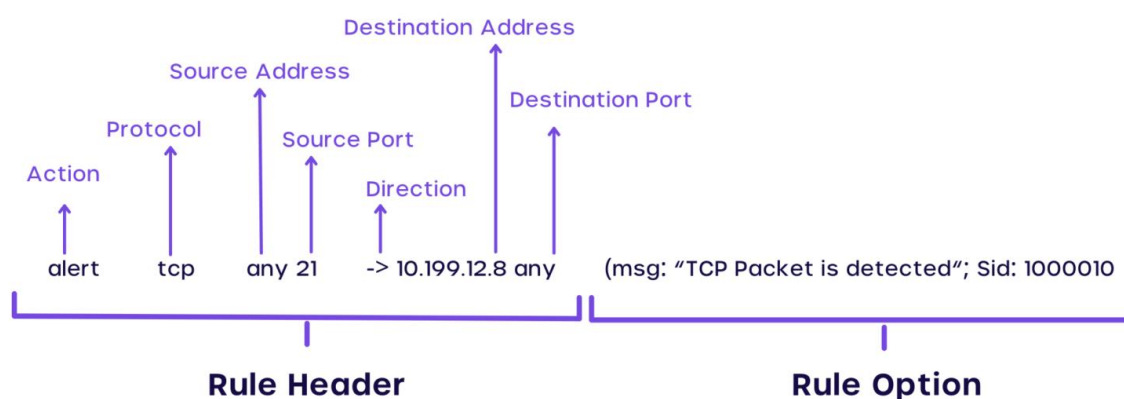


Figure 15: Snort rule structure. [36]

It has two major portions:

Rule Header – The Rule Header defines the fundamental parameters for evaluating network traffic against the specified criteria. It encompasses the following elements:

1. **Rule Action:** This element specifies the action Snort should take upon encountering a packet that matches the rule criteria. The default options include:
 - a. **Alert:** Generates an alert notification for identified suspicious activity.
 - b. **Block:** Actively blocks the malicious traffic, preventing it from reaching its intended destination.
 - c. **Drop:** Silently discards the malicious traffic without generating an alert.
 - d. **Pass:** Allows the legitimate traffic to proceed through the system.
 - e. **Log:** Records information about the detected event in a log file for further analysis.
2. **Protocol:** This element allows for the specification of a particular network protocol (e.g., TCP, UDP, ICMP) to focus the rule on a specific type of network traffic.
3. **Source IP Address:** This element defines the originating IP address or network address (using CIDR notation) of the traffic being inspected by the rule.
4. **Source Port:** specifies the source port number associated with the incoming traffic.

5. **Flow:** In the context of Snort rules, "Flow" refers to the direction of the network traffic. This element indicates the origin of the traffic (e.g., \$HOME_NET indicates traffic originating from the internal network).
6. **Destination IP Address:** Like the Source IP Address, this element defines the target IP address or network address of the traffic being inspected by the rule.
7. **Destination Port:** This element specifies the destination port number associated with the outgoing traffic.

Rule option - The Rule Options section provides further details on how Snort evaluates network traffic against the rule criteria. Some of the commonly used Rule Options include:

1. **Message:** This element defines a custom message given by the user that will be included in the alert notification or log entry when the rule is triggered.
2. **Content:** This element facilitates the inspection of the packet payload against a specified pattern or text string to identify potential malicious content.
3. **Flow:** Beyond the basic flow direction specified in the Rule Header, this option allows for more granular flow control by defining the connection type (e.g., established, stateless).
4. **sid:** This element assigns a unique identifier (SID) to each rule, enabling clear identification and management of individual rules within the Snort configuration.
5. **rev:** This element denotes the revision number of the rule, which is incremented when the rule definition is modified.

5.3 Setting up Snort:

Leveraging the established understanding of Snort and its rule writing principles, we can now proceed with the configuration of Snort within the VNF instance. The following steps outline the configuration process to be executed on the VNF machine.

1. **Snort Version Verification:** Snort has already been installed in our cloud image during instance creating. Check snort installation by command **snort -V**.

```

root@ud-83f6-f44d-4287-a92f-57fa1e31d928-vdu1-j3kvk6xdvpu3:~# snort -V

  o''-~
  ' ''~
    ~

-*> Snort! <*-
Version 2.9.7.0 GRE (Build 149)
By Martin Roesch & The Snort Team: http://www.snort.org/contact#team
Copyright (C) 2014 Cisco and/or its affiliates. All rights reserved.
Copyright (C) 1998-2013 Sourcefire, Inc., et al.
Using libpcap version 1.8.1
Using PCRE version: 8.39 2016-06-14
Using ZLIB version: 1.2.11

```

Figure 16: Snort Version.

2. Disabling Unnecessary Rules in snort.conf:

- The default Snort configuration file, "snort.conf," contains a vast collection of detection rules. For optimized performance, it is recommended to disable

unnecessary rules that are not applicable to your specific network environment.

- Use the vim text editor to open the "snort.conf" file with elevated privileges using: **sudo vim /etc/snort/snort.conf**
 - Within the "snort.conf" file, locate the rule definitions (typically starting around line 578 and extending to line 696).
 - To disable a rule line, insert a comment hash (#) at the beginning of the line. You can efficiently comment out multiple lines using the following command in vim mode - **:578,696s/^/#**
 - This command instructs vim to substitute (s) all lines (^) from line 578 to line 696 (inclusive) to begin with a comment hash (#).
3. Ensure that the line referencing the "local.rules" file remains uncommented within the "snort.conf" file. This inclusion is essential for adding your custom rules. Save and close the file.
4. **Custom Rule Creation in local.rules:**
- Now, open the "local.rules" file using vim with: **sudo vim /etc/snort/rules/local.rules**
 - This file serves as the designated location for adding your custom intrusion detection rules. Here, you can craft rules tailored to address specific threats or vulnerabilities within your network environment. Refer to the Snort rule writing guide for detailed instructions on the proper syntax and structure for creating effective Snort rules.
 - We write the custom rules in the file and then save the file.

```

1 # $Id: local.rules,v 1.11 2004/07/23 20:15:44 bmc Exp $
2 # -----
3 # LOCAL RULES
4 # -----
5 # This file intentionally does not come with signatures. Put your local
6 # additions here.
7
8 alert icmp any any -> $HOME_NET any (msg:"ICMP ping"; sid:1111;)
9
10 alert tcp any any -> $HOME_NET 8080 (msg:"Target 8080"; sid:2222;)
11
12 alert tcp any any -> $HOME_NET 5050 (msg:"Target 5050"; sid:3333;)
13
14 alert tcp any any -> $HOME_NET 80 (msg:"SYN flooding"; flags:S; flow:stateless;
15  detection_filter: track by_dst, count 10, seconds 10; sid:4444;)
16
17 alert tcp any any -> $HOME_NET 22 (msg:"SSH attempt"; flags:S; sid:5555;)
18
19 alert tcp 10.10.1.0/24 any -> $HOME_NET any (msg:"Unwanted network"; sid:6666;)
20
~

```

Figure 17: Local rules added.

5. Snort Rule Configuration Verification:

- As a final step, verify the Snort rule configuration by executing the following command on the VNF machine terminal:

sudo snort -T -i ens3 -c /etc/snort/snort.conf

- This command performs a test run of Snort using the specified configuration file ("/etc/snort/snort.conf") and network interface ("ens3"). The output will provide a summary of the Snort configuration, including the total number of loaded rules. This verification helps to identify any syntax errors or configuration issues within your custom rules.

5.4 Snort Operation and Monitoring:

Following successful Snort configuration, you can initiate its execution to commence network traffic monitoring. Snort continuously inspects network traffic traversing the designated interface, comparing packet signatures against its rule database. Upon detecting suspicious activity that matches a rule, Snort generates alerts.

Snort offers various operational modes, each tailored for a specific purpose. This project will primarily leverage two of these modes:

1. Alert mode:

- The alert mode prioritizes real-time presentation of alerts. As suspicious activity occurs on the network, Snort generates corresponding alerts and displays them directly in the console. This mode proves invaluable during initial Snort setup, for testing purposes, or for close monitoring of network activity.
- To execute Snort in alert mode, you can employ one of the following commands:
sudo snort -c /etc/snort/snort.conf -A console
or
sudo snort -q -l /var/log/snort -i <interface_name> -A console -c /etc/snort/snort.conf
- While alert mode delivers immediate results for close monitoring, it may become overwhelming for extended periods due to the continuous stream of alerts. Additionally, alerts generated in this mode are not automatically saved for later analysis.

2. Packet Logger mode:

- The alert mode, while useful for initial testing, is not optimal for long-term Snort operation. Furthermore, for effective post-detection analysis, it is essential to preserve generated alerts. This is where packet logger mode comes into play.
- In packet logger mode, Snort meticulously records all generated alerts within a user-specified log file or the default log directory. This approach offers the advantage of leveraging the captured logs to gather further network security

insights. Security information and event management (SIEM) systems or other software tools can then process these logs for comprehensive analysis, while Snort remains focused on continuous network traffic inspection.

- To initiate Snort operation in packet logger mode, utilize the following command:

```
sudo snort -q -l /var/log/snort -i <interface_name> -A fast -c /etc/snort/snort.conf
```

- In this command, the -A console option is replaced with -A fast, signifying the fast-logging mode. This mode prioritizes efficient log file creation to facilitate subsequent analysis.

```
05/27-06:33:40.785329  [**] [1:1111:0] ICMP ping [**] [Priority: 0] {ICMP} 10.10
.0.189 -> 10.10.1.22
05/27-06:33:40.791930  [**] [1:1111:0] ICMP ping [**] [Priority: 0] {ICMP} 10.10
.1.22 -> 10.10.0.189
05/27-06:33:41.783552  [**] [1:1111:0] ICMP ping [**] [Priority: 0] {ICMP} 10.10
.0.189 -> 10.10.1.22
05/27-06:33:41.787073  [**] [1:1111:0] ICMP ping [**] [Priority: 0] {ICMP} 10.10
.1.22 -> 10.10.0.189
05/27-06:33:42.786561  [**] [1:1111:0] ICMP ping [**] [Priority: 0] {ICMP} 10.10
```

Figure 18: Snort generated alert 1.

```
05/27-06:35:08.855990  [**] [1:2222:0] Target 8080 [**] [Priority: 0] {TCP} 10.1
0.0.189:20 -> 10.10.1.22:8080
05/27-06:35:09.446196  [**] [1:3333:0] Target 5050 [**] [Priority: 0] {TCP} 10.1
0.0.189:20 -> 10.10.1.22:5050
05/27-06:35:10.404540  [**] [1:2222:0] Target 8080 [**] [Priority: 0] {TCP} 10.1
0.0.189:20 -> 10.10.1.22:8080
05/27-06:35:10.988211  [**] [1:3333:0] Target 5050 [**] [Priority: 0] {TCP} 10.1
0.0.189:20 -> 10.10.1.22:5050
05/27-06:35:12.000304  [**] [1:2222:0] Target 8080 [**] [Priority: 0] {TCP} 10.1
```

Figure 19: Snort generated alert 2.

```
0.0.189:20 -> 10.10.1.22:5050
05/27-06:35:17.392121  [**] [1:2222:0] Target 8080 [**] [Priority: 0] {TCP} 10.1
0.0.189:20 -> 10.10.1.22:8080
05/27-06:35:17.878723  [**] [1:3333:0] Target 5050 [**] [Priority: 0] {TCP} 10.1
0.0.189:20 -> 10.10.1.22:5050
05/27-06:36:32.306074  [**] [1:4444:0] SYN flooding [**] [Priority: 0] {TCP} 10.
10.0.189:20 -> 10.10.1.22:80
05/27-06:36:32.908411  [**] [1:4444:0] SYN flooding [**] [Priority: 0] {TCP} 10.
10.0.189:20 -> 10.10.1.22:80
05/27-06:36:33.556619  [**] [1:4444:0] SYN flooding [**] [Priority: 0] {TCP} 10.
10.0.189:20 -> 10.10.1.22:80
05/27-06:36:34.084753  [**] [1:4444:0] SYN flooding [**] [Priority: 0] {TCP} 10.
10.0.189:20 -> 10.10.1.22:80
```

Figure 20: Snort generated alert 3.

6. Implementation of custom NIDS

While Snort offers a powerful and versatile open-source Network Intrusion Detection System (NIDS), its resource requirements can pose challenges for smaller cloud environments and virtual machines. Snort's continuous real-time network traffic sniffing creates a significant system load, particularly in memory consumption. This memory usage is crucial for comparing network packets against signature databases while maintaining network flow. If insufficient memory is available, Snort can crash, halting intrusion detection. Since NIDS software is designed for 24/7 operation, memory demands can become a limiting factor, especially considering other software running on the system. Consequently, Snort's effectiveness diminishes on low-specification systems.

This project explores an alternative approach to intrusion detection by implementing a smaller, more lightweight prototype NIDS alongside Snort for comparison. This custom NIDS offers several advantages:

- **Reduced Resource Consumption:** The lightweight design minimizes resource demands, making it suitable for environments with limited memory or processing power, where Snort might struggle.
- **User-Defined Signatures:** Unlike Snort's pre-defined rules, this NIDS grants complete control over the detection signatures. Users can tailor the system to their specific needs by adding or removing rules through code modifications.
- **Granular Detection Control:** The ability to modify code allows for highly customized detection methods. Users can define precisely how each rule or detection method should function within the NIDS.

This level of fine-grained control comes with the trade-off of requiring code editing expertise. However, for users seeking a highly customized intrusion detection solution optimized for their specific environment, this custom NIDS offers a compelling alternative to Snort.

Implementations:

The custom NIDS leverages Python and Scapy, a powerful Python library specifically designed for network packet manipulation. Scapy offers a versatile suite of functionalities. Scapy empowers users to forge, decode, send, capture, and match network packets. This granular control over network traffic is essential for crafting effective intrusion detection mechanisms. It can also handle tasks like scanning, tracerouting, probing, unit tests, attacks, and network discovery. These features empower developers to construct highly customized network tools and security applications, such as the custom NIDS implemented in this project.

6.1 Custom NIDS Packet Processing Workflow:

The custom NIDS leverages a background network sniffer that continuously captures incoming and outgoing packets on the designated network interface.

- **Packet Capture and Analysis:** Each captured packet undergoes a detailed examination through user-defined functions or methods. These functions meticulously record all packet details for further processing.
- **User-Defined Rules and Matching:** The extracted packet information is then compared against user-written rules or code. If a match is found, the corresponding function or method triggers an alert.
- **Multiple Matches and Alert Generation:** A single packet can potentially match multiple rules, resulting in the generation of several alerts for the same packet. This can be beneficial in certain situations, offering a more comprehensive view of potential threats. However, it's crucial to consider potential alert fatigue if an excessive number of rules are implemented.
- **Logging the alerts:** To facilitate further analysis and investigation, alerts generated by the methods are thoroughly recorded within a designated log file.

The current implementation can utilize Python's multithreading capabilities to enhance performance.

- **Parallel Processing:** While the main thread handles network sniffing, a separate thread concurrently performs packet filtering and data comparison. This parallel execution can significantly improve processing speed, especially when dealing with a large number of detection rules or methods.
- **Scalability with Multiple Threads:** If a vast number of detection rules are employed, the system can be further optimized by running each method in a separate thread. This approach maximizes parallelism and processing efficiency.

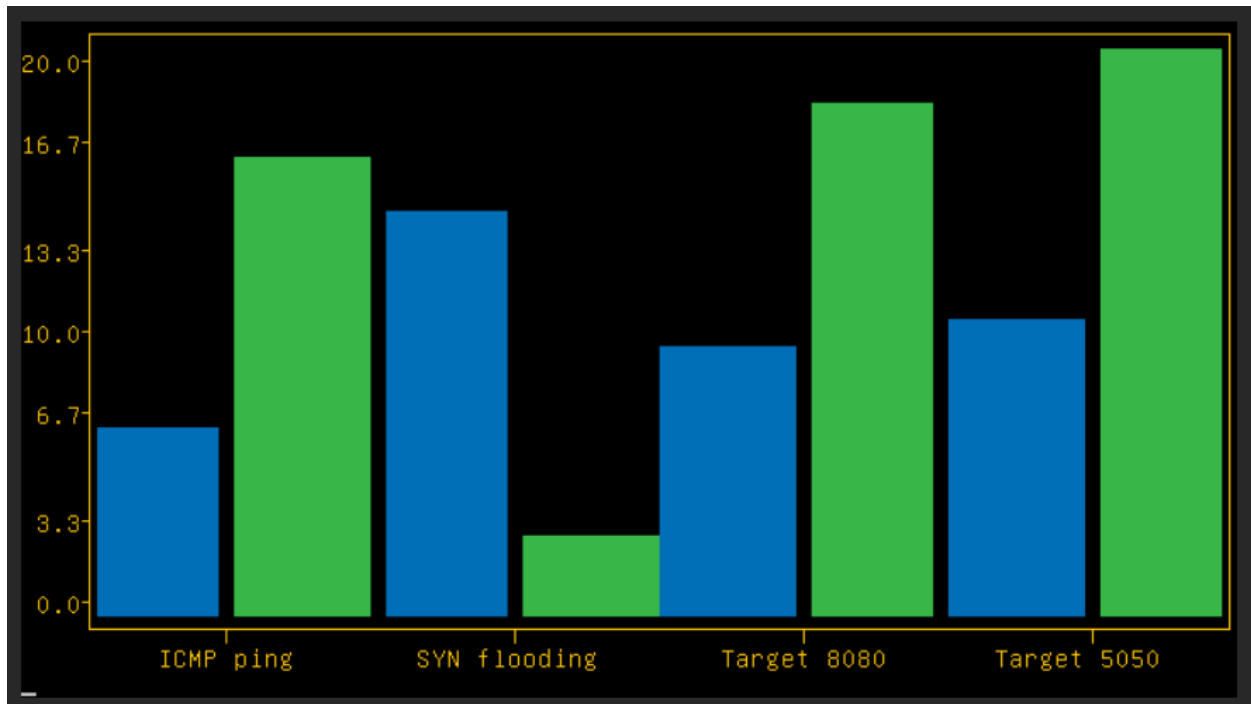


Figure 22: Comparison 2. (Blue-Short, Green-Our implementation).



Figure 23: Comparison 3.

7. Conclusion

This project investigated methods for enhancing the security of OpenStack cloud applications. The project discussed the integration of an Intrusion Detection System (IDS) within the NFV framework, outlining the necessary architecture and potential challenges. It then focused on Snort, a popular open-source NIDS, demonstrating its successful integration as a VNF within OpenStack. The project highlighted Snort's advantages, such as its extensive rule set and scalability, and detailed the deployment approach for this integration.

Furthermore, the project introduced a custom, user-specific NIDS designed for low-resource systems where Snort might be too demanding. This custom NIDS underwent development, testing, and performance evaluation through graphical visualizations to compare its effectiveness against Snort. The results of this comparison, along with a discussion of the trade-offs between Snort and the custom NIDS have been thoroughly documented.

By presenting these two viable approaches to securing OpenStack cloud environments, this project empowers organizations to make informed decisions regarding their cloud security posture. Integrating an IDS within the NFV framework offers a robust security solution, while exploring alternative, resource-efficient NIDS options like the custom solution allows organizations to adapt their security measures to their specific resource constraints.

References

- [1] *What is OpenStack* - <https://www.openstack.org/> , <https://wiki.openstack.org/wiki> , <https://docs.openstack.org/>
- [2] *VNF/NFV* - "ETSI - Standards for NFV - Network Functions Virtualization | NFV Solutions", ETSI GS NFV 002 V1.1.1 (2013-10).
- [3] *OpenStack Nova* - <https://docs.openstack.org/nova/latest/> , <https://docs.openstack.org/nova/latest/admin/architecture.html>
- [4] *OpenStack Neutron* - <https://docs.openstack.org/neutron/latest/> , <https://docs.openstack.org/neutron/latest/install/concepts.html>
- [5] *OpenStack Cinder* - <https://wiki.openstack.org/wiki/Cinder> , <https://docs.openstack.org/cinder/latest/>
- [6] *What is an Intrusion Detection System (IDS)?* - Check Point Software Technologies. 2023. Retrieved 27 December 2023.
- [7] *Snort* - <https://www.snort.org/>
- [8] *Verizon 2022 data-breach investigation report* - 2022-data-breach-investigations-report-dbir.pdf, <https://www.verizon.com/business/en-gb/resources/2022-data-breach-investigations-report-dbir.pdf>
- [9] *Suricata* - <https://suricata.io/>
- [10] *OpenStack architecture and its modular approach* - <https://docs.openstack.org/install-guide/get-started-logical-architecture.html>
- [11] *OpenStack history* - <https://en.wikipedia.org/wiki/OpenStack>
- [12] *OpenStack survey* - <https://www.openstack.org/user-survey/2022-user-survey-report>
- [13] *IDS conceptual architecture* - <https://docs.openstack.org/install-guide/get-started-conceptual-architecture.html#get-started-conceptual-architecture>
- [15] *Libvirt virtualization api* - <https://libvirt.org/>
- [16] *Libvirt Driver management daemon* - <https://libvirt.org/manpages/libvirtd.html>
- [17] *KVM* - https://linux-kvm.org/page/Main_Page, <https://www.redhat.com/en/topics/virtualization/what-is-KVM>
- [18] *OpenStack Keystone* - <https://docs.openstack.org/keystone/latest/>
- [19] *OpenStack Glance* - <https://docs.openstack.org/glance/latest/>, <https://docs.openstack.org/glance/latest/install/get-started.html>
- [20] *Glance supported image formats* - <https://www.oreilly.com/library/view/deploying-openstack/9781449311223/ch03s02.html>.
- [21] *OpenStack Swift* - <https://docs.openstack.org/swift/pike/admin/objectstorage-intro.html>

- [22] *OpenStack Heat* - <https://wiki.openstack.org/wiki/Heat> ,
<https://docs.hpc.cam.ac.uk/cloud/userguide/07-orchestration.html>
- [23] *Redhat platform 9 OpenStack architecture guide* -
https://access.redhat.com/documentation/es-es/red_hat_openstack_platform/9/html-single/architecture_guide/index
- [24] *OpenStack Horizon* - <https://docs.openstack.org/horizon/latest/> ,
https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/13/html-single/introduction_to_the_openstack_dashboard/index
- [25] *OpenStack Tacker* - <https://docs.openstack.org/tacker/latest/user/introduction.html>
- [26] *NFV redhat docs* - https://access.redhat.com/documentation/en-us/red_hat_openstack_platform/13/html/network_functions_virtualization_product_guide/ch-nfv_software
- [27] *Openstack and ETSI nfv architecture* - <https://www.oreilly.com/library/view/openstack-for-architects/9781788624510/146ebf1c-d2b1-48a7-b901-6219edebbce4.xhtml>
- [28] *OpenStack map*: <https://www.openstack.org/software/>
- [29] *What is intrusion* - Tiwari, Mohit & Kumar, Raj & Bharti, Akash & Kishan, Jai. (2017). INTRUSION DETECTION SYSTEM. International Journal of Technical Research and Applications. 5. 2320-8163.
- [30] (According to the National Institute of Standard and Technology) - Kumar, Satish & Gupta, Sunanda & Arora, Sakshi. (2021). Research Trends in Network-Based Intrusion Detection Systems: A Review. IEEE Access. PP. 1-1. 10.1109/ACCESS.2021.3129775.
- [31] *OSSEC* - <https://www.ossec.net/>
- [32] *Sguil* - <https://bammv.github.io/sguil/index.html>
- [33] *Security Onion* - <https://securityonionsolutions.com/>
- [34] *General IDS structure* - Aleksandar Lazarevic, Vipin Kumar, Jaideep Srivastava, 2008 'Intrusion Detection: A Survey', Computer Science Department, University of Minnesota
- [35] *Devstack* - <https://docs.openstack.org/devstack/latest/index.html>
- [36] *Snort rule structure* - <https://cyvatar.ai/write-configure-snort-rules/>
- [37] *NFV vs traditional network*: <https://www.blueplanet.com/resources/What-is-NFV-prx.html>