TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

DIPLOMA THESIS

Measuring Performance of 5G Cyberdefence Systems

Author:

Manos Lefakis

Committee:

Sotirios Ioannidis

Apostolos Dollas

Eftichios Koutroulis

2023

Abstract

This thesis focuses on the field of 5G network security, specifically in the area of programmable networks that use Kubernetes as a VNF controller. For these operating environments, there are currently no available solutions for efficient real-time monitoring and log aggregation that will enable network administrators to evaluate performace and security threats. While some general-purpose industry-standard solutions offer a number of features, such as leveled logs, advanced text formatting, built-in visualizations and frontends, their use introduces significant performance overheads and requires a lot of customization effort in order to be deployed in live networks.

In this thesis, a new metrics system, developed in Golang, is introduced to alleviate the shortcomings of the aforementioned solutions. It enables efficient real-time monitoring and evaluation of network services and its parser-analyzer captures information related to network service performance and security and analyzes it in order to provide valuable insights. Automation scripts and Ansible configurations were also developed to facilitate and automate the monitoring and evaluation process, offering a more efficient and effective means of measuring the performance of network services as well as the effectiveness of the security countermeasures deployed in them.

The developed metrics system has been deployed, tested and validated in the PRINCIPALS research project. PRINCIPALS strives to enhance active network service management and security through the provision of security primitives to counter diverse types of attacks. The metrics system is an integral component of the PRINCIPALS framework, providing real-time benchmarking capabilities, as well as serving as the main logging mechanism.

Contents

List of Figures

Listings

Chapter 1

Introduction

The field of network programming and security is rapidly evolving with the advent of 5G technology. What is required is an architecture and tools that enable network defenders to execute "mobile" defensive cyber operations (DCO) that can match the speed, scale, and precision of attacker tools without introducing new points of vulnerability or risking network instability.

1.1 Purpose of the Thesis

The purpose of this thesis is to address the critical issue of evaluating the performance of network services deployed within a programmable network that uses Kubernetes as a VNF controller. This involves providing a comprehensive and systematic collection of information and subsequent analysis of the network services and security primitives offered by such networks. By conducting this research, the goal is to gain a better understanding of the effectiveness and limitations of programmable networks in terms of network management and security. This will enable the community to identify areas where improvements can be made so as to contribute to the advancement of the field.

To achieve the stated objective, an efficient and effective metrics system has been developed to evaluate the performance of the services and primitives utilized within programmable networks. Experiments and simulations have been carried out using attack mitigation demo scenarios. These have been used to generate relevant data that can be utilized to improve the design and overall performance of programmable networks. The final system has been integrated and deployed within the PRINCIPALS cyber defence infrastructure. As such, it has been effectively used as a means to provide an in-depth evaluation of the PRINCIPALS framework's network's security infrastructure.

1.2 Thesis Contribution

In order to meet the objective of evaluating the performance of network services within a programmable network that uses Kubernetes as a VNF controller, a metrics system was developed on top of a custom logging system. The logging system was purpose-built to provide comprehensive data collection and analysis capabilities required by the metrics system.

This system is capable of monitoring the network services in real-time and provides metrics that can be used to assess the performance of the network services and security primitives.

As previously mentioned, the metrics system was developed as part of the PRINCIPALS project, which served as the testbed for this research. This thesis makes the following contributions to the PRINCIPALS project:

- A complete real-time metrics system, developed in Golang, consisting of the following components:

    1. A logging library that acts as an entry point to the metrics system. It provides a logging API similar to Golang's log library, while in the background is forwarding logs to a logging agent.
    2. A log-forwarder, an alternative entry point to the logging system. It is an executable, which can be deployed as a sidecar container[3] alongside any security application that logs to stdout. The log-forwarder is responsible for forwarding its standard input stream to a logging agent.
    3. A logging agent, packaged with Docker, which is deployed on each node of the Kubernetes cluster. The agent serves as an endpoint for log entries generated by the logging library or log-forwarder, and forwards them to a centralized metrics server. The logging agent serves as a mid-point in the log aggregation process, facilitating the analysis of system logs across the distributed network.
    4. The central metrics server, packaged with Docker, which is deployed on a single node of the cluster. The server provides centralized log aggregation, real-time sorting of unordered log entries, log parsing for real-time monitor and evaluation functionality using traditional parsing methods inspired by compiler-based log analysis techniques, and report-style updates on special events.

- Automation scripts for managing the framework's infrastructure components, which include:

    1. Managing the Docker registry
    2. Providing abstraction in Kubernetes deployment configuration
    3. Getting and filtering information about the k8s cluster, like a wrapper to k8s' CLI commands
    4. Reducing the repetitive work overhead of the setup of the infrastructure. During the process of this thesis, the infrastructure had to be continuously set up due to changes in the deployment structure or the need to test different scales of the cluster. This process was reduced to a one-click solution by utilizing and combining:
        - Vagrant
        - Ansible
        - Unix tools (e.g., Bash, SSH)

This thesis represents a substantial advancement in the field of network performance evaluation, as it offers a powerful tool that can precisely gauge and monitor the performance of network services that have been deployed within a programmable network in real-time.

Ultimately, this research will contribute to the advancement of network management and security by providing valuable data to inform future development and optimization efforts.

1.3 Thesis Overview

Chapter 2 provides a comprehensive overview of the current state of programmable networks and security. It covers the introduction of 5G technology, its impact on networking and security, and the concepts of software-defined networking and network function virtualization. Additionally, the chapter delves into the openflow protocol and the security threats in programmable networks (for example DDoS attacks) and the techniques used to mitigate them. The importance of logging systems in measuring performance is also discussed, highlighting the differences between k8s' native logging system and third-party logging systems for Kubernetes. The chapter concludes by offering an overview of common practices in metrics collection and evaluation.

Chapter 3 outlines the design and architecture of the PRINCIPALS framework. By introducing the PRINCIPALS framework, the main operating environment under which the metrics system will be deployed is defined and the requirements for this system are specified. As such, the main concept and overall architectural design of the proposed metrics system is presented thereafter.

The implementation of the proposed metrics system is detailed in Chapter 4. More specifically, the three main parts of the metrics system, namely the logging library (clog), the logging agent and the analysis server are presented along with the results of deploying the logging system in three attack mitigation scenarios.

Chapter 5 presents related work in the field of logging systems for programmable networks and security.

Chapter 6 validates the performance and reliability of the metrics system.

Chapter 7 concludes the thesis and provides future directions for the development of the metrics system.

Chapter 2

Programmable Networks and Security

In recent years, the explosive growth of mobile and internet traffic has resulted in an increased demand for high-speed and efficient network architectures. The fifth-generation (5G) wireless network technology has emerged as a promising solution to meet these demands. However, the traditional network architecture is not capable of handling the complex requirements of 5G networks. Therefore, a new paradigm of networking, Software-Defined Networking (SDN) and Network Function Virtualization (NFV), has been proposed to overcome these limitations [11]. This chapter aims to provide an in-depth understanding of SDN, NFV, and OpenFlow, with a focus on their applications in 5G networks, as well as the security threats associated with programmable networks.

2.1 5G Deployment Options and Network Architecture

Fifth Generation (5G) is a cutting-edge wireless communication technology that aims to revolutionize various industries, including healthcare, automotive, logistics, smart cities, energy, and manufacturing. It offers fast data transfer, low latency, and extensive device connectivity.

The 5G system, defined as a 3GPP system, comprises:

- The 5G Radio Access Network (RAN)
- The 5G Core Network (5GC)
- The User Equipment (UE)

The 5G deployment options are specified in 3GPP, utilizing either the existing Evolved Packet Core (EPC) or the 5GC network standard. The 5GC is a cloud-native architecture with virtualization at its core, providing:

- Superior network slicing
- Quality of Service (QoS)
- Control and data plane separation through Software-Defined Networking (SDN)

The 5GC enables the optimization of the network for specific use cases, enhancing the overall network performance.

The three primary use cases of 5G are

- **Enhanced Mobile BroadBand** (eMMB)

  eMMB offers high-speed data transfer with low latency to support enhanced mobile broadband services.

- **Massive Machine-Type Connectivity** (mMTC)

  mMTC supports the connectivity of vast numbers of low power consumption devices, such as Internet of Things (IoT) devices.

- **Ultra-Reliable and Low-Latency Connectivity** (URLLC)

  URLLC enables low-latency and highly reliable communication for mission-critical applications such as autonomous vehicles, remote surgery, and industrial automation.

5G can be introduced either in Standalone (SA) mode using 5GC or in non-SA mode utilizing a mix of EPC and 5GC, which offers multiple migration options. The SA mode using 5GC offers better network performance and enables operators to leverage the full potential of 5G. Non-SA mode enables the gradual transition from the existing network infrastructure to the 5G network.

In conclusion, 5G technology provides numerous opportunities for various industries and enables the deployment of new use cases that were previously impossible. The deployment options for 5G are being defined in 3GPP, offering operators several migration paths. As 5G evolves, it is expected to transform the way we interact with technology and boost productivity in various industries. [9]

2.1.1 5G Core

The 5G core network is the central component of the 5G network architecture that provides connectivity and services to the end-users. It is responsible for managing the entire network and ensuring the delivery of data and services between different network functions and devices. The 5G core network is designed to be flexible, scalable, and secure, and it supports a wide range of applications, services, and devices. [6]

The main functions of the 5G core network include:

1. Network slicing: The 5G core network enables the creation of virtual networks called network slices, which can be customized to meet the specific requirements of different applications and services.

2. Quality of service (QoS) management: The 5G core network ensures that different applications and services receive the necessary resources and bandwidth to meet their QoS requirements.

3. Security: The 5G core network provides advanced security features such as encryption, authentication, and authorization to ensure the confidentiality and integrity of the network and its users.

4. Multi-access edge computing (MEC): The 5G core network supports MEC, which allows applications and services to be deployed closer to the end-users, reducing latency and improving the overall user experience.

5. Virtualization: The 5G core network utilizes network function virtualization (NFV) and software-defined networking (SDN) technologies to increase flexibility, reduce costs, and enable rapid service deployment.

6. Mobility management: The 5G core network manages the mobility of devices as they move between different network areas, ensuring seamless connectivity and service delivery.

7. Service orchestration: The 5G core network enables the dynamic orchestration of services and network resources to meet changing service demands and optimize resource utilization.

Overall, the 5G core network plays a crucial role in delivering the benefits of 5G technology, including high-speed connectivity, low latency, and support for a wide range of applications and services.

2.1.2 Software Defined Networking - SDN

SDN is a networking paradigm that separates the control plane from the data plane in network devices. In traditional networks, switches and routers contain both the control plane and the data plane. In contrast, SDN separates these two planes, allowing network administrators to control the network using a centralized controller, which provides a global view of the network. This separation of the control plane and data plane enables greater flexibility, scalability, and agility in network management and allows for the automation of network configurations.

SDN architecture consists of three main components:

- the data plane
- the control plane
- the application plane

The data plane comprises the network devices, such as switches and routers, which forward packets based on their destination addresses. The control plane is responsible for network management, including routing protocols and network policies. The application plane contains the network applications that use the network resources, such as load balancers and firewalls.

2.1.3 Network Function Virtualization - NFV

Network Function Virtualization (NFV) is a networking paradigm that aims to simplify the deployment and management of network services by virtualizing network functions. NFV virtualizes network functions such as firewalls, intrusion detection systems, load balancers, and other functions, allowing them to run on any standard hardware, including servers, switches, and routers. This approach replaces the traditional model where each network function is implemented on dedicated, purpose-built hardware with a more flexible, software-based model. This enables network operators to reduce their capital and operational expenses by utilizing existing infrastructure, rather than investing in proprietary hardware.

NFV provides several benefits for network operators by enabling network functions to be deployed as Virtual Network Functions (VNFs) that can be dynamically allocated, scaled, and placed on demand according to the network's requirements. This flexibility allows for faster time-to-market, improved cost-effectiveness, and more manageable and orchestrated network services. Operators can quickly deploy new services, update existing ones, and adjust network infrastructure without significant hardware upgrades or changes, thanks to NFV's ability to instantiate and scale network functions on demand, which enables them to respond quickly to changing network demands. [8]

Overall, NFV is a powerful paradigm that offers many benefits to network operators. By virtualizing network functions and enabling dynamic deployment, NFV can improve the flexibility, scalability, and cost-effectiveness of network services, making it a critical component of modern networking architectures.

2.1.4 Virtual Networking Technologies for Cloud Environments: OpenFlow, Openvswitch, and AntreaOVS

- **OpenFlow** is a communication protocol that is used between the centralized controller and the data plane devices in an SDN architecture. The protocol enables the controller to program and configure the behavior of network devices, including switches and routers. OpenFlow enables network administrators to control the network using a centralized controller, providing a global view of the network, and enabling automation of network configurations.

  OpenFlow consists of two main components:

  - **The controller**: The controller is responsible for managing the network and defining network policies. It communicates with the switches in the network and is responsible for directing traffic flows based on the policies it has defined. The controller can be implemented on a separate device or as part of a software-defined networking (SDN) controller.
  - **The switch**: The switch is responsible for forwarding packets based on the policies defined by the controller. When a packet arrives at the switch, it is inspected to determine the appropriate action based on the policies defined by the controller. The switch then forwards the packet to the appropriate destination or to another switch in the network. OpenFlow switches are typically implemented as specialized hardware devices, but can also be implemented in software running on a server.

  OpenFlow allows for the dynamic configuration of network policies, enabling network administrators to respond quickly to changing network conditions. [7]

- **Openvswitch** is an open-source, multi-layer virtual switch that provides advanced networking features to virtual machines (VMs) and containers. It can be used in a variety of virtualization environments, including KVM, Xen, and VirtualBox. Openvswitch allows network administrators to create complex network topologies, including virtual LANs (VLANs), load balancing, and traffic shaping. Openvswitch supports OpenFlow, allowing network administrators to use OpenFlow to control the behavior of Openvswitch.

- **AntreaOVS** is an open-source networking and security solution for Kubernetes that is built on top of Openvswitch. It provides advanced networking features such as load balancing, network policy enforcement, and traffic routing for Kubernetes clusters. AntreaOVS uses Openvswitch to provide network virtualization and to connect Kubernetes pods to the physical network.

With the use of OpenFlow, Openswitch, and AntreaOVS, network administrators have access to advanced networking features that enable them to control and configure the behavior of network devices. These tools provide a global view of the network, enabling automation of network configurations and allowing for efficient and effective testing and emulation of network deployments. By abstracting the network infrastructure to a level where testbeds and emulations can replicate actual deployments, network administrators can create complex network topologies with advanced features. This results in a reliable and scalable solution for networking and security in virtualized environments such as Kubernetes clusters.

2.2 Security Threats

The programmability of SDN and NFV brings several security challenges that need to be addressed. The centralization of control in SDN networks and the dynamic nature of NFV deployments present new attack surfaces that can be exploited by malicious actors. The following are some of the security threats associated with programmable networks:

- **Denial-of-service (DoS) attacks:** SDN controllers are potential targets for DoS attacks, which can result in the disruption of the network. An attacker can flood the controller with a large number of requests, causing it to become overloaded and unable to process legitimate traffic.
- **Man-in-the-middle (MitM) attacks:** SDN networks are susceptible to MitM attacks, where an attacker intercepts and alters network traffic. The centralized control plane of SDN networks makes them vulnerable to MitM attacks, as a compromised controller can instruct switches to forward traffic to an attacker-controlled device.
- **Data breaches:** The virtualization of network functions in NFV can increase the risk of data breaches. A compromised VNF can result in the leakage of sensitive information.
- **Malware propagation:** Malware can propagate quickly in SDN networks due to the dynamic nature of network configurations. A compromised VNF can propagate malware to other VNFs, switches, and controllers in the network.
- **Unauthorized access:** The centralization of control in SDN networks can result in unauthorized access to the network. A compromised controller can provide an attacker with access to the entire network, including sensitive data.
- **Insider threats:** The programmability of SDN and NFV networks increases the risk of insider threats, where a trusted user abuses their privileges to compromise the network. Insiders with access to the controller or VNFs can modify network configurations, resulting in the disruption of the network or the leakage of sensitive information.

To mitigate these security threats, network administrators need to implement appropriate security measures, such as access control, encryption, and monitoring. Additionally, security should be integrated into the design of programmable networks from the outset, rather than being added as an afterthought. Programmable networks are vulnerable to security threats, such as Distributed Denial-of-Service (DDoS) attacks, which can bring down a network by overwhelming it with traffic. These attacks are increasingly sophisticated, and it is crucial to have measures in place to mitigate them.

2.2.1 DDoS Attacks

Old security threats, known from traditional networks, like DDoS attacks, are still present in programmable networks, but the attack surface has expanded with the introduction of new technologies and protocols. For example, software-defined networking (SDN) introduces new potential vulnerabilities in the control plane, and network function virtualization (NFV) introduces new attack vectors in the virtualized network functions. Therefore, securing programmable networks requires a comprehensive approach that includes both traditional security measures and new techniques that address the specific challenges of programmable networks.

Distributed Denial of Service (DDoS) attacks have become a major concern for organizations. In a DDoS attack, an attacker uses a network of compromised devices, such as computers or IoT devices, to flood a targeted network or server with traffic, rendering it unavailable to legitimate users. These attacks can cause significant damage to organizations, including loss of revenue, reputation, and customer trust.

There are different types of DDoS attacks, and understanding them is crucial or protecting networks and systems from service disruption. Three common types of DDoS attacks are HTTP flooding, UDP flooding, and SYN flooding, each of which exploits different vulnerabilities to overwhelm the target server and make it unavailable to legitimate users.

- **HTTP flooding**, also known as HTTP flood attack, is a type of DDoS attack where a large number of HTTP requests are sent to a server with the intention of overwhelming it and making it unavailable to legitimate users. This type of attack is particularly effective against web servers that process a high volume of HTTP requests, such as e-commerce sites or online gaming platforms.
- **UDP flooding**, also called UDP flood attack, is a type of DDoS attack that targets the user datagram protocol (UDP), a connectionless protocol that is commonly used for video streaming and online gaming. In a UDP flood attack, a large number of UDP packets are sent to the victim's server, overwhelming its ability to process incoming traffic and resulting in service disruption.
- **SYN flooding**, or SYN flood attack, is a type of DDoS attack that exploits vulnerabilities in the way TCP/IP connections are established. In this attack, the attacker sends a large number of SYN packets to the victim's server, but does not complete the connection by sending the final ACK packet. This causes the server to wait for the ACK packet, tying up its resources and making it unavailable to legitimate users. SYN flooding attacks can be difficult to detect and mitigate, as the attack traffic appears to be legitimate connection attempts.

2.2.2 DDoS Mitigation Techniques

DDoS attacks pose a serious threat to network security, and various techniques can be used to mitigate their effects.

Some of these techniques include:

- **Scrubbing services**, which filter out malicious traffic and allow legitimate traffic to pass through. These services can be deployed on-premises or in the cloud, and are often provided by third-party service providers.
- **Rate-limiting**, which limits the amount of traffic that can be sent to a particular IP address or port. While effective in mitigating low-level DDoS attacks, this technique may not be sufficient for large-scale attacks.
- **Software-defined networking (SDN)**, which enables network administrators to dynamically reconfigure network policies in response to attacks. SDN can facilitate the deployment of intelligent traffic analysis tools that can identify and block malicious traffic in real-time.

In securing programmable networks against DDoS attacks, it is crucial to implement a combination of traditional security measures and new techniques specific to programmable networks. Failure to do so can result in significant damage to an organization's revenue, reputation, and customer trust. Therefore, network administrators must remain vigilant in implementing these techniques to prevent DDoS attacks from causing harm to their organizations.

2.3 Common Practices on Metrics

2.3.1 Overview

Metrics play a crucial role in evaluating the performance and effectiveness of programmable networks. Software-defined networking (SDN) presents a challenge in obtaining metrics for effective network performance analysis. To address this challenge, several techniques and tools have been developed and continue to evolve. Some common practices for obtaining metrics in programmable networks (SDN) include:

- **Flow-based measurement**: This technique involves measuring and collecting statistics for individual flows in the network, including packet loss, throughput, and delay.
- **End-to-end measurement**: This technique involves measuring the performance of a network path between two endpoints, including metrics such as latency, jitter, and packet loss.
- **Packet-based measurement**: This technique involves measuring and analyzing individual packets in the network to detect and diagnose performance issues.
- **Traffic engineering**: This involves the use of network algorithms and tools to optimize network performance and capacity, including load balancing and traffic shaping.
- **Network monitoring**: This involves the use of tools and techniques to monitor network traffic and identify anomalies and security threats.

In addition to the above techniques, the use of machine learning algorithms for log analysis, performance measurement, and optimization of network slicing has been explored in research literature. Open-source network simulators such as NS-3 and Mininet have also been utilized to obtain network metrics in programmable network environments. Specific tools and techniques commonly used to obtain metrics in programmable networks include:

- **Network Traffic Analyzers** (such as Wireshark)
- **Network Performance Monitors** (such as NetFlow and sFlow[4])
- **Network Probes and Sensors**
- **Software-defined Network Controllers** (such as OpenDaylight and ONOS)
- **High-Level Monitoring Systems** (such as Prometheus[5])

Further research is required to investigate the effectiveness and limitations of these tools and techniques, as the field of programmable networking continues to evolve rapidly.

2.3.2 Log analysis overview

Real-time logging platforms are essential for measuring network performance and detecting issues in software-defined networking (SDN) controllers. Compared to traditional logging systems, they provide advantages such as:

- Real-time processing of log data
- Streaming to cloud servers
- Quick notifications, alerts, and visualization of events
- No need for a bespoke analytics platform
- Effective monitoring of wireless network segments

Monitoring and analyzing network performance is crucial in identifying issues and anomalies. Real-time logging platforms offer a powerful tool for network administrators to respond to issues effectively. [10] The log analysis landscape in software defined networking (SDN) environments is composed of two major approaches:

- **Machine learning algorithms**

  The use of machine learning algorithms in log analysis for software defined networking (SDN) environments has advantages over traditional parsing and compiler-based techniques. These algorithms can efficiently analyze large volumes of log data, detect patterns indicating security risks, and continuously improve their accuracy. They are also capable of handling diverse and unstructured log data. However, machine learning algorithms have limitations, including the production of false positives and negatives, high computational requirements, and dependence on training data quality. Choosing between these two techniques will depend on the specific goals and demands of the log analysis system.

- **Traditional parsing and compiler-based techniques**

  Traditional parsing and compiler-based log analysis techniques offer advantages such as a deterministic and straightforward approach, customization, and computational efficiency. However, these techniques may not be suitable for complex SDN environments and may be time-consuming and error-prone with unstructured log data. Additionally, they may not effectively identify complex patterns and behaviors, making them less effective for detecting security threats.

Both methods have their own strengths and limitations, and the choice between them will vary based on the goals and demands of a particular log analysis system. Effective use of metrics can help network administrators identify potential issues and optimize network performance. Metrics can also be used to inform network design and configuration decisions, such as determining the optimal number of switches or routers needed for a particular network. Moreover, real-time metrics can enable real-time network fine-tuning in programmable networks.

2.4 Cloud-Based Logging Systems

2.4.1 Kubernetes Native Logging System

Kubernetes is a popular container orchestration platform, that can serve as the VNF (Virtual Network Function) controller of a programmable network. The Kubernetes native logging system collects and stores logs generated by containers running on the Kubernetes platform. The system aggregates logs from all the containers running on the nodes in the cluster and stores them in a centralized location, making it easier to monitor and troubleshoot issues. The Kubernetes logging system comprises two main components:

- The **Kubelet**, which runs on each node in the cluster and is responsible for collecting and forwarding logs to the API server.
- The **Kubernetes API server**, which provides an interface for retrieving logs. The logs are stored in a persistent storage backend, such as Elasticsearch or Splunk, where they can be easily searched and analyzed.

One advantage of using the Kubernetes native logging system is that it is already integrated into the Kubernetes platform, making it easy to set up and use. Additionally, it provides a centralized location for storing and accessing logs, making it easier to troubleshoot issues across the entire cluster.

However, the Kubernetes native logging system has some limitations, such as a lack of flexibility in log collection and storage. It may not be suitable for organizations with specific logging requirements, such as those that require compliance with regulatory standards.

2.4.2 Third-Party Logging Systems for Kubernetes

While Kubernetes provides a native logging system, third-party logging systems can offer additional functionality and flexibility to organizations. These systems provide more options for log collection and storage, enabling organizations to customize their logging infrastructure to meet specific requirements.

- **Fluentd and Logstash**, are two popular open-source log collectors that support a wide range of input and output plugins. They can collect logs from various sources, including containers running on Kubernetes, and forward them to various storage backends, such as Elasticsearch, S3, or Hadoop. While these systems are highly flexible, setting them up and configuring them can be complex and may require additional resources and expertise.
- **Rsyslog**, is another open-source logging system that can collect and forward logs from various sources, including Kubernetes containers. It provides advanced filtering and processing capabilities, enabling organizations to customize how logs are collected, processed, and stored. Rsyslog supports various input and output modules, enabling it to collect logs from various sources and forward them to various storage backends, such as Elasticsearch or Syslog. One of the advantages of using Rsyslog is its advanced filtering and processing capabilities, which enable more efficient troubleshooting and analysis of issues.

While third-party logging systems offer more flexibility and customization options compared to the Kubernetes native logging system, they may also require additional resources and expertise to set up and maintain. Organizations should carefully evaluate their logging requirements and resources before choosing a logging system.

Chapter 3

System Design

In this chapter the architecture of PRINCIPALS is presented, as well as the proposed architecture for the metrics system, including the high-level overview of the system components, the design considerations and requirements.

3.1 PRINCIPALS framework

3.1.1 Innovative claims

The PRINCIPALS architecture will provide a novel mechanism for conducting defensive cyber-security operations (DCO) at a scale, pace, and precision that is unprecedented. PRINCIPALS will enable the detection and tracking of malicious activity across a 5G architecture, and provide the tools for effectively and rapidly countering such activity. PRINCIPALS will allow operators to write defensive (as well as network diagnostic) tools, which we call AMElets.

This powerful extensibility model can by summarized to:

- A novel architecture with safety and security guarantees derived from a combination of carefully constructed semantics.
- A domain-specific language (DSL) for expressing transient network computation
- Strict resource management.
- Cryptographic protections.
- Distributed authorization.

3.1.2 High Level Overview

The PRINCIPALS node architecture, as illustrated in figure 3.1, is based on three pillars:

- **Flexibility**: Any defensive architecture must be able to adapt to ever-changing threats. PRINCIPALS offers a complete and controllable programmability in all layers of the network stack.
- **Performance**: To ensure the scalability of PRINCIPALS, it is necessary to overcome the overhead associated with:

  - Per-packet or per-flow computation in near-real-time, which requires:
    - efficient handling of intra-node and inter-node communication.
    - Significant processing capabilities for complex analytical computations
    - Sufficient memory resources
  - Execution of multiple applications on the same infrastructure, which increases system complexity and the overall execution overhead.

- **Security**: PRINCIPALS is designed to support the injection of code in the data path, often by users that may not be fully trusted. Fine-grained security models are needed to increase usability as more precise policies can be specified and enforced.

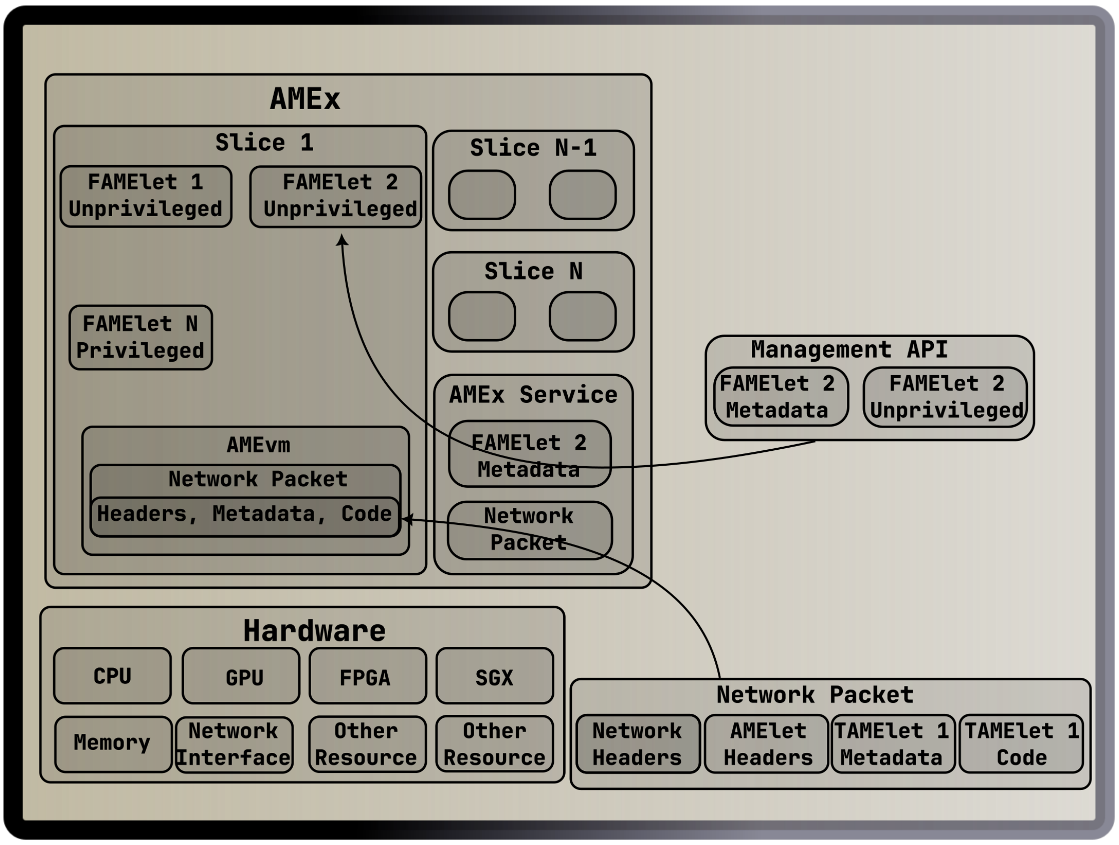


Figure 3.1: The PRINCIPALS node architecture.

3.1.3 Design

The following list provides information about the PRINCIPALS core components, At the core of this architecture is the Active Management Environment (AME), which enables safe programmability and adaptability in 5G networks by providing:

- **AMEis**: A novel domain-specific language (DSL) that offers inherent safety and security guarantees. AMEis is the instruction set that TAMElets are expressed in when transmitted in the network. Is very similar to BPF language. AMEis achieves two goals:

  - It is able to alter the state of FAMElets and consequently the state of a slice.
  - It can rely on native and performant execution of heavy weight tasks.

- **AMElets**: Custom-yet-adaptive logic expressed as code deployed on demand inside the network, possibly at scale.
  - FAMElets: software modules that execute inside AME execution runtime. FAMElets can be implemented in any Turing complete language. They come in the following forms:
    1. Unprivileged: User level programs that run inside a network slice (most common form).
    2. Privileged: Kernel modules running inside a slice.
    3. Management: Programs running outside network slices and inside the host operating system of the principals node.
  - TAMElets: are implemented in AMEis. They can be thought of mobile programs that are designed to execute as they transit through the 5G network. TAMElets are carried in network packets with a special header. The packet also encapsulates a payload of AMEis code along with metadata. The code payload is executed as it traverses the network.

- **AMEx**: The execution runtime. It is composed of a number of services and APIs and is responsible for controlling and mediating all accesses and calls between components (be it hardware or software). It provides:
  - The space where all FAMElets execute.
  - An interpreter (AMEvm) for the execution of TAMElets.
  - Facilities for loading/unloading AMElets using the AME service.
  - Guarantees that AMElets do not interfere with each other.
  - Mediation of communication between AMElets.
  - Mediation of communication between AMElets and hardware resources of the 5G node.
  - Grouping of AMElets into slices.
  - Overall resource management per slice.
  - Guarantees that all operations are in line with the security policy set by the infrastructure operators.

- **AMEvm**: is a simple VM that can run the mobile code carried by the mobile AMElets (TAMElets). It is a simple interpreter that executes the instructions defined in the AMEis instruction set. It serves as a sandbox for executing the code relaying calls to FAMElets functionality that resides inside the AMEx.

The implementation of the PRINCIPALS framework requires multiple components and functionalities, including FAMElets that are essential in achieving the desired outcome. These components, including FAMElets, must be packaged in a way that allows their deployment across different devices. To accomplish this, the framework utilizes Docker images, which provide flexibility in designing Network Functions and allow for implementation in any programming language. The Docker images are designed to be deployed using Kubernetes, which acts as a Virtual Network Function (VNF) controller, enabling efficient deployment of the framework. By utilizing Docker images and Kubernetes, the framework's components and functionalities can be deployed and managed in a scalable and flexible manner, providing a reliable and efficient solution for network infrastructure.

Docker images offer a variety of benefits that make them a suitable choice for the deployment of the framework components. One of the primary advantages is the flexibility they offer in the design of Network Functions. By allowing these functions to be implemented in any programming language, Docker images enable the creation of customized and efficient Network Functions that can be tailored to specific needs. This flexibility in design is critical in enabling the efficient deployment of the framework in diverse environments.

The docker images are hosted in a secure docker registry, accessible by authorized AMEx components.

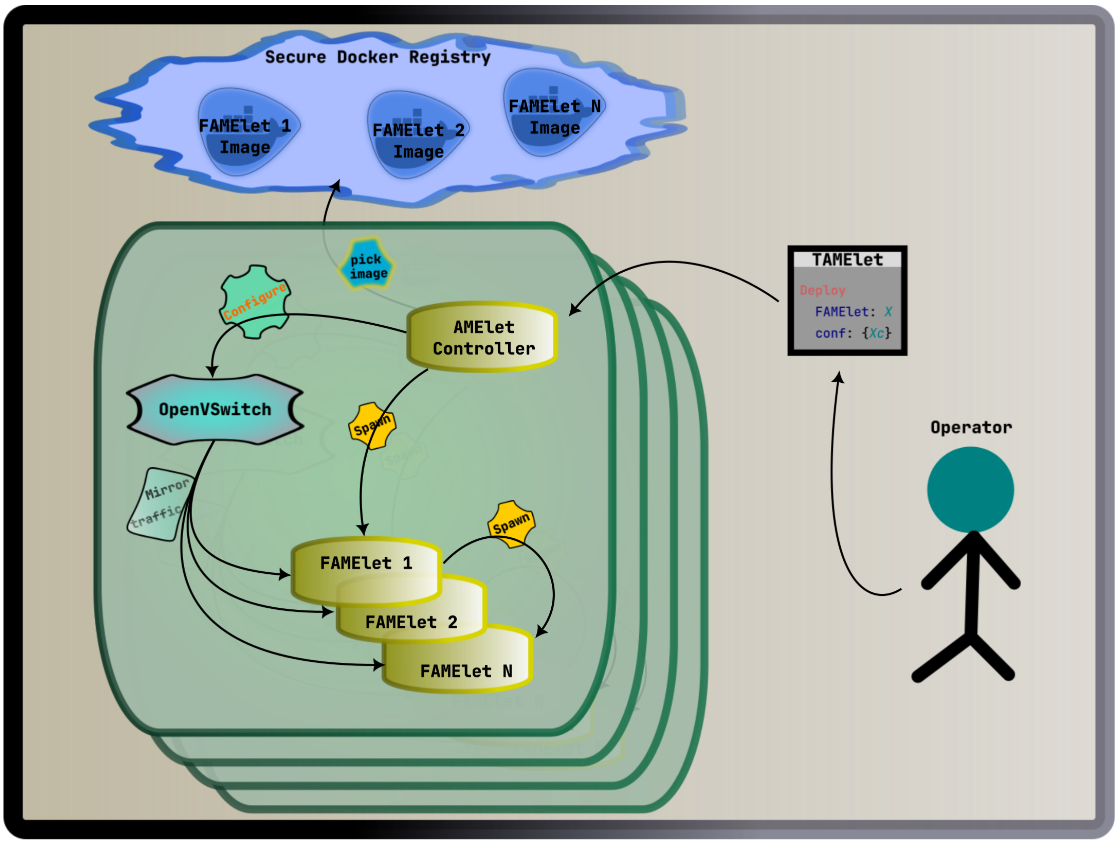


Figure 3.2: High level overview of PRINCIPALS framework

3.1.4 Kubernetes as a VNF controller

The deployment of these Docker images is simplified by utilizing Kubernetes as a VNF controller. Kubernetes provides a robust and scalable platform for the deployment of Docker images, enabling the efficient management of the framework components. As such, the combination of Docker images and Kubernetes as a VNF controller is a critical aspect of the PRINCIPALS framework, that enables the flexible and efficient deployment of Network Functions in diverse environments.

In addition to the ease of deployment provided by Docker images, Kubernetes offers a comprehensive set of built-in tools that enable the implementation of a robust programmable network architecture. Out of the box, Kubernetes provides tools such as services, service accounts, daemonsets, network attachment definitions, and namespaces, which are essential for building a highly scalable and efficient network infrastructure.

- **Pods** are the smallest deployable units of computing that you can create and manage in Kubernetes. Pods can run Docker images or other container runtimes, with deployments and daemonsets being the common ways to manage pods in Kubernetes. [1]
- Pod **deployment** is an essential aspect of managing groups of identical pods in Kubernetes. It provides a declarative way to specify the desired state of a group of pods and ensures that they are healthy and available to serve traffic. By defining a configuration file, you can create the desired number of replicas, container image to use, environment variables, and other necessary details. One of the most significant benefits of a deployment is the ability to scale the number of pods up or down based on demand. Additionally, it provides rolling updates that help in updating a subset of pods at a time. Overall, pod deployments are a flexible and powerful way to manage groups of identical pods in Kubernetes. [1]
- A **DaemonSet** is a Kubernetes object that ensures that a specific pod runs on all or some nodes in a cluster. It is typically used for system-level tasks such as logging or monitoring, where it is important to have one instance of the pod running on each node to collect data or perform other administrative tasks.
- **Services** are one of the key tools provided by Kubernetes that enable the exposure of a deployment of pods as a network service, which can be accessed by other pods or external entities. This functionality simplifies the networking infrastructure, allowing pods to be accessed via a single IP address and providing load balancing capabilities. By utilizing services, PRINCIPALS gives access to operators and implements the internal networking. [1]
- **Service accounts**, another essential tool provided by Kubernetes, allow for the management of permissions and access control within the cluster. Service accounts provide a mechanism for controlling access to resources within the cluster, ensuring that only authorized entities can access the network resources. [1] With this tool, PRINCIPALS ensure secure control of any OpenFlow switch (virtual or hardware).
- Kubernetes **volume mounts** are a valuable feature that allows storage volumes to be attached to containers in a pod, facilitating data persistence even after container restart or termination. Various storage backends such as local disks, network file systems, and cloud storage solutions can be used. The read-write access modes, access policies, and storage capacity can all be customized to meet specific requirements. [1] These capabilities make volume mounts a crucial component of Kubernetes' container orchestration abilities, particularly for running stateful applications. When creating a secure communication channel between pods while decoupling from existing networking models, volume mounts offer a viable solution. Specifically, the host path volume type of Unix socket is suitable for establishing a local network connection between pods without exposing the socket externally. This approach is particularly useful when implementing custom communication protocols or utilizing existing Unix utilities that depend on Unix sockets for inter-process communication.

- In addition to these functionalities, Kubernetes provides the ability to define **network attachment definitions** that enable the dynamic configuration of network interfaces, and namespaces which provide isolation and resource management capabilities.

To further expand Kubernetes' capabilities and enable more advanced networking features, various plugins and add-ons are available, including some for network virtualization, load balancing, service mesh, and more. Such plugins can assist organizations manage and secure their Kubernetes clusters, and to deploy modern, cloud-native applications more efficiently.

One popular plugin for Kubernetes is the Container Network Interface (CNI), which provides a standard interface for configuring network connectivity for containerized applications.

Advanced networking features can be enabled with CNIs like Antrea and Multus :

- **AntreaOVS** The Antrea OVS plugin is an open-source networking and security solution for Kubernetes that is built on top of OpenVSwitch. It provides advanced networking features such as load balancing, network policy enforcement, and traffic routing for Kubernetes clusters.

- **Multus** The multus CNI plugin is a Kubernetes extension that allows multiple network interfaces to be attached to a single pod. This allows for the creation of more complex network topologies in Kubernetes and enables the use of multiple network interfaces for different applications.

The combination of Antrea OVS plugin, multus CNI plugin, and OpenVSwitch provides a powerful solution for creating virtual networks in Kubernetes and testing network functions in a software-defined network (SDN) environment.

While OpenVSwitch supports the OpenFlow protocol and allows network administrators to control the flow of network traffic using a centralized controller, it cannot directly control physical OpenFlow switches/routers in the same way as virtual switches/routers. Physical OpenFlow switches/routers require their own controllers to operate, which would have to be integrated to the overall network architecture. Thus, a topology set-up with Antrea, Multus, and OpenVSwitch cannot be used interchangeably with physical OpenFlow switches/routers, as it would only be deployed with OpenVSwitch. However, the combination of these technologies can still provide a highly flexible and programmable solution for testing network functions in a software-defined network environment using virtual switches/routers.

A topology set-up with Antrea, Multus, and OpenVSwitch can provide several benefits beyond testing, even in a physical deployment. The combination of these technologies can enable the deployment of advanced networking features, such as network policies, load balancing, and traffic shaping, across both virtual and physical network infrastructure. This can improve network performance, security, and reliability, and enable the deployment of modern, cloud-native applications.

Furthermore, by leveraging technologies such as Antrea, Multus, and OpenVSwitch, vendor lock-in can be reduced, while increasing the portability of network infrastructure.

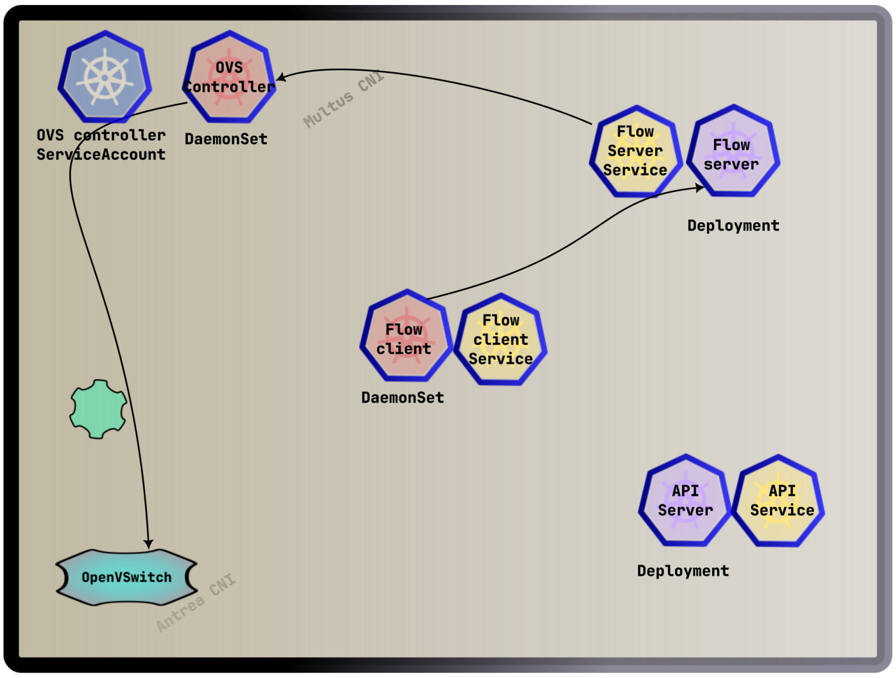
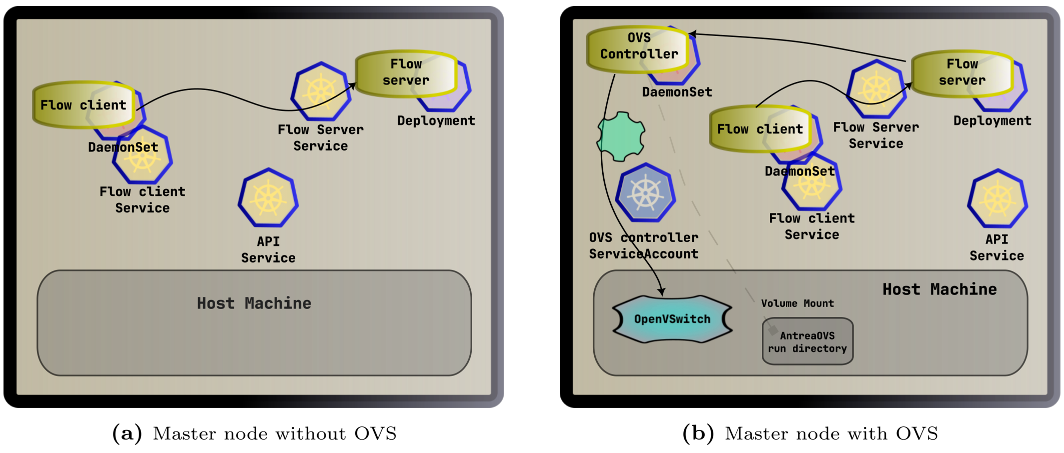


Figure 3.3: Kubernetes view of PRINCIPALS core components: Purple represents a Deployemnt, red a DaemonSet, yellow a Service and blue a ServiceAccount.

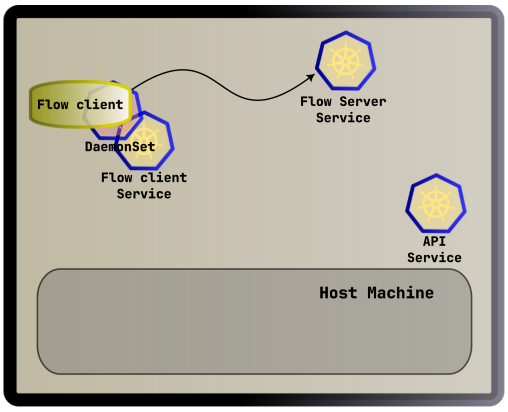


(a) Master node without OVS

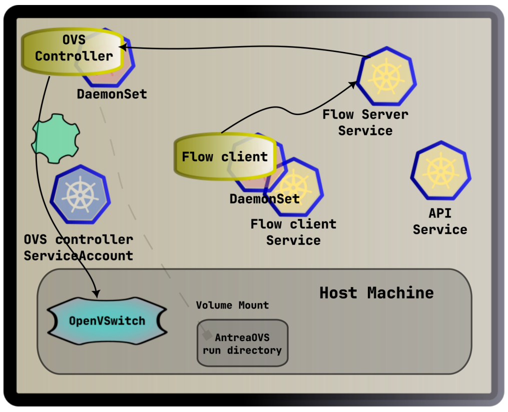
(b) Master node with OVS

Figure 3.4: PRINCIPALS master node: Flow Server is deployed on master node. The master node may or may not have an OVS Controller.

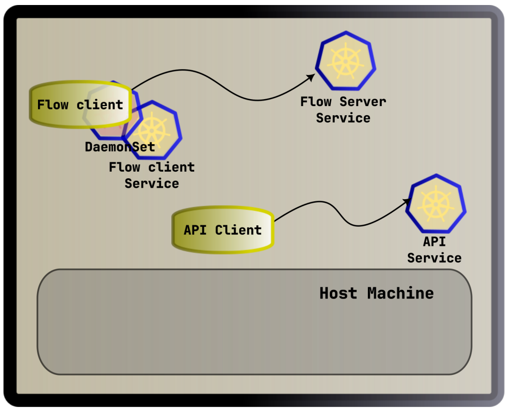
PRINCIPALS nodes are the spaces where all FAMElets execute. Figure 3.5 show some of the different forms that a node could take. The figure displays only the core FAMElets, including flow-client, OVSController, API-Server, and API-Client.



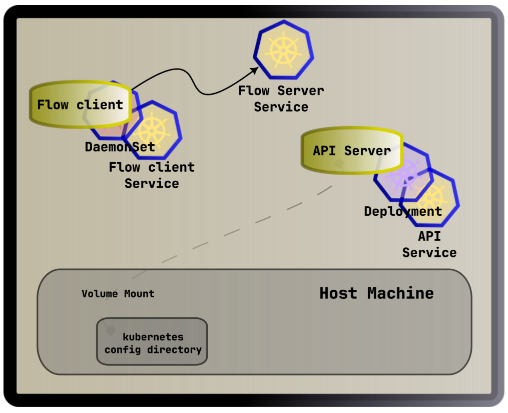
(a) Node without OVS Controller.



(b) Node with OVS Controller



(c) PRINCIPALS node for administration.



(d) PRINCIPALS API server could be in any node.

Figure 3.5: PRINCIPALS node: where FAMElets are deployed. A node could be bridged to OpenVSwitch by AntreaOVS.

3.1.5 Testbed

the PRINCIPALS model is designed under some assumptions regarding the necessary properties of the underlying compute platforms and the nodes on which the PRINCIPALS runtime operates, as well as the attacker capabilities:

- The runtime itself is protected from malicious interference by adversaries -remote and local- (i.e. Attackers that co-resident with a PRINCIPALS instance on a 5G node). This means that the runtime is isolated from attacker-controlled or attacker-owned slices (e.g. Through virtual-action), and that hardware supply chain or close access attacks are not taken into consideration.
- The underlying platform offers protection against the side channel attacks that could reveal sensitive information(e.g. cryptographic keys) used by the runtime and AMElets.
- The user plane will host malicious agents.
- 5G infrastructure will be under steady attack by persistent, well-resourced adversaries from both the user plane and from inside the 5G Network Operator organizations.
- 5G Access Network, core Network functions and elements, and endpoints have undiscovered security vulnerabilities and eventually these elements will be compromised.
- At least some of the nodes will offer some ability to securely initialize and attest to the integrity of a PRINCIPALS runtime.

The Kubernetes cluster deployed uses Docker v20.10.5 as the virtual machine driver for its pods. Networking is handled by Open vSwitch v2.14.0, which is an SDN compatible virtual network switch that supports up to the latest OpenFlow specifications.

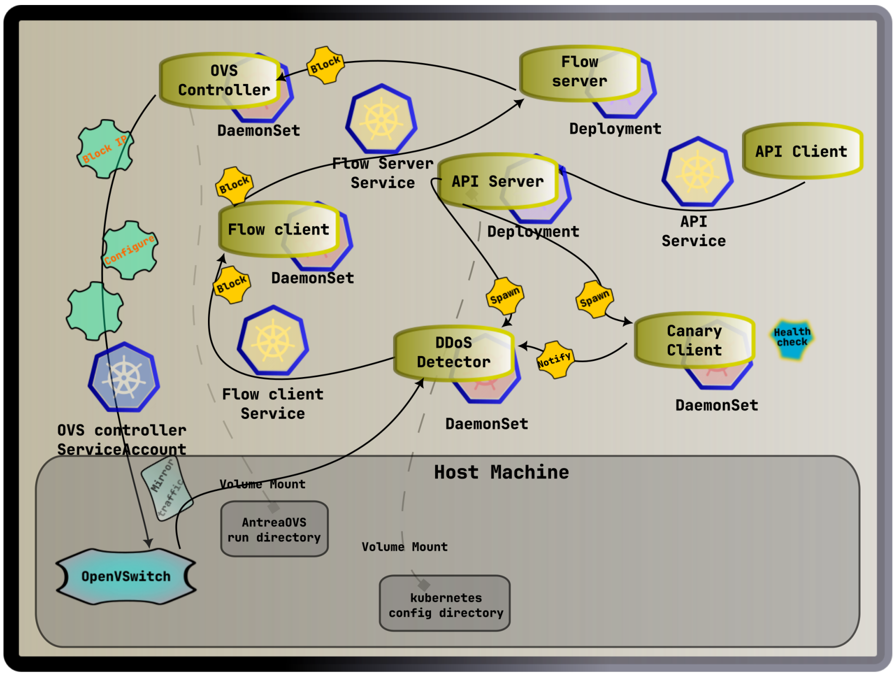


Figure 3.6: DDoS detection and mitigation components

Several pods are created for the DDoS detection and mitigation:

- The **canary client**, a service that monitors a service or a link to detect congestion. When congestion happens, a **DDoS detector** is launched, which locates and forwards the offending IP address(es) to the **Flow server**. Every node contains an instance of the canary client, whether it has an OVS Controller or not.
- The **DDoS Detector**, a service which analyzes mirrored traffic, searching for DDoS attackers. It is deployed as a DaemonSets in every node.
- The **Flow Client**, a service that listens for TCP requests from the **DDoS detector** pod. Each TCP request contains the malicious IP address to be blocked. The malicious IP address(es) is packaged along with the 'block' command as a JSON. This package is then forwarded through a TCP connection to the **flow server** pod. Every node contains an instance of the Flow Client.
- The **Flow Server**, a service that accepts TCP connections from **Flow Clients** from all nodes in the cluster. It is also connected to all **OVS Controllers** through TCP connections. When the service recieves a package by a **Flow Client** service, it is broadcasted in all OVS controllers. The **Flow Server** is only deployed on master node.
- The **OVS Controller** exposes a service that accepts TCP connections from the **Flow Server**. It runs on the same pod as the OpenvSwitch bridge and performs the requested actions on the OVS bridge.
- The **API server** pod, a service that listens TCP connections from the **API clients**. This FAMElet is responsible for deploying other FAMElets. It can be deployed in any node with all the necessary permissions for managing the Kubernetes cluster.

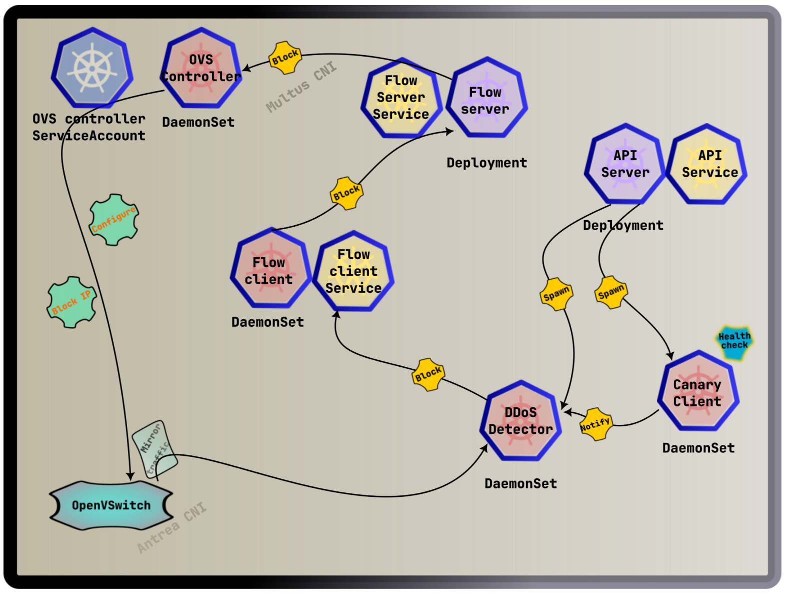


Figure 3.7: Kubernetes architecture for DDoS detection and mitigation components

On every node are deployed tens of pods, acting as benign clients, generating IoT traffic according to the traffic modeling measurements of real IoT devices. Each pod aggregates a large number of UEs with the cost of fidelity loss. Another set of pods is acting as malicious clients, generating DDoS traffic according to test scenarios (UDP flood, HTTP flood, SYN flood). Malicious traffic is also aggregated, each pod representing around 10k attackers.

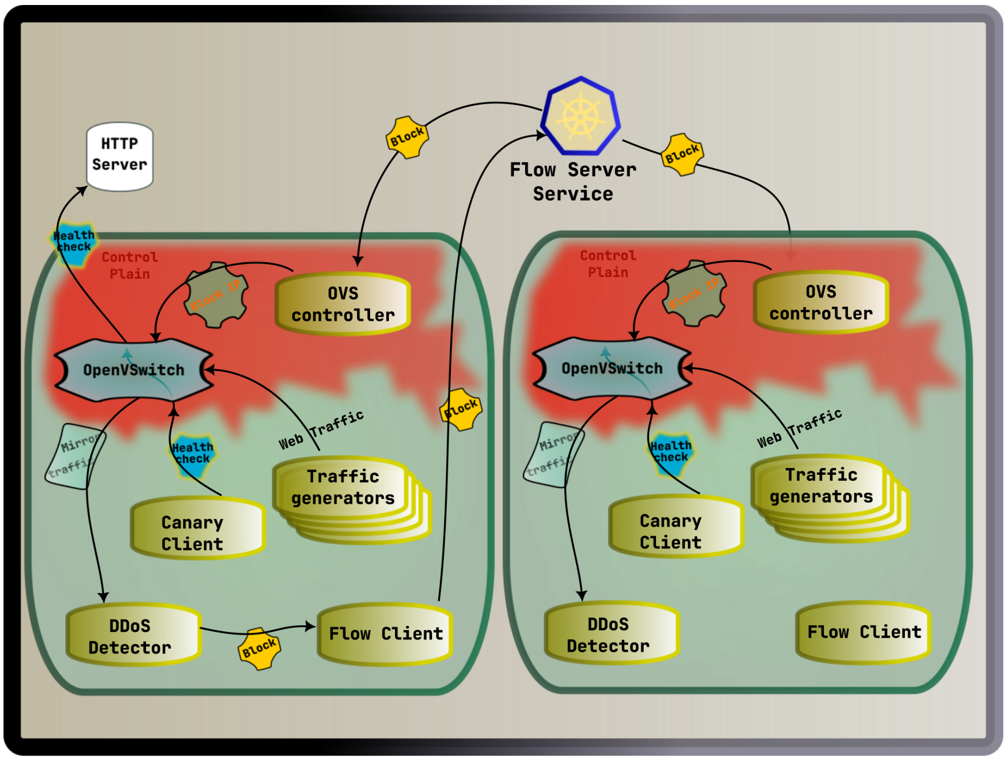


Figure 3.8: DDoS detection and mitigation components in a two-node scenario. The API server, client, and flow server are not shown in this overview in order to keep it simple.

3.2 Metrics system

3.2.1 Requirements

Designing a flexible and adaptable architecture is of paramount importance to ensure effective live monitoring, log aggregation, and analysis in a constantly evolving and uncertain cybersecurity landscape. The security primitives design is subject to changes, and unknown threats may arise at any given time, necessitating the need for an architecture that can accommodate new requirements seamlessly. As the cybersecurity environment is continuously developing, the architecture must be designed to be evolvable and capable of integrating new functionalities as needed. It is crucial to undertake research into evolvability to ensure the framework can cope with new requirements and changes in the threat landscape over time.

3.2.2 Concept

Centralized processing of the cluster's log stream can become unmanageable at scale. In scaled SDNs, minimizing per-packet computation is crucial for monitoring the state of the network and analyzing its behavior. The proposed design aims to reduce the computational overhead of log analysis and network traffic by transforming the stream of logs into a reduced stream of events, while distributing part of the computation away from the central aggregation point.

Since communication in PRINCIPALS primarily occurs via TCP connections with certificate validations and authorizations, data such as response delay and the response itself can be structured into valuable information to monitor the network's overall behavior. For instance, instead of logging the request traffic and then comparing timestamps in the central aggregation point to evaluate the response delay, the delay could be computed directly by the pod's runtime. Essentially, rather than analyzing traffic centrally in a per-packet basis, each pod producing the information can also perform a small portion of behavioral analysis, thus minimizing the computational overhead of the aggregator.

3.2.3 Design

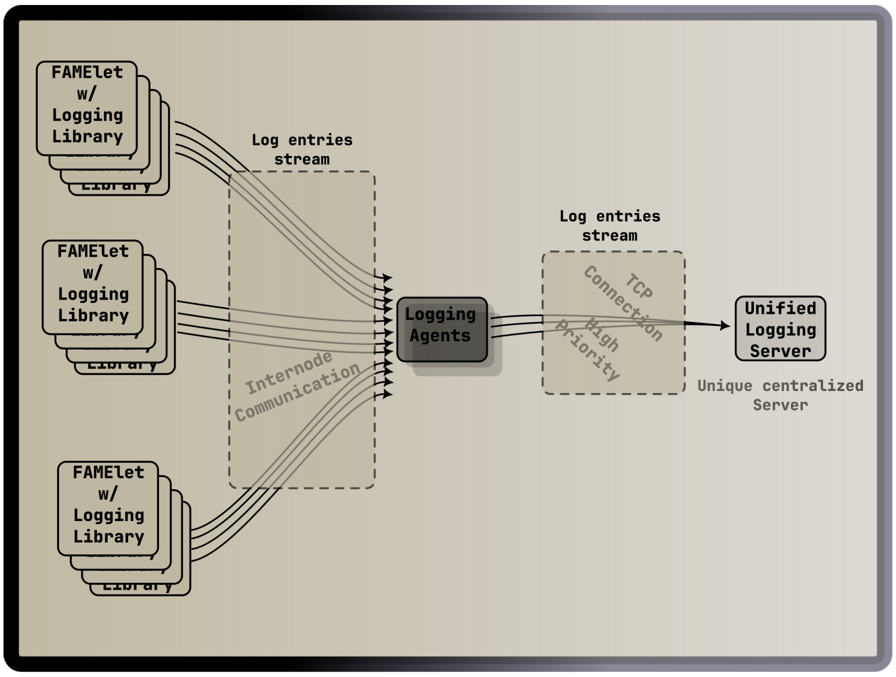


Figure 3.9: Metrics system high-level overview

The system is divided into three main components:

1. A fast, minimal **logging library** that includes part of the data analysis and preprocessing functionality and communicates with a log aggregation agent. The library is designed to have the same API as the pre-existing log library (Golang's `log` library), to ensure backwards compatibility. All existing debug and info logs will be preserved as is, while all extra functionality will be wrapped inside the library core.

2. An **agent service**, with fast serving capabilities, so that the executables that produce logs are not blocked because of congestion. Every Kubernetes' node must have at least one agent.

   The connection between the library and the agent should not be considered uncompromisable. Decoupling from classic network security and authorization techniques could be a good practice, as it would take advantage of the orchestrator's functionality (Kubernetes). For instance, using a Unix socket that resides on the containerized file system of the executable that uses the logging library, and leaving Kubernetes to handle how this socket is mounted or forwarded to the other endpoint, decouples the security management of the data channel from the implementation of the actual logging functionality.

   In case a malicious executable uses the logging library, it would try to log into a Unix socket, when the connectivity of such socket depends on the deployment permissions of the malicious pod. If the malicious pod is able to communicate with the logging agent, that would mean that the AME part responsible for deploying pods with special permissions is already compromised.

   By employing the security policy and permissions needed to deploy the socket connectivity (e.g. with Volume Mounts), we can assume that only FAMElets and TAMElets would be able to use the library.

   There are two options for the agent deployment:

   (a) a **DaemonSet**. That would result in exactly one agent per node. Depending on the scale of the network, the agent may need many worker threads to serve the Unix socket (or sockets).

   (b) a **Deployment**. In case that one agent per node is not enough, Kubernetes can be configured to ensure that multiple replicas of a pod are running on each node in a cluster. This can be achieved by setting the "podAntiAffinity" field in the Deployment's configuration to "requiredDuringSchedulingIgnoredDuringExecution", along with the appropriate label selector. Such configuration ensures that each node will have at most one pod with a specific label, which effectively distributes the replicas across the nodes. By setting the "replicas" field to a value greater than the number of nodes in the cluster, the Deployment will ensure that there are multiple replicas of the pod running on each node. That can the deployment a little bit more complicated, but allows the agent as a program does not need to be multithreaded and care about handling multiple sockets as a load balancer.

3. A **logging service**, available only to logging agents. This service is responsible for:

   (a) **Log aggregation**: This service will act as a centralized point of aggregation for all logs (debug logs and events). Regarding the debug logs, the requirements of a log aggregator are:

   - on-demand real-time log output. This can easily be achieved by printing to stdout and let Kubernetes' logging system API act like a frontend for the logging service.
   - log preservation for future inspection. Preserving log entries could be challenging if the data amount is huge. Techniques like file rotation should be applied.

   (b) **Log reordering and filtering**: The system is designed with the evaluation of specific network functions as its main requirement, measuring response time being its most common functionality. Since Debug logs existed prior to the logging system integration, many debug logs do not serve any purpose on log analysis. The server is able to identify logs that are registered as ”interesting” and filter out the rest of the stream.

Also, the log stream may include unordered entries, due to traffic congestion and the race conditions in accessing the service, from the agents' side. It is therefore important to keep the clocks of the nodes and pods synced with a trusted NTP server (stratum 2 at most), so that the server will be able to correctly sort the logs and pass them to the live-parser.

(c) **Log analysis**: The last mentioned yet most crucial part of the system. The logging service should be able to -at least- measure the response of a given network function by timestamp comparison. A more sophisticated approach is to design a system reliant to noise. Noise is considered to be:

   i. Packet loss, which can be mitigated using a TCP connection.

   ii. False state detections (e.g. server unreachable), which can be mitigated by aggregating and curating events

   After removing any noise the, stream of logs is transformed to a stream of events, which is then forwarded to the parser. The design of the parser is a combination of finite state machines and compiler based techniques.

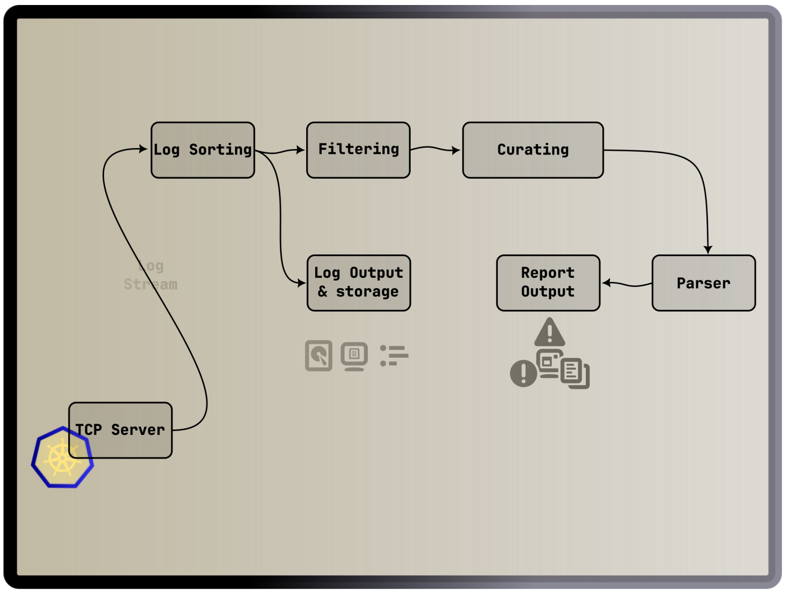


Figure 3.10: Metrics system functionality

In scaled topologies, log traffic could affect the overall performance of the logging server, essensially affecting the performance of the FAMElet by blocking on channel congestion. To mitigate this, channels between agent and server could be separated into channels for the Aggregation Server(s) and channels for the Analysis Server. The Aggregation channel can be rate limited so it does not impact the network's capacity. A UDP connection can be used for aggregation channel, and multiple replicas of aggregation servers could be used as load balancers.

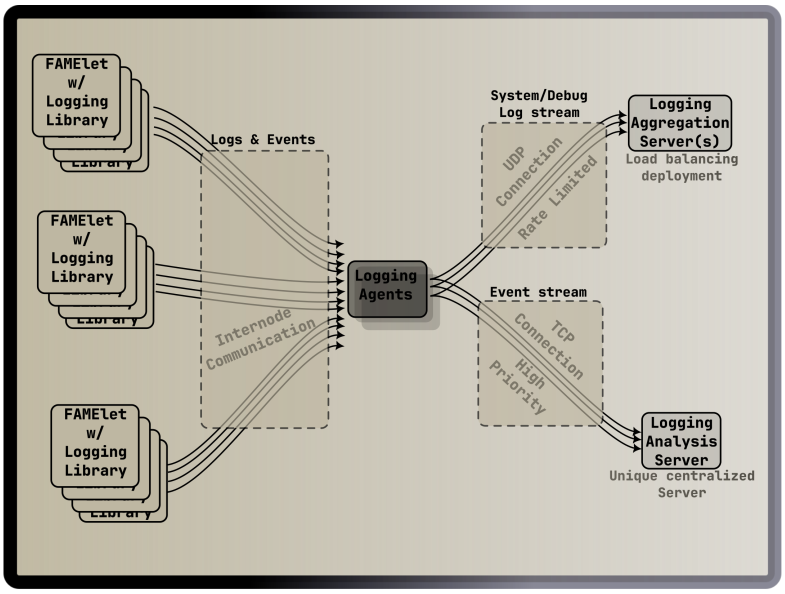


Figure 3.11: Metrics system with separate streams for log aggregation and log analysis

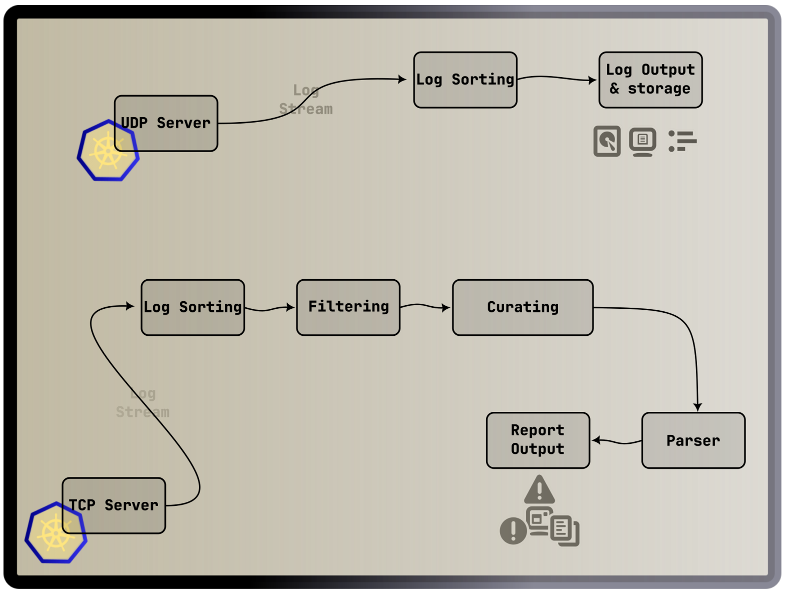


Figure 3.12: Metrics system functionality after log channel separation

Such separation essentially allows the Analysis channel to carry less data than the Aggregation channel. Log quantity can be significantly reduced by transfering computations like Ripple eliminatin algorithms in the logging library, and to the agents. This ultimately allows the Analysis channels to transfer less -but more valuable- data, compared to the whole log stream, that can also be prioritized, so that the insights of the network are not bottlenecked by the channel's capacity.

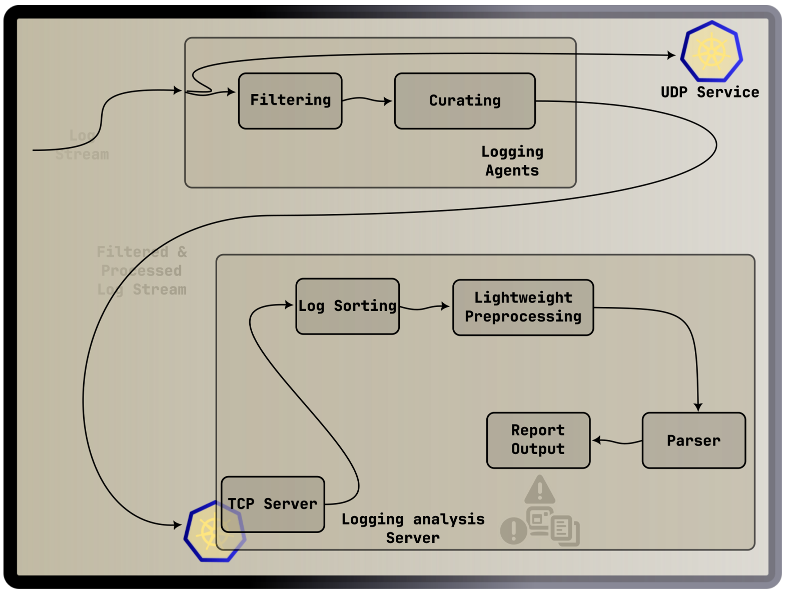


Figure 3.13: Metrics system functionality. Moving part of processing before final aggregation.

Chapter 4

Implementation

In this chapter, the detailed design and implementation of each component of the metrics system proposed in chapter 3 is presented. Furthermore, the integration of the metrics system with the PRINCIPALS evaluation platform is showcased.

As previously stated, the system requirements mandate the ability to measure various metrics related to DDoS attack mitigation. The cssystem's implementation aims to provide as many of the following metrics as possible:

1. **Percentage of attack traffic dropped**: Determine the percentage of attack traffic that is detected and dropped by the DDoS defenses.
2. **Legitimate traffic's goodput**: Determine the goodput of legitimate traffic reaching its destination.
3. **Delay and loss rate**: Measure the delay and loss rate of legitimate packets.
4. **Percentage of legitimate packets delivered**: Determine the minimum packet delivery rate needed to keep services running during the attack.
5. **Delay in detecting and responding to the attack**: Measure the time it takes to detect the attack and initiate mitigation response.
6. **False Positive Rate**: Calculate the false positive rate of the DDoS detection mechanism based on legitimate network traffic.
7. **Legitimate services availability during the attack**: Determine the number of services able to receive sufficient traffic to remain available during the attack.
8. **Amount of attack packets and number of hops traversed**: Measure the collateral damage caused by the attack and how closely defenses can block the attack.
9. **Impact of the DDoS attack on the network without any mitigation in place**: Determine the impact of the DDoS attack on the network without any mitigation in place, and the impact of the mitigation algorithm on the attack.
10. **Application Throughput and Latency**: Measure the impact of the DDoS attack on service/application quality of service while the mitigation algorithm is in effect, such as on VoIP calls, 4k streaming video, and FTP.

11. **QoS service metrics**: Analyze the Mean Opinion Score time series distribution of the VoIP service as a QoS service metric.

12. **Achieving BAA metrics**: Evaluate whether the mitigation algorithm achieved the BAA metrics (60 seconds for 1 billion node botnet in phase 1).

13. **Infrastructure Switch Support**: Determine the optimal distribution of infrastructure switches throughout the network that need to be instrumented to detect and mitigate against the DDoS attack, considering the Benefit-Cost Analysis.

The metrics system plays a critical role in PRINCIPALS by serving as the main logging engine in the framework and providing real-time metrics and evaluation of running services.

4.1 Metrics and Logging System Architecture Overview

The **log flow** begins with the clog **logging library**, which produces the logs. The logs are then collected by the **logging agents** and forwarded to the **metrics server**. The logging agents are responsible for serving all instances of clog and forwarding the logs to the main server.

Once the logs are forwarded to the metrics server, they are **saved** to persistent storage. The server is also responsible for **exposing** the log entries for real-time inspection through the native Kubernetes logging system. The log entries may be received in a slightly shuffled order, so the server sorts them to ensure they are in the correct order.

In addition to log aggregation, the system includes **log analyzing** functionality. The main metrics server **filters** log entries for parsing and has **log parsing** functionality that can **produce metrics** related to DDoS attack mitigation. It can also generate reports with interesting metrics and events, as well as provide a **real-time state report**.

To facilitate easy inspection of the reports generated by the metrics server, a metrics system's **frontend** is included. This frontend consists of a set of shell scripts that act as wrappers for Kubernetes API calls and provide basic functionality such as filtering. Together, these components form a robust metrics system that is critical for monitoring and evaluating services within the PRINCIPALS framework.

4.2 Metrics Server

The current implementation is a monolithic multithread server, containing all the functionality shown in figure 3.10. The scale of the currently tested scenarios permits this approach.

4.2.1 Agent Side Communication

The server accepts TCP connections from all the agents and serves them concurrently. There is implemented an aggregation internal channel, as in the agent. The server pings all the agents in a fixed time interval. That way the agents can anytime know if the server is up.

It is worth noting that the metrics server can be deployed without utilizing the logging agents and the logging library, as long as the logs keep a specific format (see Appendix A). It is not recommendet to do so, because the complete system guaranties that no logs are missed, and no execution time overhead is introduced to the security primitives due to network congestion when logging at scale (see Chapter 6).

4.2.2 Log Sorting

A thread consumes the aggregation channel and feeds the log stream to a sorting mechanism. This mechanism caches a variable number of logs and sorts them by their timestamps. The number of log entries cached depends on the log rate and the dispersion of the logs in terms of their creation time and arrival.

The sorted logs are then forwarded to two separate internal channels:

1. A channel for the mechanism that saves and outputs logs
2. A channel for the analyzing mechanism

Listing 4.1: Simplified data flow in the server. Safety checks and concurrency control have been removed.

```
func handleConnection(c net.Conn, toSorter chan []byte){
    reader := bufio.NewReader(c)
    str, err := reader.ReadBytes('\n')
    log.Printf("Received log: \%s", string(str))
    toSorter <- str
}

func main() {
    ...
    toSorter = make(chan []byte, 512)
    toOut = make(chan []string, 512)
    toAnalyser = make(chan []string, 512)
    ...
    go outputLogs(toOut)
    go analyseLogs(toAnalyser)
    go sortLogs(toSorter, toAnalyser, toOut)

    listening: for {
        cli, err := listener.Accept()
        go handleConnection(cli, toSorter)
    }
}
```

4.2.3 Log Analysis

This implementation involves the procedural parsing of log entries using a finite state machine (FSM) approach, wherein the log entries serve as input, and the FSM's states represent different stages of the attack. The analysis process entails the traversal of states and the tracking of transitions from one state to another. The final state signifies the successful mitigation of the attack, and a report is generated with various metrics, including the percentage of goodput before, during, and after the attack, the percentage of attack traffic dropped, and the detection and response delay.

Upon reaching the final state, the metrics server produces a report that is capable of re-evaluating the validity of the time measurements by cross-checking the timestamps and identifying any out-of-order entries that may have affected the traversal of states. The addition of the metrics system to the PRINCIPALS framework has enabled accurate fine-tuning of the detector's and canary's algorithms parameters, leading to improved detection and mitigation capabilities eliminating any false detection, while successfully mitigating all three DDoS attack scenarios.

The system is able to measure the following:

- dDos attack detection delay

- dDos attack response delay
- dDos attack mitigation delay
- percentage of legitimate packets delivered
- percentage of attack traffic dropped

In order to accurately determine the delay times, the system requires knowledge of when the attack begins. For testing purposes only, the attackers are allowed to log into the system, enabling the system to identify the start of the attack. Another crucial timestamp is when the attack takes effect, which means when the attacking server becomes unable to serve regular clients. By analyzing the log entries of the Canary pods, the system can ascertain when the server is unreachable. The Canary pod, responsible for monitoring, logs whether it receives a response when pinging the server. However, there are instances where the Canary's request times out without the server being down, and conversely, there are occasions during an attack when the server is down but the Canary pod occasionally receives a response. The analyzer possesses a mechanism that accurately determines the server's actual state, unaffected by the aforementioned "noise."

The timestamps for other significant events, such as the detection of attackers and the execution of blocking commands, are precisely determined based on the logs generated by the DDoS detector pod, the flow server, and the API server.

4.2.4 Log Output and Preservation

The current implementation depends on the Kubernetes logging system for live monitoring. The metrics server prints the log entries for Kubernetes to manage. As for persistent storage, the entries are stored in a file. File rotation is not implemented as it was not mandatory due to the quantity of the log entries in the testing scenarios.

Listing 4.2: Log output

```go
func outputLogs(logs chan []string){
    for {
        msg := <-logs
        timestamp, _ := strconv.ParseInt(msg[3], 10, 64)
        tt := time.Unix(timestamp / 1000000, 1000*(timestamp % 1000000))
        print(tt.UTC().Format(time.StampMicro), " ", strings.Join(msg[:3], " "), "
            ", strings.Join(msg[4:], " "))
        fmt.Fprint(clusterLogging, tt.UTC().Format(time.StampMicro), " ", strings.
            Join(msg[:3], " "), " ", strings.Join(msg[4:], " "))
    }
}
```

4.3 Logging Library (clog)

The clog library is developed as part of `logging package` and is the way to insert logs to the logging system and is responsible for:

1. minimal formating adding the host name and name of executable to the log entry
2. not blocking during logging
3. forwarding log entries to node's logging agent

4.3.1 API and integration

In order to incorporate the logging library into an existing Golang program that utilizes the `log` library, it is only necessary to substitute the `log` import with `logging`. By renaming , the `logging` import to `log` seamless integration can be achieved, as the library implements exactly the same calls as the `log` library.

Listing 4.3: Importing the library

```
import (
    ...
    "flag"
    log "logging"
    "net"
    ...
)
```

4.3.2 Library - Agent Communication

Communication between the logging agent is achieved using a Unix Socket. As both the agent and the executable that uses the library run on the same node, this is the most direct method. The socket resides in the host machine's file system and is mounted by Kubernetes to the pods that use the library and agent of the node.

With a simple write call, the formated log entry is forwarded to the agent.

Listing 4.4: Write to socket

```
    if err == nil {
        _, err = l.conn.Write(p[:np])
    }
```

If the write call to the agent's socket fails, `fixit()` function is called and a mechanism to manage the reconnection to the agent is launched.

Listing 4.5: Handling failed writes

```
var control struct {
    req chan struct{}
    ans chan struct{}
}

func fixit() {
    agMx.Lock()
    if !tryingToConnect{
        agMx.Unlock()
        control.req<- struct{}{}
        <-control.ans
    } else { agMx.Unlock() }
}

var agMx sync.Mutex
var tryingToConnect bool = false

func stateCheck() {
    for {
        <-control.req
        control.ans<- struct{}{}
        agMx.Lock()
        if !tryingToConnect {
            tryingToConnect = true
            agMx.Unlock()
            connectToAgent(0)
            agMx.Lock()
            tryingToConnect=false
        }
        agMx.Unlock()
    }
}
```

This mechanism runs in parallel. If the agent does not respond, the FAMElet does not block and continues to execute. During this period, any logs generated are output to the standard output stream. When the pod reconnects to the agent, it reports the number of missed logs produced during the outage. Currently, the implementation does not collect the missed logs after the reconnection because the agent going down is highly unlikely in the current testbed. However, the logs remain available through the Kubernetes API. Therefore, if there comes a point where agents are expected to go down, implementing the collection of missed logs will be straightforward.

4.4 Logging Agent

4.4.1 Client Side Communication

The agent is implemented as a multithread application. The agent is listening to the Unix Socket for clients and spawns a server routine for each client. It accepts every conncection to the socket without any security measures. It is assumed that the socket is a secure place in the filesystem of the host-machine, non accessible by adversaries. The server routines route the log entries they receive to a single internal go channel[2]. That channel acts as an aggregation channel.

4.4.2 Server Side Communication

A thread is serving the aggregation channel, forwarding the entries to the logging server via a TCP connection. Like in clog library, there is implemented a mechanism to ensure a reliable connection with the server.

Listing 4.6: Similar mechanism as in Listing 4.5 to manage disconnections with the server. The only difference to this implementation is that the call to fixit will block until connection is up, ensuring that no logs are missed. Using a buffered aggregation channel enables the continuous serving of the unix sockets regardless of whether the other end of is blocked.

```go
func fixit() {
    log.Println("Fix it")
    control.req<- struct{}{}
    <-control.ans
}

func stateCheck() {
    for {
        <-control.req
        _, err := fmt.Fprintf(srv, "\%s agentPing\\n", *args.nodeName)
        if err != nil {
            log.Println("Closing broken connection to server")
            closeServerConnection()
            err = connectToServer(0)
        }
        control.ans<- struct{}{}
    }
}
```

On top of that mechanism, there is a goroutine spawned that waits pings from the server in fixed time intervals. If there are missed pings, the mechanism for reconnection is launched.

4.5 Integration and results

4.5.1 Attack scenario Integration

As previously mentioned, three DDoS attack mitigation scenarios were set up. The purpose of the metrics system is to measure the performance of the mitigation procedure, so it had to be integrated to the existing codebase.

To integration process involved:

1. Patching the logging library to PRINCIPALS security components:
   - OVS Controler
   - Flow Server
   - Flow Client
   - API Server
   - API Client
   - DDoS Detectors
   - Canary Clients
2. Mounting the logging volume to the deployements of the above components.
   (a) Kubernetes top-level yaml configuration files, as for the core components.
   (b) Deep inside PRINCIPALS API's code, where Golang Kubernetes api library was utilized for the deployement of the detectors and the canary.
3. Deploying logging agents as DaemonSet
4. Deploy the metrics server on master node.

4.5.2 Demo Automation

Kubernetes' YAML configuration files are a valuable abstraction for utilizing the Kubernetes API; however, they can be less convenient when there is a need for repeated deployment of the same cluster with minor changes to the image version. During the development and testing of the PRINCIPALS framework and the metrics system, various scenarios required repeated deployment with different versions of the FAMElets to facilitate comparison of results. To avoid ending up with multiple YAML files with minimal differences, a higher abstraction level was implemented using Bash and GNU Coreutils.

The implemented approach utilized template files with placeholders, together with a minimal configuration file defining the image versions and registry address. Subsequently, a Bash-based YAML generator was designed to create the YAML files for the attack scenario. This higher level of abstraction allowed for the easy generation of multiple YAML files, eliminating the need for manual editing of each file.

The approach described above provided a more streamlined and efficient method for deployment, saving time and minimizing errors that could arise from manual editing. This approach could be useful in other scenarios requiring the deployment of multiple Kubernetes clusters with slight variations. Overall, the utilization of Bash and GNU Coreutils for the generation of YAML files proved to be a successful approach in the development and testing of the PRINCIPALS framework and the metrics system.

4.5.3 Results

Three attack mitigation scenarios were tested with the metrics system itegrated:

For UDP flooding, the link of UEs is flooded with 1500-byte UDP packets generated by a traffic generator developed by the performer. For HTTP flooding, large POST requests of 1MB each were sent to a target HTTP server to flood the link of UEs. Finally, for SYN flooding, a large volume of SYN packets to a target server to deplete its socket state was generated by a traffic generator developed by the performer. The malicious traffic modelling is focused solely on congesting the resources under attack (bandwidth or sockets) and does not follow any specific scaling criteria as long as the attack is successfully executed.

Two clients were used to evaluate the impact of the DDoS attacks and mitigations on real applications. The first one is a SIP client and server to emulate thousands of calls per second. The sipp traffic generator4 was used for the VoIP related measurements. For all the experiments, the G.711 codec was used as the underlying codec for RTP (voice) streams. The second is a secure copy (scp) client that is found in most Linux distributions. The scp client is used to measure the impact on file transfer applications.

Before we describe the evaluation of each attack scenario, it is important to document the application-level metrics that are used to demonstrate the impact of DDoS attacks. For the VoIP application two metrics were calculated:

1. The percentage of successful/failed calls: how many calls were able to go through before, during and after the attack is mitigated.
2. The Mean Opinion Score: for the successful calls what is the perceived quality of calls before, during and after the attack is mitigated. Since we use the G.711 codec for the RTP (voice) streams, the maximum score that can be achieved is 4,4.

For the file transfer application, the transfer rate (unit is Mbps) was used to evaluate the impact of the DDoS attack.

- **Udp flood attack** The next four graphs show the attack timeline for the UDP flood attack. The first two graphs display the VoIP metrics on the secondary Y-axis while the third and fourth graph concern the file transfer application. We observe that in all graphs, the detection delay is 9 seconds. The detection delay is defined as the time between the first attack packet was sent and the time the detection algorithms flags the malicious UEs for blocking. In the case of VoIP we observe that both the number of successful calls and Mean Opinion Score (MOS) is sustained during the first 8 seconds of the attack (this is due to the facts that some alls were already in progress and that the link is not immediately saturated) but then both metrics rapidly deteriorate. The number of successful calls drop to 1% while MOS drops to 0.9 during the attack. In the case of file transfer, the rate drops from 110Mbps down to 40Mbps during the attack.

  The VoIP application recovers immediately after the attack is mitigated. We observe that successful call rate return to 100% and MOS goes back to 4,4 after the attack is mitigated (around second 45 of the timeline). The file transfer also recovers but it takes close to two and half minutes to go back to the original transfer rate before the attack takes place. This is due to how the TCP protocol works and how congestion is handled by the Linux kernel (secure copy protocol is TCP-based). Figure 4.4 shows the recovery progress of the file transfer after the attack.

  In all scenarios, the response time was maximum 16 seconds and the false positive rate is 0%.

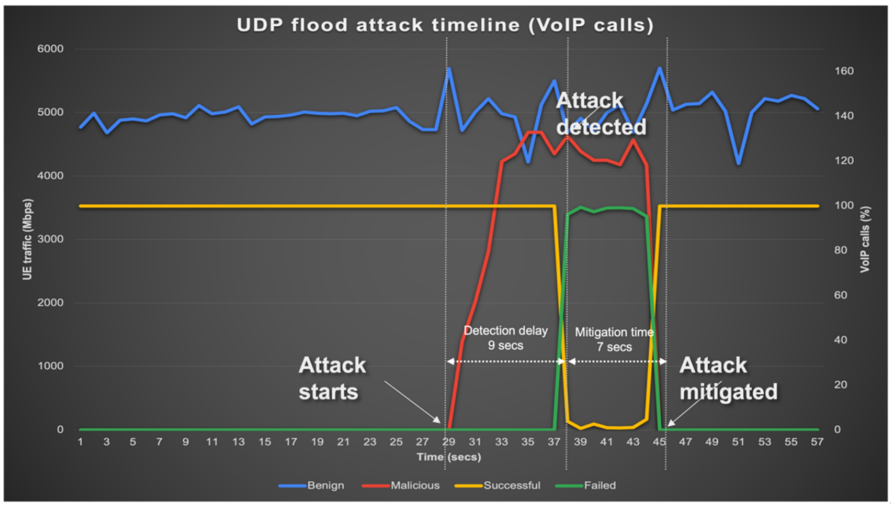


Figure 4.1: UDP flood attack timeline (VoIP calls)

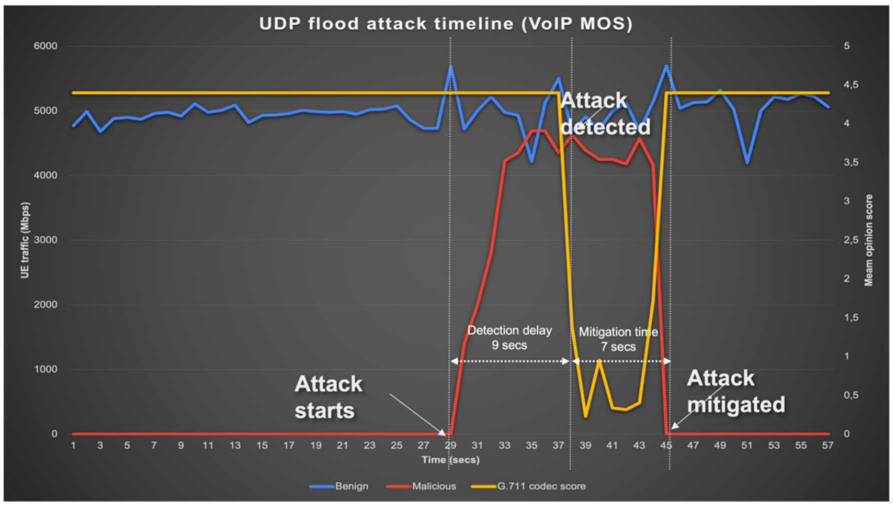


Figure 4.2: UDP flood attack timeline (VoIP MOS)

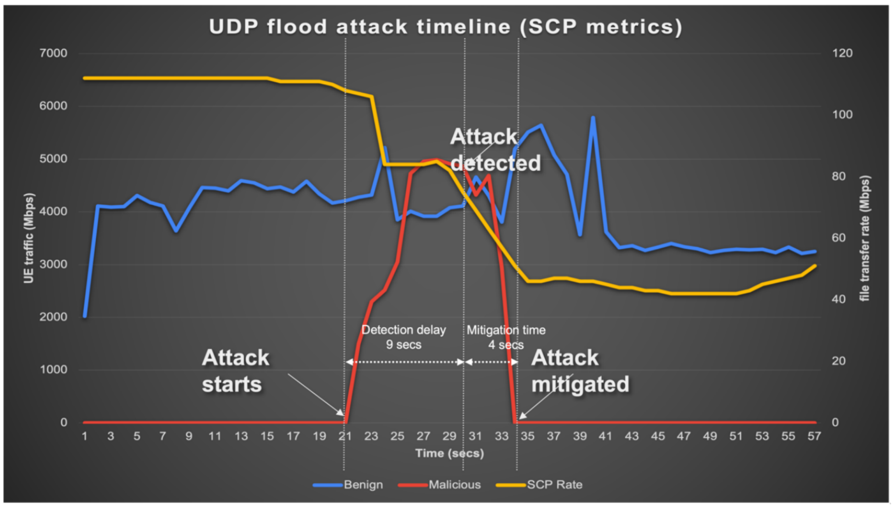


Figure 4.3: UDP flood attack timeline (SCP metrics)

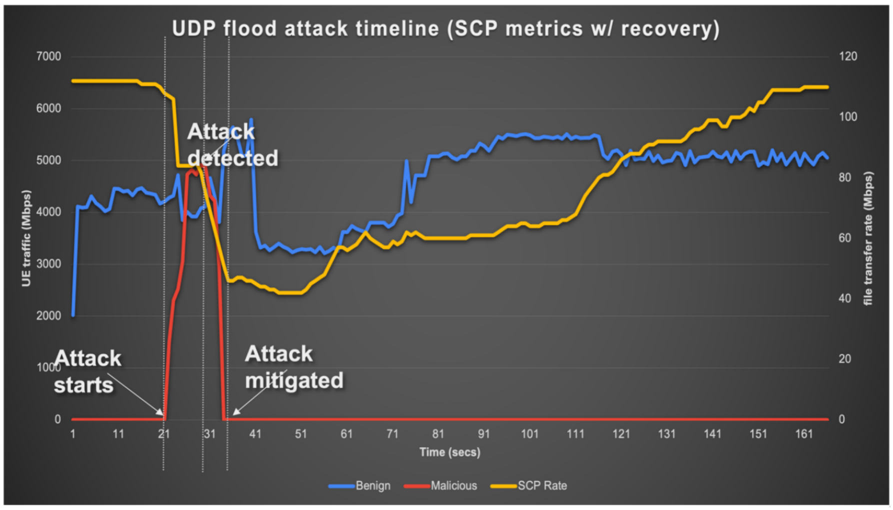


Figure 4.4: UDP flood attack timeline (SCP metrics) with recovery progress

- **HTTP flood attack** The next three graphs show the attack timeline for the HTTP flood attack. The first two graphs display the VoIP metrics on the secondary Y-axis while the third graph concerns the file transfer application. In the case of VoIP we observe that both the number of successful calls and Mean Opinion Score (MOS) drop immediately once the attack starts. Successful calls drop to approximately 40% during the attack and MOS goes down to 1,5. In the case of file transfer, the rate drops from 110Mbps down to 25Mbps during the attack.

  The VoIP application recovers immediately after the attack is mitigated. We observe that successful call rate return to 100% and MOS goes back to 4,4 after the attack is mitigated (around second 45 of the timeline for VoIP and second 35 for timeline of SCP). The file transfer also recovers immediately once the attack is mitigated. This is due to how the TCP protocol works and how congestion is handled by the Linux kernel (secure copy protocol is TCP-based). Figure 4.7 shows the recovery progress of the file transfer after the attack.

In all scenarios, the response time was maximum 28 seconds and the false positive rate is 0%.

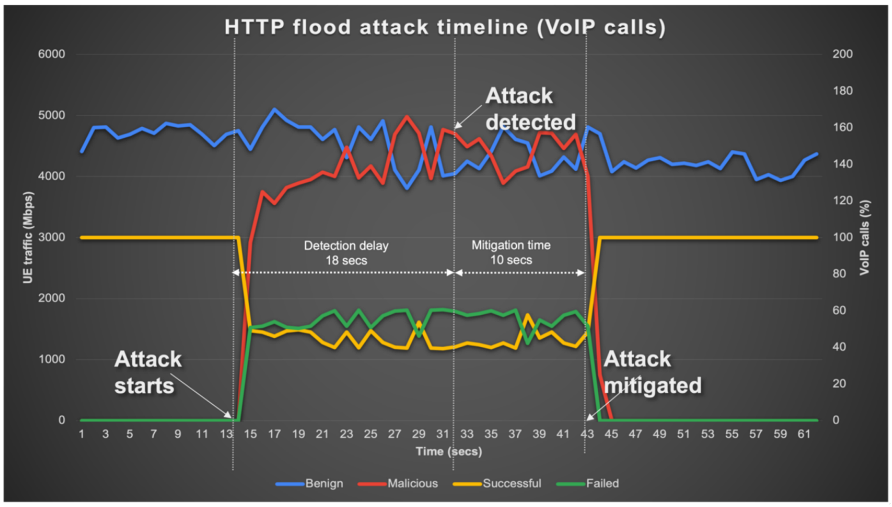


Figure 4.5: HTTP flood attack timeline (VoIP calls)

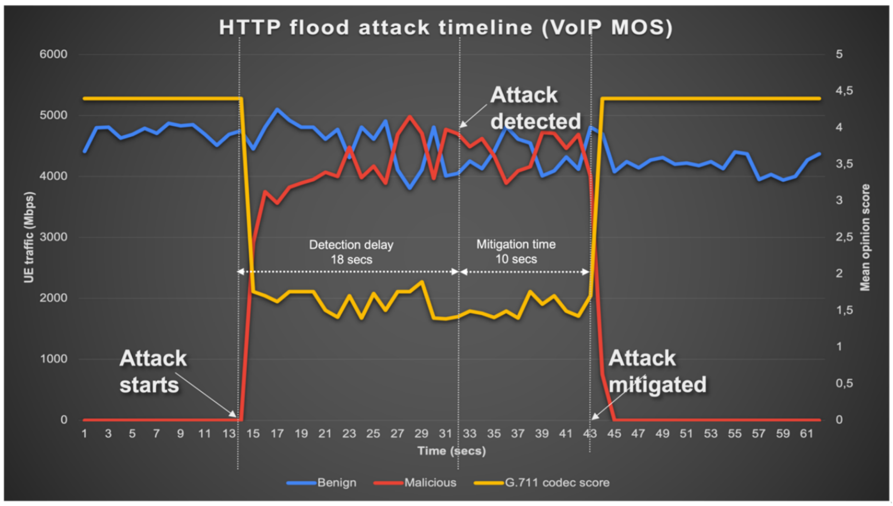


Figure 4.6: HTTP flood attack timeline (VoIP MOS)

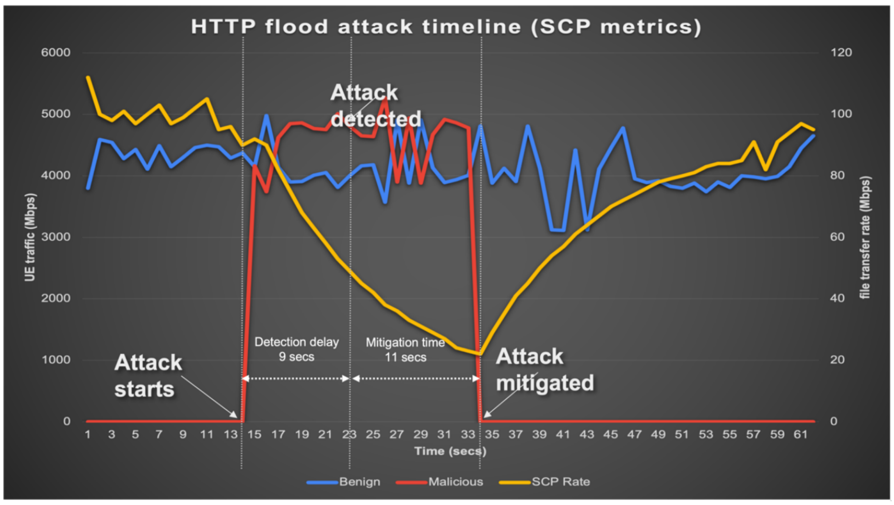


Figure 4.7: HTTP flood attack timeline (SCP metrics)

- The next three graphs show the attack timeline for the SYN flood attack. The first two graphs display the VoIP metrics on the secondary Y-axis while the third graph concerns the file transfer application. The SYN flood attack has insignificant impact on both applications. The reasons are threefold. First, the impact of a SYN flood attack on the bandwidth is minimal. Since SYN packets are very small (close to 60 bytes), even a large number of SYN packets per second cannot congest a link. The total amount of traffic generated is close to 100Mbps (as a reminder the link is 10Gbps). Secondly, the SYN flood attacks an HTTP server so it has no material impact on the SIP service which is UDP-based or the file transfer since we do not attack the used services. Finally, the targeted host has SYN-cookies implemented by default (and most Linux distributions out of the box nowadays) so the attack cannot deplete resources on the target machine. SYN cookies prevent the host of allocating file descriptors unless the UE completes the TCP handshake.

  In all scenarios, the response time was maximum 14 seconds and the false positive rate is 0%.

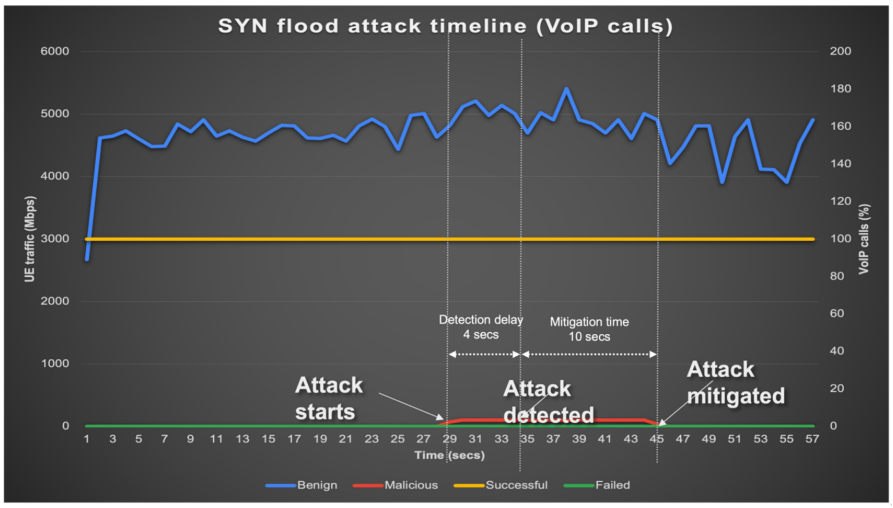


Figure 4.8: SYN flood attack timeline (VoIP calls)

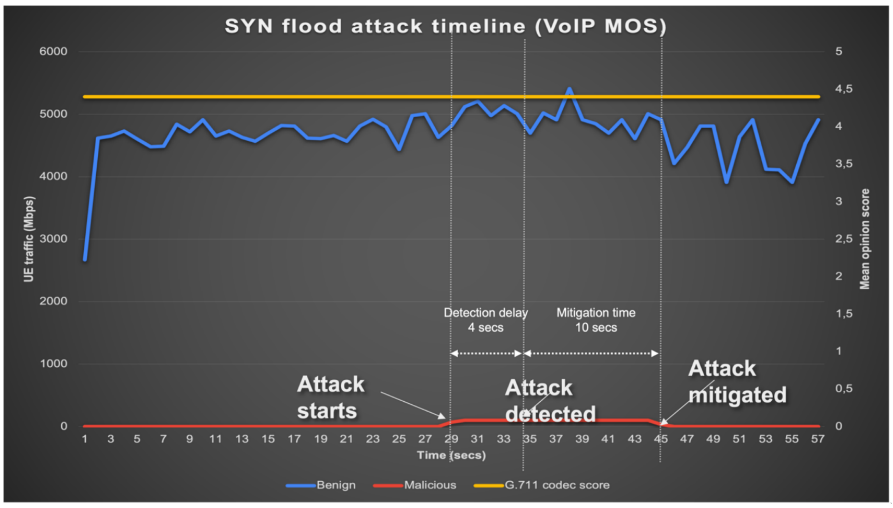


Figure 4.9: SYN flood attack timeline (VoIP MOS)

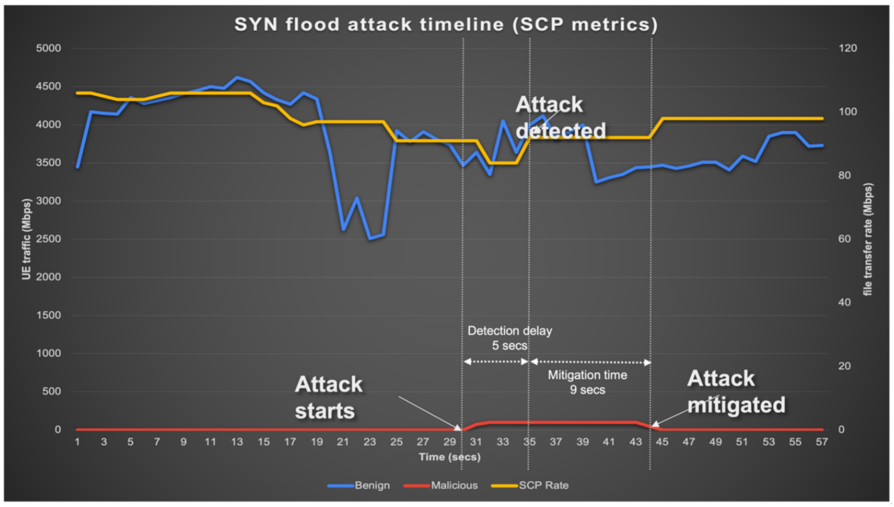


Figure 4.10: SYN flood attack timeline (SCP metrics)

Chapter 5

Related Work

This chapter discusses the related work in the field.

5.1 Real-time monitoring of SDN networks using non-invasive cloud-based logging platforms

Real-time monitoring of SDN networks using non-invasive cloud-based logging platforms [10] describes the deployment of a system architecture that integrates a cloud-based, real-time log-analysis platform with a Software Defined Networking (SDN) infrastructure. The goal is to provide network administrators with instant feedback on the health of the network by collecting log data from host machines, OpenFlow switches, and SDN controllers in a non-invasive manner.

The paper discusses a system architecture for monitoring and analyzing Software-Defined Networking (SDN) network logs in real-time or historical data, by collecting all log events produced by controllers and network components via log management agents and pushing the data to the log management and analysis platform deployed in the cloud. The log management agent automates data collection from the log files and forwards it to the log management platform, and the log files are in a structured or unstructured format. The experimental setup section discusses an SDN Network Test Manager to run a series of tests to stress the network and controller with predefined traffic profiles, packet types, and security attacks, and recommends a list of log events to be considered in monitoring the SDN network, including the SDN controller logs, switch events, and host events. The User Feedback Channel is built around notification capabilities of the log analytic platform to notify the network administrator about all potentially critical events that may affect the overall performance of the network, possible network attacks and the context of the malicious behavior, and report the overall health of the network.

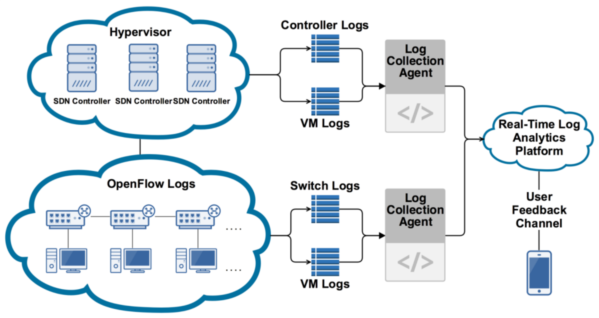


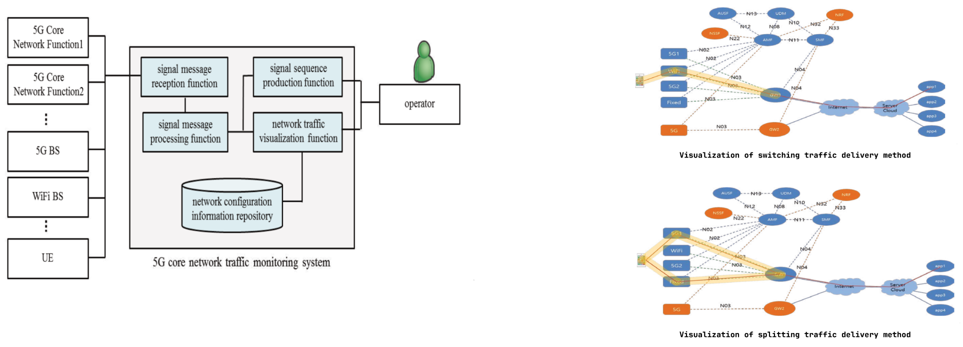
Figure 5.1: Proposed system architecture [10]

The proposed architecture uses a commercially available correlation platform to analyze log data and provide network administrators with a real-time view of the network status.

The authors did not provide details on how they performed log analysis and log parsing in the paper. However, they mentioned that the log analysis platform should be able to ingest any type of log data produced by virtualized network architecture, and the log files must be read as the events happen and streamed to the log analysis platform. They also mentioned that the log management platform should be able to understand the Key-Value-Pair format, and should include the support for Regular Expressions to extract values from unstructured log data or scan packet content if necessary. The authors recommended using TCP to transport log messages to the analysis platform to assure the reliability of the monitoring data.

As stated in this paper, previous work has highlighted the importance of network health monitoring and performance measurement in SDN networks. However, there is a lack of tools and techniques to meet the requirements of network monitoring in such a wide scope, especially in the field of SDN/NFV.

Traffic monitoring system for 5G core network[6] proposes a 5G core network traffic monitoring system that allows operators to easily identify switching and splitting traffic using monitoring messages from the 5G core network functions. The system provides a visual representation of the signal message handling process, session handling process, and traffic flow handling process for switching and splitting, enabling operators to monitor these processes more easily.


5G Core Network Function1
5G Core Network Function2
5G BS
WiFi BS
UE
signal message reception function
signal sequence production function
signal message processing function
network traffic visualization function
network configuration information repository
5G core network traffic monitoring system
operator
Visualization of switching traffic delivery method
Visualization of splitting traffic delivery method

(a) Architecture **(b)** Visualizations

Figure 5.2: 5G core network traffic monitoring system[6]

The proposed 5G core network traffic monitoring system is a complex system that includes several functions such as signal message reception, processing, and visualization. The system receives signal messages transmitted from the 5G core network function, base station (BS), and terminal user equipment (UE), and analyzes them to visualize the connection state and flow state of the network.

Chapter 6

Validation

To test and validate the metrics system, stress tests were conducted on a topology comprising two physical nodes. The central metrics server was deployed on one of the nodes. Tester programs were executed to measure the execution time overhead caused by the logging component of the system and to ensure that no logs were missed in scaled topologies. Additionally, the tests were performed using the native Golang log library, which was configured to log directly to the metrics server, using the same log format. Three sets of tests were carried out, utilizing 1000, 10000, and 100000 parallel running testers on each node.

The testers are small programs that perform 10 pings to a known server (e.g., Google, Facebook) and log a message between each ping.

Both the clog and log library successfully recorded all logs without any missing entries. However, there was a noticeable difference in execution time between the two.

Below, we present a chart illustrating the speedup of our logging system compared to the standard log library.

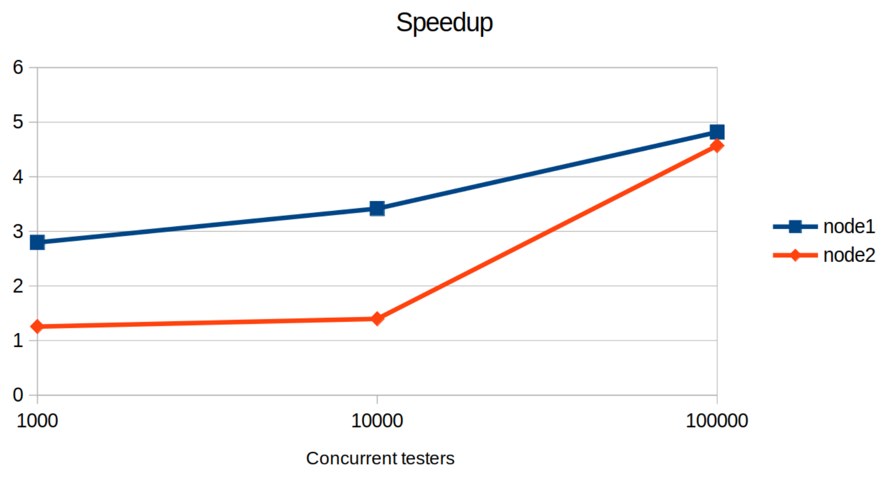


Figure 6.1: Performance Comparison of Golangs log library and clog logging library

The following two graphs depict the execution time in milliseconds on each node. It is important to note that the metrics server is deployed on node 1. Consequently, the testers running on node 1, as well as the logging agent, utilize the loopback device of the node.

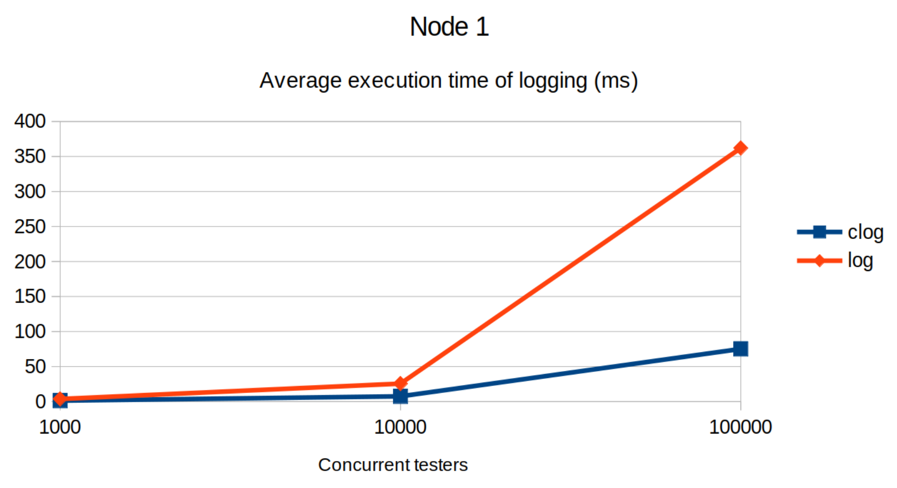


Figure 6.2: Execution Time Comparison

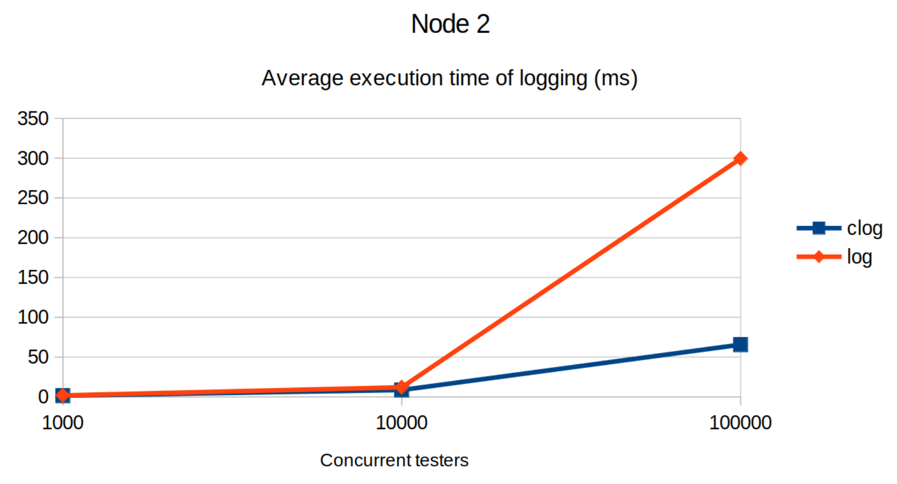


Figure 6.3: Execution Time Comparison

These findings and performance comparisons contribute to the overall assessment of the metrics system, highlighting its efficiency. The successful completion of stress tests and the analysis of execution times serve as a strong foundation for the system's reliability and scalability in handling logging and performing timestamp comparisons for measuring delay.

Chapter 7

Conclusion

In conclusion, this thesis has focused on the development of a metrics system in Golang to address the challenges associated with real-time monitoring and log aggregation in the context of 5G network security, particularly within programmable networks utilizing Kubernetes as a VNF controller. The metrics system has been successfully implemented and validated as part of the PRINCIPALS research project, providing a valuable component for network service management and security evaluation.

7.1 Future Work

Future work on the metrics system includes improving the flexibility and versatility of the parser component. Currently, the parser is tightly coupled to the attack mitigation scenarios presented in 4.5.3, with parsing parameters such as keywords and pod names hardcoded into the system. This limits the system's applicability to other use cases and scenarios.

One possible direction for future work is to develop a more dynamic and configurable parser that can adapt to different scenarios and requirements. This could involve implementing a more advanced configuration system, utilizing standardized configuration files in which the keywords in logs of pods would be determined as long as the meaning of them as for the system's state.

One promising avenue for future work is the development of a more dynamic and configurable parser that can effectively adapt to diverse logging scenarios and requirements. To this end, the implementation of an advanced configuration system that incorporates standardized configuration files is a plausible strategy. Such a system would enable greater flexibility and customization in the parser, thereby enhancing its ability to accurately parse and interpret logs generated by pods within the Kubernetes cluster.

By defining specific keywords in the logs and their corresponding meanings with respect to the system's state, this approach can facilitate the integration of new log sources, while minimizing the need for manual configuration and modification of the parser's code.

Moreover, the FSM approach used in log parsing has the potential for further abstraction and improvement. One potential enhancement is to automate the process of tuning Network Functions based on the current state of the network. This could involve the parser suggesting or even deploying TAMElets that would be tailored to address the current network state.

Bibliography

[1] (2022) Kubernetes documentation. [Online]. Available: https://kubernetes.io/docs/home/

[2] (2023) Channels in golang. [Online]. Available: https://golangdocs.com/channels-in-golang

[3] (2023) Kubernetes sidecar container examples. [Online]. Available: https://www.golinuxcloud.com/kubernetes-sidecar-container/

[4] (2023) Netflow vs. sflow: What's the difference? [Online]. Available: https://www.kentik.com/blog/netflow-vs-sflow/

[5] (2023) Prometheus overview. [Online]. Available: https://prometheus.io/docs/introduction/overview/

[6] E. Kim and Y.-i. Choi, "Traffic monitoring system for 5G core network," in *2019 Eleventh International Conference on Ubiquitous and Future Networks (ICUFN)*. Zagreb, Croatia: IEEE, Jul. 2019, pp. 671–673. [Online]. Available: https://ieeexplore.ieee.org/document/8806155/

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," vol. 38, no. 2, pp. 69–74. [Online]. Available: https://dl.acm.org/doi/10.1145/1355734.1355746

[8] T. Nakamura, "Network Functions Virtualisation – White Paper on NFV priorities for 5G," White Paper, 2017.

[9] S. Peterson, "5g mobile networks: A systems approach," p. 67.

[10] B. Siniarski, C. Olariu, P. Perry, T. Parsons, and J. Murphy, "Real-time monitoring of SDN networks using non-invasive cloud-based logging platforms," in *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. Valencia, Spain: IEEE, Sep. 2016, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/7794973/

[11] F. Z. Yousaf, M. Bredel, S. Schaller, and F. Schneider, "NFV and SDN—Key Technology Enablers for 5G Networks," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2468–2478, Nov. 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8060513/

Appendices

Appendix A

Metrics server log entry format

The metrics server accepts TCP connections and expects a series of log entries. The format of the log entries can be configured by changing the numbers in the following snippet:

Listing A.1: Log entry header format configuration. Snippet from parser.go, part of the metrics server

```go
const (
    HeaderFormatNode int   = 0
    HeaderFormatPod        = 1
    HeaderFormatCmd        = 2
    HeaderFormatTimestamp = 3
    LogEntryMsgStart       = 4
)
```

The above configuration expects log entries to have the folloing format:

```
<nodeName> <podName> <command> <timestamp> <message>
```

The timestamp should not be formated as a string. It should be the number of microseconds elapsed since January 1, 1970 UTC.