Version-Sensitive Network Traffic Classification for Kubernetes Applications

Aleksi Hirvensalo

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 31.12.2024

Thesis supervisor:

Prof. Tuomas Aura

Thesis advisors:

Tuomas Takko, D.Sc. (Tech.)

José Luis Martin Navarro, M.Sc. (Tech.)

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Aleksi Hirvensalo

Title: Version-Sensitive Network Traffic Classification for Kubernetes Applications

Date: 31.12.2024 Language: English Number of pages: 8+65

Department of Computer Science

Supervisor: Prof. Tuomas Aura

Advisors: Tuomas Takko, D.Sc. (Tech.), José Luis Martin Navarro, M.Sc. (Tech.)

Network traffic classification is a crucial area in cybersecurity and network management, enabling effective monitoring and analysis of data flows. However, existing methods often lack the granularity needed to identify subtle differences, such as those between application versions, limiting their utility in dynamic, real-life scenarios. Despite the growing importance of detailed traffic analysis, there has been no research into fine-grained classification methods to differentiate between application versions. Current techniques fall short in addressing the subtle nuances in network behavior introduced by different versions of the same application. Furthermore, there are no suitable datasets that would allow exploring version-sensitive network traffic classification.

This thesis introduces a novel, version-sensitive framework for network traffic classification. The framework is designed to detect and distinguish subtle changes between application versions by integrating machine learning and fingerprinting mechanisms. The methodology involves data collection, fingerprint generation, and classification using a custom experimental setup within a Kubernetes environment. The proposed framework demonstrates the ability to accurately classify and differentiate application versions, achieving an accuracy rate of 95.9%, even in dynamic network scenarios. Additionally, the research contributes to the field by publishing a new dataset, which provides a foundation for future studies on fine-grained traffic analysis.

This research underscores the potential for enhancing network security and management through advanced traffic classification techniques. By paving the way for more adaptive and precise systems, this work contributes a significant step forward in the development of fine-grained network traffic analysis tools.

Keywords: Network traffic classification, Kubernetes, Version detection, Fingerprinting, Helm, Machine learning

AALTO-YLIOPISTO
PERUSTIETEIDEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Aleksi Hirvensalo

Työn nimi: Versiokohtainen verkkoliikenneluokittelu Kubernetes-ympäristössä

Päivämäärä: 31.12.2024 Kieli: Englanti Sivumäärä: 8+65

Tietotekniikan laitos

Työn valvoja: Prof. Tuomas Aura
Työn ohjaajat: Tuomas Takko, D.Sc. (Tech.), José Luis Martin Navarro, M.Sc. (Tech.)

Verkkoliikenteen luokittelu on keskeinen osa kyberturvallisuutta ja verkonhallintaa, mahdollistaen tehokkaan tietoliikenteen seurannan ja analysoinnin. Nykyiset menetelmät eivät kuitenkaan kykene hienojakoiseen verkkoliikenteen luokitteluun, kuten sovellusten eri versioiden tunnistamiseen, mikä rajoittaa niiden hyödyllisyyttä dynaamisissa, todellisissa käyttötilanteissa. Huolimatta yksityiskohtaisen tietoliikenneanalyysin kasvavasta merkityksestä, hienojakoisiin luokittelumenetelmiin, jotka osaavat luokitella sovellusversioita, ei ole kohdistunut tutkimusta. Nykyiset tekniikat eivät kykene huomioimaan saman sovelluksen eri versioiden aiheuttamia hienovaraisia muutoksia. Lisäksi versiokohtaisen verkkoliikenteen luokittelua varten ei ole saatavilla tietoaineistoja.

Tämä diplomityö esittelee uuden versiokohtaisen menetelmän verkkoliikenteen luokitteluun. Menetelmä yhdistää koneoppimista ja sormenjälkimekanismeja, ja sen tavoitteena on havaita ja erottaa sovellusversioiden väliset hienovaraiset muutokset. Menetelmä sisältää tiedonkeruun, sormenjälkien luomisen ja luokittelun, joka toteutetaan räätälöidyllä kokeellisella alustalla Kubernetes-ympäristössä. Esitetty menetelmä osoittaa kykynsä luokitella ja erotella sovellusversiot tarkasti, saavuttaen jopa 95,9%:n tarkkuuden dynaamisissa verkkoskenaarioissa. Lisäksi tämä tutkimus edistää alaa julkaisemalla uuden tietoaineiston, joka tarjoaa pohjan tuleville hienojakoisen liikenneanalyysin tutkimuksille.

Tämä tutkimus korostaa hienovaraisten verkkoliikenteen luokittelutekniikoiden potentiaalia verkon turvallisuuden ja hallinnan parantamisessa. Mahdollistamalla mukautuvammat ja tarkemmat järjestelmät työ tarjoaa merkittävän edistysaskeleen hienojakoisten verkkoliikenteen analysointityökalujen kehittämisessä.

Avainsanat: Verkkoliikenneluokittelu, Kubernetes, Versio tunnistus, Sormenjälki, Helm, Koneoppiminen

Preface

I want to thank my supervisor Tuomas Aura and Professor Kimmo Kaski for making this thesis possible and facilitating everything. I also want to thank my advisors Tuomas Takko and José Luis Martin Navarro for their invaluable advice and support. Their expertise and insights have been an important part of this thesis. Finally, I want to thank Lauri Pykälä for helping me, especially in the beginning, and always finding time to answer my questions.

Otaniemi, 31.12.2024

Aleksi Hirvensalo

Contents

Symbols and abbreviations

Symbols

O	Big O notation

Abbreviations

AI	artificial intelligence
API	application programming interface
ASCII	American Standard Code for Information Interchange
BoF	bag of flows
CI	continuous integration
CNCF	Cloud Native Computing Foundation
CNN	convolutional neural network
CQL	Cassandra Query Language
CRM	customer relationship management
CSG	common substring graph
CSV	comma-separated values
DDoS	distributed denial of service
DNS	domain name system
DPI	deep packet inspection
ENIP	EtherNet/IP
ERP	enterprise resource planning
ESP	encapsulating security protocol
FN	false negative
FTP	file transfer protocol
HTTP	hypertext transfer protocol
HTTPS	hypertext transfer protocol secure
IANA	Internet Assigned Numbers Authority
ICMP	internet control message protocol
IMAP	internet message access protocol
IDS	intrusion detection system
IP	internet protocol
IRC	internet relay chat
IT	information technology
IoT	internet of things
JSON	JavaScript object notation
k8s	Kubernetes
ML	machine learning
mTLS	mutual transport layer security
NN	neural network
OSI	Open Systems Interconnection
P2P	peer-to-peer
PCAP	packet capture

PGSQL	PostgreSQL
PMPROXY	performance co-pilot protocol proxy
POP	post office protocol
QoS	quality of service
SCM	supply chain management
SMPP	short message peer-to-peer
SMTP	secure mail transfer protocol
SPI	stochastic packet inspection
SQL	structured query language
SSH	secure shell
SemVer	semantic version
TCP	transmission control protocol
TLS	transport layer security
TP	true positive
TTL	time to live
UDP	user datagram protocol
VM	virtual machine
VPN	virtual private network
WWW	world wide web
YAML	yet another markup language

1 Introduction

In the past few decades, the digital landscape has been transformed by numerous rapid technological advancements, leading to an increased reliance on networked systems for both personal and organizational needs. This transformation has been paralleled by the expansion and growing complexity of network traffic, which now encompasses a wide variety of applications, services, and protocols. As data flows increase in volume and diversity, the necessity for efficient and accurate traffic classification becomes increasingly important. Network traffic classification, which involves identifying and categorizing data packets based on their characteristics, has critical applications in areas such as cybersecurity, network optimization, and quality of service (QoS) assurance. However, the dynamic and evolving nature of modern network traffic poses significant challenges to traditional classification approaches.

Traffic classification techniques have evolved considerably from early port-based methods, which relied on standardized ports for protocol identification, to more sophisticated payload-based and machine learning-based methods that utilize advanced statistical models and algorithms [61]. Despite the progress, several issues remain unaddressed. First, the existing methods only classify traffic on a coarse granularity. Second, existing datasets are relatively scarce and often do not resemble real-world scenarios. In addition, many existing datasets are old and becoming quickly outdated. Furthermore, the ability to detect subtle differences between application versions, an essential feature for security monitoring and application performance assessment, remains a largely unexplored domain in the field of network traffic classification.

This thesis explores a novel approach to network traffic classification that aims to address these challenges by developing a version-sensitive framework that can capture the nuances between version increments. Effectively this enables the framework to differentiate between different application versions. The proposed framework leverages machine learning (ML), fingerprinting, and signature-based analysis to detect small but significant changes in network traffic patterns that correspond to different application versions. By doing so, it contributes to a more nuanced and adaptable classification system.

The primary research question addressed in this thesis is: **How can fingerprinting and machine learning techniques be employed to distinguish between different versions of networked applications based on observed network traffic?** To answer this question, the study focuses on three objectives: developing a robust data collection process, generating distinct fingerprints for each application version, and applying classification algorithms to differentiate between these versions accurately. These objectives provide a structured approach to evaluating version-sensitive traffic classification's effectiveness.

The context of this thesis is Kubernetes, an open-source platform for automating deployment, scaling, and managing containerized applications [6]. Kubernetes has become widely adopted in cloud computing environments due to its flexibility and scalability, making it a suitable choice for experimenting with dynamic application versions in a controlled setting. The framework developed for this thesis captures network traffic from multiple instances of the same application version running

in Kubernetes and then utilizes a fingerprinting mechanism and machine learning algorithms to detect and classify version-specific traffic patterns.

Through this research, this thesis contributes to the existing body of knowledge on network traffic classification by introducing a version-sensitive approach that can improve accuracy in identifying changes within applications. This is particularly valuable in cybersecurity, where even small, untracked changes in application versions can introduce vulnerabilities. The findings from this research may help the traffic classification field to create more application- and environment-agnostic frameworks. By addressing these issues, the proposed approach enhances the adaptability and precision of traffic classification, paving the way for improved network management and security monitoring solutions. Furthermore, the thesis contributes to the network traffic classification field by providing a new dataset.

In summary, this thesis aims to bridge a gap in network traffic classification by providing a comprehensive framework for detecting application versions based on network traffic analysis. The following chapters present the background and current landscape of traffic classification techniques, detail the methodology and framework developed for this study, and discuss the experimental findings and their implications for future research.

2 Background

The background section introduces the necessary concepts to follow this thesis.

2.1 Machine learning

Machine learning is a subset of artificial intelligence (AI) that focuses on developing systems capable of learning from data, recognizing patterns, and making decisions with minimal human intervention [33]. Instead of being explicitly programmed with rules, machine learning models are trained on data to find patterns and to make decisions based on those patterns. Machine learning models can improve over time by being trained on new data.

Machine learning requires a dataset to train machine learning algorithms. A dataset is a collection of data points which can be, for example, numbers, images, or text. The data points can either be labeled or not. A label is a tag that provides information about the class of a datapoint. For example, a picture of a dog can be labeled as a dog or an animal.

A suitable machine learning paradigm needs to be chosen based on the type of dataset and specific goals. Three common paradigms are supervised learning, unsupervised learning, and semi-supervised learning [33].

Supervised learning uses labeled data to make accurate predictions [13]. Its primary objective is to establish a relationship between inputs and outputs, enabling the model to accurately predict or classify new, unseen data. During training, the algorithm minimizes the discrepancy between predicted outputs and actual labels by adjusting its parameters. There exists a great number of training algorithms that are suited to implement supervised machine learning, such as random forest, decision trees, and linear regression [62].

Before a supervised training algorithm can be used, the dataset needs to be split into training and test sets [62]. A training algorithm will take the training dataset as input and then output a model. The model is then evaluated against the test dataset. If the model is deemed accurate enough for the given task, it can then be employed to predict results on unseen data. If not, the training algorithm can be rerun with different datasets, or the algorithm can be tweaked or switched completely. This process can then be repeated until a sufficient model is found.

On the one hand, unsupervised learning uses unlabeled data to discover hidden patterns without human intervention [33]. This approach is particularly effective for tasks like exploratory data analysis and pattern recognition due to its ability to detect similarities and differences in data. Additionally, unsupervised learning is used to reduce the number of features in a model through dimensionality reduction.

On the other hand, semi-supervised learning provides a balance between supervised and unsupervised learning approaches [33]. It works by using a small, labeled dataset to assist the classification and feature extraction process while leveraging a larger, unlabeled dataset. This method addresses the challenge of limited labeled data, which can be insufficient for traditional supervised learning. Furthermore, it is useful when labeling large amounts of data is too expensive or time-consuming.

Deep learning is a subset of machine learning that can implement the three aforementioned machine learning paradigms: supervised, unsupervised, and semi-supervised learning [13]. Deep learning is based on neural networks (NNs). NNs consist of layers of nodes, which include an input layer, one or more hidden layers, and an output layer [13]. Each node functions as an artificial neuron and connects to the next node. Each node has an associated weight and a threshold value. When the output of a node exceeds its threshold, the node becomes activated and transmits its data to the next layer of the network [13]. Conversely, if the output is below the threshold, no data is sent to the following layer. Deep learning is characterized as "deep" if it uses NNs with three or more hidden layers, typically it uses hundreds or thousands of layers [32].

The following subsections cover popular machine learning techniques and their implementations.

2.1.1 Ensemble learning

Ensemble learning is a machine learning technique that combines two or more models to produce more accurate results [43]. It addresses a common machine learning problem called **bias-variance tradeoff**. **Bias** describes how well a machine learning algorithm is able to find relevant relations between features and targets. Higher bias means that the algorithm has not been sufficient at finding those relations. This leads to underfitting. **Variance** describes the training algorithm's resistance to fluctuations in the training set. Low resistance to fluctuations means that small changes in the training data will change the model drastically. Therefore, the model cannot perform well against unseen data which leads to a situation where the model performs poorly with the test data.

The lower the bias and variance, the better the performance of the model. A singular training algorithm can lead to different models with their own bias and variance values. By combining multiple models, research has shown that ensemble learning can reduce the overall error rate compared to singular models [53].

There are multiple techniques to implement ensemble learning in practice. One of the most common ones is called **bagging**. Using bagging, a singular training algorithm receives multiple resampled sets of the training data [22]. This process is referred to as bootstrap resampling. Bootstrap resampling takes n samples of the training data then it duplicates some data points and excludes some data points completely. Each resampled dataset is given to the training algorithm which will produce different models with each having its own bias and variance. Each model then predicts a result, and a majority vote is used to determine the final result.

2.1.2 Decision tree

A decision tree is a machine learning algorithm used for both classification and regression tasks [54]. It models decisions and their potential outcomes in a tree-like structure, with each internal node representing a feature test, each branch signifying the test's result, and each leaf node indicating a class label or continuous value. Figure 1 describes a simple decision tree process.

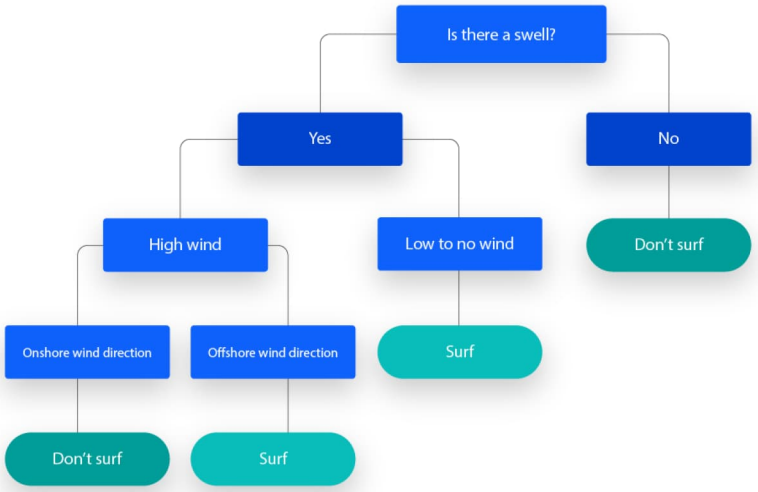

Figure 1: Example of a simple decision tree [31]

The decision tree learning algorithm works by performing a greedy search to identify how to optimally split the tree [31]. The optimal split can be measured with, for example, information gain or Gini impurity. The goal is to create pure groups where each branch or leaf node contains mostly or entirely data points that have the same label. However, when the tree size grows, this can lead to data fragmentation where some branches or leaf nodes contain only a small number of data points. This in turn leads to overfitting. Pruning is used to reduce overfitting by removing branches that do not improve the model's accuracy.

A single decision tree can be subject to high bias or variance [57]. **Random forest** is a training algorithm that addresses these problems by combining multiple decision trees with ensemble learning. Random forest extends the bagging method by also sampling features, resulting in a forest of decision trees where the trees have low correlation [14]. Random forest can be used either for classifying or regression tasks. In classification tasks, it will take the majority vote and in regression tasks, individual results are averaged.

2.2 Containers

Containers are lightweight, portable, and self-contained environments that bundle an application and all its dependencies, such as libraries, binaries, and configuration files, needed to run that application [39]. Containers allow consistent application performance across various computing environments by isolating the application from the host system. This makes them highly valuable for deployment, scalability, and maintaining consistency across development, testing, and production environments.

Compared to virtual machines, containers can be viewed as virtualized applications whereas virtual machines contain the operating system as well [39]. Multiple containers can be run within the same (virtual) machine, allowing the containers to share the same operating system. Thus, making it a more lightweight virtualization

technique compared to virtual machines. Figure 2 illustrates the difference between containers and virtual machines.

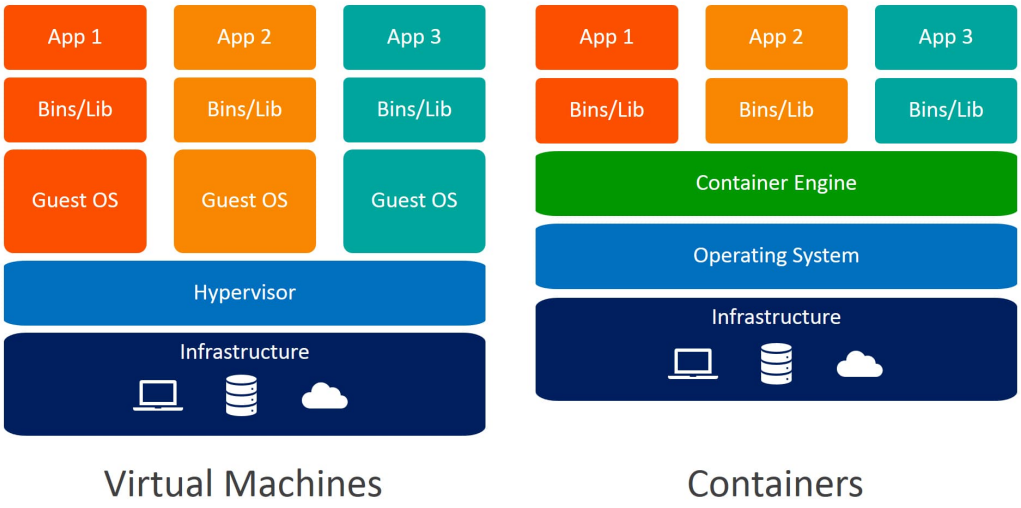

Figure 2: Container and virtual machine comparison [37]

2.3 Kubernetes

Kubernetes, or k8s, is an open-source container orchestration tool [6]. It is common to deploy applications using containers. However, several tasks must be addressed, including deploying and upgrading containers, ensuring uninterrupted service, and maintaining scalability. Kubernetes has been developed to handle these tasks automatically. Some of the services that Kubernetes provides are service discovery, load balancing, storage orchestration, automated rollouts or rollbacks, self-healing, secret management, batch executioning, and horizontal scaling.

2.3.1 The architecture

The main component of Kubernetes is a cluster. It consists of a control plane plus a set of worker machines, or nodes, that run containerized applications [3]. **The control plane** is responsible for making global decisions about the cluster, for example, starting up a new Pod. The main components in the control plane are kube-apiserver, kube-controller-manager, kube-scheduler, and etcd. Kube-apiserver exposes the Kubernetes API that lets one query and manipulate the state of objects in Kubernetes. Etcd is a key value store used as a backing store for all cluster data. Kube-scheduler assigns newly created Pods to Nodes. Kube-controller-manager manages controller processes. A controller process tracks at least one Kubernetes resource type. The resource has a spec field that describes the desired state of the resource. Controller might try to maintain and reach the desired state but often it sends messages to the API server if the resource is not in its desired state. A **Pod** is

a set of one or more containers within a Kubernetes cluster with shared storage and networking [8].

Nodes may be virtual or physical machines, depending on the cluster [3]. A node hosts Pods, which in turn runs containers responsible for executing the actual workload of the application. Each Node is managed by the control plane and includes the services necessary to run Pods. The main Node components are a kubelet and a container runtime. The kubelet operates on each Node within a cluster, ensuring that containers are active within a Pod. It processes a set of specifications and ensures the containers described in those specs are functioning correctly and remain healthy. The container runtime handles the execution and lifecycle management of containers.

Figure 3 provides an overview of the core components in Kubernetes and how they interact with each other.

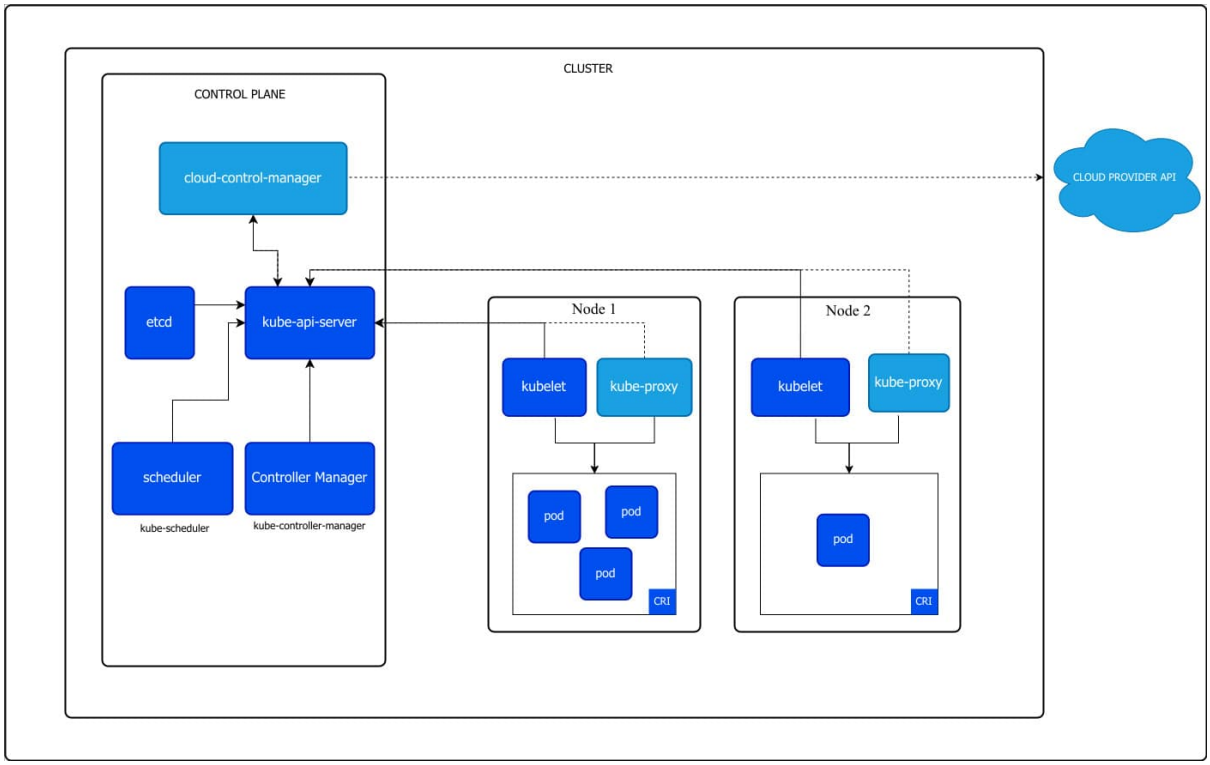

Figure 3: Kubernetes architecture [3]

2.3.2 The network model

The Kubernetes network model consists of several components [10]. Each Pod is assigned a unique IP address that is accessible cluster-wide. A Pod has its own private network namespace, shared by all containers within the same Pod, enabling them to communicate with each other over localhost. Pod network, or cluster network, facilitates communication between all Pods, ensuring they can interact regardless of whether they are located on the same node or across different nodes. It also allows agents on a node, such as the kubelet, to communicate with all Pods on that node.

The **Service API** provides a stable, long-lived IP address for an application implemented by one or more backend Pods, with the flexibility for the individual Pods making up the Service to change over time [10]. [10]. A **Service** is an abstraction in Kubernetes that defines a logical set of Pods and a policy for accessing them [9]. By creating a Service, an application running within a Kubernetes cluster is exposed through a single, outward-facing endpoint, even as the underlying Pods change dynamically due to scaling or failure. The Service API accomplishes this by monitoring Service and EndpointSlice objects to track the Pods associated with each Service, and it configures the data plane to route traffic to its backends using operating system or cloud provider APIs to intercept or modify packets as needed. To expose these Services externally, the Gateway API (a successor to Ingress) acts as an interface between the cluster and external clients. Additionally, network traffic can be controlled with a built-in Kubernetes API called NetworkPolicy, which allows for granular control over traffic flow within the cluster.

2.3.3 Network traffic

Within a Kubernetes cluster, a variety of network traffic can be observed, depending on the application's configuration and the environment. Some of the key sources include:

- **Service Discovery and DNS Traffic:** To be able to communicate with a Service, its IP address needs to be discovered. There are two Service discovery methods in Kubernetes: environment variables and DNS [9]. Environment variables simply store the address of a Service as a variable that is accessible to other services. DNS services are implemented with cluster-aware DNS servers, such as CoreDNS. They operate by watching the Kubernetes API for new Services and creating a new DNS record for each one.

  If a Pod needs to communicate with some other Service or an external resource, it needs to send a DNS query to be able to discover the service [9]. The discovery process can be observed in the DNS query packets and DNS response packets.

- **Service and Pod-to-Pod Communication:** Service and Pod-to-Pod refer to communication that happens between Pods in a cluster [10]. The communication occurs either directly between Pods or through Services. Communication can happen with a wide variety of protocols, such as HTTP, HTTPS, and WebSocket.

  Pod-to-Pod communication is possible by making the Pods aware of each other's IPs [9]. However, if a Pod dies and is restarted, it receives a different IP address. All the other Pods need to then be made aware of the new IP. Service-to-Service communication is more robust as it negates this issue by abstracting the Pods behind a Service object. The Service object has a unique IP address which remains the same throughout its lifecycle.

- **Health Checks and Probes:** Health checks and probes are an important tool for Kubernetes to determine the health, liveness, and readiness of the containers [5]. Kubernetes uses this information to manage the container states correctly. The probes and their responses are defined by application developers and therefore they can vary between applications. The probes can be implemented in three ways: HTTP probe, TCP probe, and command probe. HTTP probe is implemented by making an HTTP request to a specific endpoint. The probe is successful if the HTTP response code is between 200 and 400. The TCP probe opens a TCP connection to the container on a specific port. The probe is successful if the connection is also successful. The command probe runs a specific script inside the container. If the command exits with status 0, the container is considered healthy. Note that the command probe is the only probe that does not produce network traffic directly.

  Kubernetes offers three types of probes for health checks: liveness, readiness, and startup probes [5]. The purpose of the liveness probe is to check if a container is still running. If the liveness probe fails, Kubernetes assumes the container is in a bad state and will try to restart it. The readiness probe determines if the container is ready to serve traffic. If the readiness probe fails, Kubernetes will remove the container from the Service's load balancer until it passes. The startup probe is used to check if a container has started. It prevents Kubernetes from prematurely killing containers that have slow startup times.

- **Kubernetes API and Control Traffic:** Communication with the application involves HTTPS traffic to and from the Kubernetes API server, which enables interaction with cluster resources [10]. Additionally, traffic flows between the kubelet and individual Pods to perform health checks, such as liveness and readiness probes, ensuring application stability. Packet routing and load balancing within the cluster are managed by kube-proxy, which directs traffic efficiently between services and endpoints.

- **Ingress and Egress Traffic:** This covers external access to the application and outgoing traffic from Pods to external resources, such as APIs or databases [4]. Clients outside the cluster will access the application through an Ingress controller (e.g., Nginx, Traefik).

- **Load Balancer Traffic:** The purpose of a LoadBalancer is to reduce the stress on individual pods by spreading out the traffic to multiple Pods. Incoming traffic is routed through a load balancer when using the LoadBalancer service type, where it is then forwarded to specific node ports that ultimately route traffic to the appropriate application pods [10].

- **Telemetry and Monitoring Traffic:** Metrics are collected via HTTP requests from Prometheus to the `/metrics` endpoints of application pods while logging traffic is forwarded from Fluentd or other logging agents to centralized logging services [48].

2.3.4 Pod lifecycle and PodStatus

The Pod lifecycle consists of various phases that a Pod transitions through, from its creation to termination [7]. Each Pod has a status field which contains a **PodStatus** object which in turn holds the Pod phase value. When a Pod is first created, it enters the "Pending" phase. In this phase, the Pod has been accepted by the Kubernetes system, but one or more of the containers within the Pod has not been started. The "Pending" phase occurs when the scheduler is still looking for a node to run the Pod on or there are not enough resources. Once the container(s) inside the Pod is successfully created and started, the Pod moves into the "Running" phase. This transition occurs when at least one container is running and it stays in this phase while all the containers are operating as expected, i.e., the probes are successful. Pod can turn into "Succeeded" phase if it has finished executing. This is common with batch jobs and tasks that have a defined end. If a Pod is intentionally removed or deleted, it will enter a "Terminating" phase.

Pods also have phases to indicate failure [7]. A Pod can enter a "Failed" phase which indicates that the Pod was unable to complete its workload successfully. If the control plane cannot determine the Pod's phase it will enter a "Unknown" phase. Figure 4 illustrates the Pod phases and their transitions.

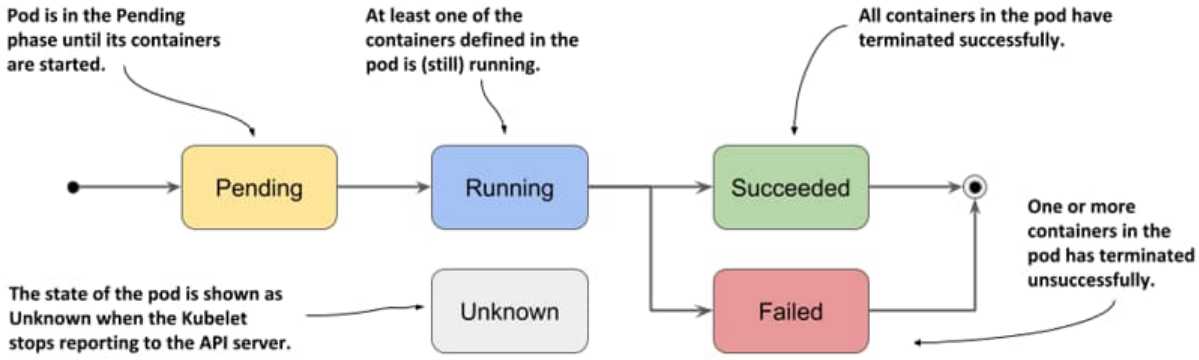

Figure 4: Pod phases [41]

Pods also have conditions that closely relate to the lifecycle phases but represent a more specific status regarding the health and availability of the containers inside the Pod [7]. The Pod starts with the "PodScheduled" condition which indicates that the Pod has been scheduled to a node. Once all containers in the Pod are ready, the condition transitions into "ContainersReady" condition. When the Pod is ready to accept traffic and handle requests, the condition is set to "Ready". A special "Initialized" condition is reserved for init containers to indicate they have been successfully completed. Init containers are special containers that run before the main containers in a Pod. Figure 5 illustrates the possible Pod conditions and how the conditions are transitioned.

Inside a Pod, container states are used to track individual containers [7]. Before a container is running, it is assigned a "Waiting" state. The "Waiting" state indicates there are some operations that need to be completed before start up. "Running"

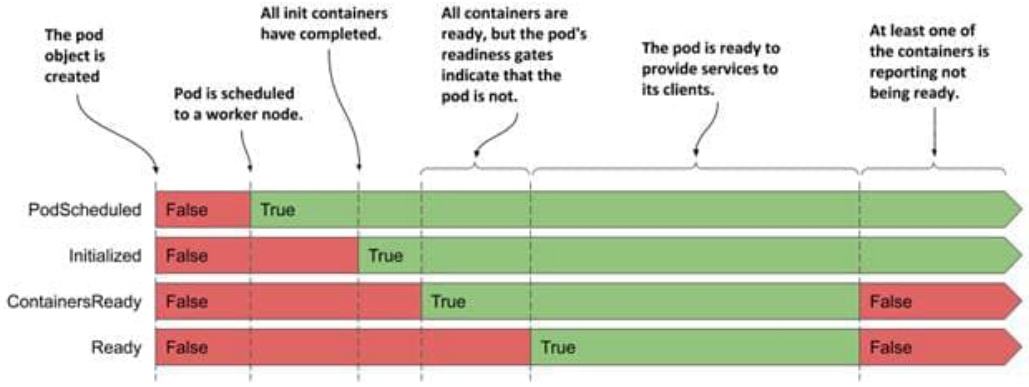

Figure 5: Pod conditions [41]

state indicates that the container is executing without issues. "Terminated" state occurs when a container has either completed successfully or failed for some reason.

2.4 Helm

Helm is a package manager for Kubernetes applications [2]. It allows users to easily install, deploy, and upgrade Kubernetes clusters via Helm charts.

A Helm chart is a complete package that organizes and encapsulates all the files and configuration needed to define, install, and manage a Kubernetes application [2]. Helm charts include templates for each Kubernetes resource, a *values.yaml* file for configuration defaults, and metadata in the *Chart.yaml* file that describes the chart's version, name, and dependencies. By bundling templates, configuration, and documentation together, Helm charts simplify application deployment and version control, making it easy to deploy consistent, configurable applications across environments, and share them with other users or teams.

Helm templates are YAML configuration files in Helm charts used to define the Kubernetes resources that Helm will deploy, manage, and maintain on a Kubernetes cluster [2]. Helm templates utilize the Go templating language to enable dynamic configuration, allowing users to create reusable and configurable templates that adapt based on different deployment environments or configurations.

Artifact Hub is a web-based platform and open-source project designed to help users find, share, and manage Kubernetes packages and other cloud-native artefacts from a variety of different sources [55]. It serves as a centralized repository that hosts, indexes and organizes a wide range of cloud-native artefacts, such as Helm charts, Kubernetes operators, and Falco rules.

2.5 Semantic versioning

Semantic versioning, or SemVer, is a widely accepted standard for versioning in software development [47]. It follows a three-part numeric format, where the first number indicates the major version, the second indicates the minor version and the third indicates the patch version. The numbers are separated by dots. For example,

given Helm chart version 23.1.0 for WordPress, 23 is the major version, 1 is the minor version, and 0 is the patch version (in this case no patch has been made for that minor version). Figure 6 illustrates the SemVer version format.

```
MAJOR.MINOR.PATCH -> 23.1.0

Major version = 23
Minor version = 1
Patch version = 0
```

Figure 6: Semantic versioning example

The numbers are semantic, i.e., they convey meaning [47]. Incrementing major versions indicates that there are breaking changes. These changes can include significant feature additions or architectural overhauls that require developers or users to adjust their usage of the software.

Incrementing a minor version means the changes are backward-compatible [47]. This means users should be able to upgrade without breaking their existing setup.

Incrementing patch version indicates that backward-compatible bug fixes have been made [47]. This is typically used for small fixes, security patches, or performance improvements.

2.6 Anomaly detection

Anomaly detection is a method used to identify patterns that deviate from expected behavior [17]. These unusual patterns, often called anomalies, outliers, exceptions, or aberrations, are critical to recognize, especially in applications where deviations might signal fraud, threats, or system failures. The term “anomalies” is most commonly used to describe these nonconforming patterns, and the practice of anomaly detection dates back to 1887 when it was first studied by statisticians in an effort to understand and isolate irregular data points [18]. Since its origins, anomaly detection has become essential in various fields such as credit card fraud prevention, cybersecurity, fault detection in critical systems, and military surveillance [17].

In anomaly detection, defining a baseline of typical behavior is crucial. This baseline is based on recorded data that represents the expected operations within a given context. Anomalies, then, are data points or patterns that do not align with this established baseline. In the context of network security, for instance, anomaly detection often involves monitoring network traffic to identify abnormal packets that could signal potential security threats. By observing traffic across multiple application setups, a baseline of normal network activity is created, and any packet deviating significantly from this model is flagged as an anomaly.

With the rapid growth of cyberthreats in recent years, anomaly detection has gained even greater importance, especially for safeguarding critical infrastructure

[35]. Attacks targeting sectors like IT, finance, healthcare, and energy have become more frequent and sophisticated, with network-based attacks like Distributed Denial-of-Service (DDoS) attacks representing a significant threat. Security measures such as encryption, firewalls, and authentication protocols can help mitigate some risks, but these defenses alone cannot address all potential threats. Intrusion Detection Systems (IDSs) have therefore become key tools in identifying and responding to cyber-attacks.

IDSs are commonly divided into signature-based and anomaly-based systems [35]. Signature-based IDSs operate by identifying known patterns or sequences associated with specific attacks. While effective at detecting well-known attacks, these systems are limited because they cannot recognize new or previously unseen threats, often called zero-day attacks. Anomaly-based IDSs, on the other hand, detect deviations from an established baseline of normal activity. This approach enables them to recognize new or unknown threats but poses the challenge of defining clear boundaries between normal and abnormal behavior, which can lead to false positives.

The unpredictable nature of today's cyber-attacks underscores the need for ongoing research and development in anomaly detection [35]. By improving techniques for detecting deviations within network traffic, security experts can better protect sensitive systems from a wide range of cyber threats. Researchers are actively exploring new methods to enhance the precision and reliability of IDSs, seeking to minimize false positives while effectively identifying real threats. These efforts are crucial in an era where the security of digital networks and data is a growing concern across industries.

2.7 Packet capture (PCAP)

PCAP is a file format used to capture and store network traffic [26]. The PCAP format allows to capture and analyze network packets (the basic units of data exchanged over a network).

A packet is a small unit of data sent over a network. When data is transmitted, such as a file transfer or a database query, it is divided into smaller chunks called packets, which are then sent individually over the network and reassembled at the destination.

PCAP files are used to store network packets for further analysis. The packets captured in a PCAP file can include a variety of information, like the source and destination IP addresses, the protocol used (TCP, UDP, etc.), port numbers, and the actual data being transmitted [26]. A PCAP file typically contains two sections, header information and packet data. Header information includes metadata like the time the packet was captured, the length of the packet, and other details. Packet data consists of the actual bytes that were transmitted over the network.

While the PCAP file format is standardized and consistent across operating systems, the APIs used to capture packets differ. On Linux, packet capture is implemented using *libpcap*, which was originally developed by the *tcpdump* team at Lawrence Berkeley National Laboratory [24].

3 Related work

This section introduces the related work in the network traffic classification field. Common methods are described, and their strengths and limitations are identified. This lays the foundation for the decisions made in developing the framework introduced in this thesis. Finally, version classification is introduced as a new research direction in the context of network traffic classification.

3.1 Network traffic classification

Network traffic classification is identifying and categorizing traffic into different classes or categories based on predefined criteria such as protocols, applications, services, or behavior [56]. Network traffic classification is a key technique used in networking to understand the types of data flows, monitor network usage, optimize performance, and enhance security.

Network traffic classification has received a lot of attention in the past 30 years. In this thesis, the focus is on the more recent and general traffic classification surveys, i.e., surveys that give a general overview of the traffic classification field and do not focus solely on a single method or approach. The surveys in question are by Valenti et al. [56], Zhao et al. [61], Sheikh et al. [52] and Azab et al. [11].

Zhao et al. [61] split traffic classification into four distinct levels of granularity, each offering varying degrees of detail. The first level is the most granular level, where traffic from the same application is further categorized based on the specific service provided, such as downloading or chatting. On the second level, traffic is classified according to specific applications, providing detailed results that include application names like Google, YouTube, and Facebook. The third level groups traffic based on the protocols used, such as SMTP, POP, and IMAP. Since multiple applications can use the same protocol, this level is less specific than the application-based classification. The fourth level of classification is the broadest level, where traffic is categorized by application type or function. Applications with similar purposes are grouped together without identifying individual applications by name. For example, all email-related protocols could be classified under a "Mail" category. Table 2 lists the granularity levels as identified by Zhao et al. [61].

The traffic classification procedure starts by collecting data from a network environment. Data can be collected from four levels, packet level, flow level, connection level, and host level [61]. Packet-level features can include packet header, payload, and activity information such as Time To Live (TTL) and the flags of the packet header such as TCP FIN and SYN. Flow-level data is an aggregation of packets that share the same attributes. Packets are commonly aggregated together if they share the same source IP, destination IP, source port, destination port, and protocol. Flow level data also often includes aggregating the packets within a certain time interval. Connection level data contains information between two hosts. It can include the number of connections, connection level, and connection duration. Host-level data is collected from the local host. It can include host activities, host changes, and resource consumption. Based on the survey by Zhao et al. [61], most existing traffic

Table 2: Levels of Traffic Classification Granularity, based on survey by Zhao et al. [61]

Level	Granularity	Description
1	Most Granular	Traffic is categorized based on specific services provided within an application (e.g., downloading, chatting).
2	Application-Specific	Traffic is classified according to specific applications, identifying names like Google, YouTube, or Facebook.
3	Protocol-Based	Traffic is grouped by protocol (e.g., SMTP, POP, IMAP). This level is less specific since multiple applications may use the same protocol.
4	Broadest	Traffic is categorized by application type or function, grouping similar-purpose applications without identifying individual application names (e.g., "Mail" for all email-related protocols).

classification solutions are built on packet and flow level data. Few solutions use connection-level data. Host-level data is rarely seen.

Even though many studies focus on a single level of data, it is important to note that these levels can be mixed in the data collection process [29]. Combining features from multiple levels of data can offer better accuracy [61].

The second step involves choosing the features. This process plays a vital role in determining the classifier's performance, speed, and accuracy [52] as it identifies and retains the most significant and relevant features from the raw data. The goal is to remove irrelevant, redundant, or noisy features that do not significantly contribute to the traffic classification model. This can help reduce computational complexity and improve model performance by focusing on the features that provide the most meaningful information [61]. This step is optional, especially if using deep learning methods [11].

The third step involves extracting the features. Feature extraction, also referred to as feature reduction, is the process of transforming raw data into a set of new features that are more informative for the classification task [46]. Unlike feature selection, where one chooses from existing features, in feature extraction, we derive new features from the original data, often by combining or transforming them into more meaningful representations. Feature extraction often involves techniques like dimensionality reduction to simplify the data without losing important information. This can be crucial for handling the high dimensionality of network traffic data. Note that the feature extraction process is optional.

The fourth step is the decision process. It involves using the data with the selected features and performing pattern matching, statistical analysis, or machine learning to classify the data [52].

Finally, the decision process needs to be validated. The validation process evaluates the results of prior traffic classifications to assess the accuracy of the classification algorithms [52]. This is done by comparing the values obtained from the original dataset with the experimental outcomes. Comparing the values allows users to measure the accuracy of the traffic classification technique. However, gathering the full range of categories within the original dataset poses a significant challenge. A common method for labeling traffic is using ground truth collection, which involves port collection and deep packet inspection (DPI) tools. Despite their popularity, these methods may produce unreliable information and require substantial computational resources to process traffic labels.

Figure 7 illustrates the traffic classification process steps and their relationship with each other.

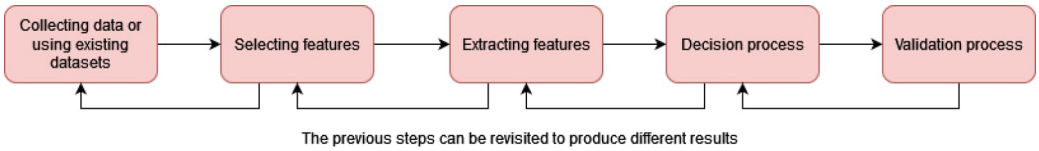

Figure 7: Overview of the traffic classification process

Four common traffic classification techniques appear in the literature: port-based classification, payload-based classification, statistics-based classification, and behavior-based classification [11, 52, 61]. Furthermore, Zhao et al. [61] covers a fifth technique called correlation-based classification. This thesis also explores deep learning techniques as a stand-alone, sixth technique inspired by [11]. Other surveys either do not cover deep learning or cover it as part of some other technique such as statistics-based classification.

3.1.1 Port-based classification

Port-based classification uses the network packet's header information to obtain the source and destination port values [56]. Port values can be used to detect the used network protocol as Internet Assigned Numbers Authority (IANA) has allocated standard port numbers for different services or protocols [30]. Port-based classification's advantages are its low resource requirements as it only considers singular values from each packet and does not require further processing of payloads [11]. However, its disadvantages are masquerading and many applications deploying on non-standard or dynamic ports [11]. Masquerading refers to a technique where standard ports are used to deliver other protocol traffic, such as sending malware traffic over HTTP port 80. Deploying on a non-standard or dynamic port means that an observer cannot determine what protocol was used based on the port number alone. Due to these limitations, port-based classification has only been relevant in the early days of the internet [56].

Port-based classification can provide sufficient accuracy when it comes to protocol identification. Moore and Papagiannaki [44] conducted an empirical study where they observed approximately 70% accuracy in identifying the protocol over all packets.

They concluded that port-based identification could not identify most bulk data transfer packets. In their study, this issue was due to File Transfer Protocol (FTP). Even though FTP uses standard ports, in passive mode it can assign random port numbers from the server to facilitate the actual file transfer. Moore and Papagiannaki [44] also observed that the port-based approach cannot classify peer-to-peer traffic. Sen et. al [51] complement these findings in their empirical study regarding port-based identification in peer-to-peer applications. They found that standard ports only accounted for 30% of their dataset thus port-based identification offered relatively poor performance.

Furthermore, port-based classification performs poorly in more complex classifying tasks, such as application or service identification. This is due to classes being defined by more features than just the used port number. This is supported by Sen's et. al findings in their empirical study [51].

3.1.2 Payload-based classification

Payload-based classification methods enhance classification accuracy compared to port-based classification [61, 11]. Instead of inspecting the packet's port numbers, payload-based methods inspect the packet's payload. The payload can be used to generate a signature by, for example, determining the characters or bits that describe the packet. The payload information can also be used to aggregate the packet data. By not relying on the port numbers, payload inspection can avoid the problems of random port assignments and masquerading. Payload-based classification can be divided into deep packet inspection (DPI) and stochastic packet inspection (SPI) [56]. There exists ambiguity in the term payload-based classification. This generally refers to using DPI or SPI [11, 52, 61, 56].

DPI, sometimes referred to as signature-based identification, searches the payload of the packets for characteristics that define a protocol, application, or services [56]. These characteristics can be, for example, signatures, patterns, or keywords. DPI has proven to achieve great accuracy and has been implemented in several open source projects, such as the Linux kernel.

Khandait et al. [36] proposed a DPI framework that creates application signatures by using application keywords found in the payload. The framework was able to detect different protocols with 90% to 100% accuracy. Furthermore, there are multiple empirical studies that use DPI to detect either protocols or applications, mostly focused on protocol classification [11, 49, 56]. All the studies achieve great classifying accuracy and are therefore more focused on making the DPI frameworks more efficient in terms of computing and memory requirements.

However, DPI requires significantly more resources as analyzing the payloads is more computationally intensive than just analyzing the port numbers [11, 56]. Another drawback of DPI is encrypted traffic such as HTTPS and TLS [11]. Encrypted traffic in its nature does not reveal any information about the encrypted content and therefore cannot be used to generate any signatures. Finally, defining the characteristics of the traffic can be time-consuming and challenging [56].

SPI is a technique designed to address challenges in partially encrypted traffic

classification and predefining the characteristics by automatically identifying distinctive patterns for specific protocols [56]. It introduces methods that go beyond traditional payload-based classification by focusing on probabilistic and statistical tools to improve efficiency. SPI aims to automate the identification of unique patterns within a protocol. For example, some approaches, like Common Substring Graphs (CSG), are used to efficiently detect common string patterns in packets. This data structure improves the recognition of recurring packet features. By automatically identifying the unique patterns, one does not need to thoroughly understand the underlying network traffic.

Several approaches within SPI leverage statistical analysis of packet payloads [56]. One method directly uses the values of the initial bytes of a packet's payload as features for machine learning algorithms [25]. Another method applies the Pearson Chi-square test to study the randomness of the first few payload bytes, which helps in modeling the syntax of the protocol used by an application [19]. This latter method is especially effective for handling protocols with partially encrypted payloads, such as Skype or P2P-TV.

Although SPI has proven to achieve high accuracy in traffic classification and partially encrypted traffic classification, it is a computationally demanding process [61].

Zhao et al. [61] inspects the payload-based classification combined with ML methods. Combining ML methods with payload-based classification methods can mean either that the ML training algorithm uses input that has been derived from the payloads or that the ML method is combined with DPI or SPI, for example using DPI to classify unencrypted traffic and ML to classify encrypted traffic based on flow level data. The existing ML methods in the payload-based classification domain can be divided into supervised, unsupervised, and semi-supervised classification. Zhao et al. [61] discovered that most of the research was done with supervised classifiers. Furthermore, most of the features were inferred from the flow-level data, using the first few packets of a flow. Some studies also used the first few bytes of the payload. As with DPI and SPI, the ML methods also achieved great accuracy in classifying applications and protocols.

One example of a payload-based ML classifier is from Wang et al. [58] which proposed a traffic classification method using packet payload. They extracted the first N bytes of the flow payload, identified common substrings from byte strings of a given application class, and consolidated them into a single token set, removing duplicates. Feature extraction was applied to reduce the token set size, and six machine learning algorithms were used to assess performance, achieving over 99.5% accuracy. The method is efficient in classifying applications and protocols.

Finally, it is important to note that with payload-based classification network privacy and regulations are a major concern. The payload-based technique involves a detailed examination of packet contents, which can lead to breaches of privacy policies and regulatory violations [52].

3.1.3 Statistics-based classification

Statistics-based traffic classification methods rely on flow-level statistical features rather than the packet payload for identifying traffic types [52]. Key features include metrics like average packet size, number of packets, and flow duration. These classifiers differentiate between applications based on statistical differences in network behaviors, leveraging probabilistic models and machine learning techniques for improved accuracy. Unlike payload-based methods, they are lightweight, work with fully encrypted traffic, and are adaptable to evolving traffic patterns. However, they tend to be less accurate than deep packet inspection (DPI), limiting their current commercial use.

The flows can be constructed by for example, aggregating the packets that have the same port in five-minute intervals where each aggregated row contains information on the number of packets sent in that interval, the average payload size, the average length of the session, and the protocol, to name a few of the most important attributes [11].

In the last ten years, statistics-based classification has been researched extensively [11]. Most of the statistics-based classification methods use machine learning. Statistical machine learning classification can be broken down into three parts: supervised machine learning, unsupervised machine learning, and semi-supervised machine learning.

The statistics-based classification is heavily focused on supervised machine learning [11, 61]. In an extensive traffic classification survey, Azab et al. [11] categorize the statistical supervised machine learning approaches into three categories: full-flow monitoring, sub-flow monitoring, and detecting untrained versions. Full flow monitoring encompasses the whole lifecycle of the capture, i.e., each packet in the flow is considered. Sub-flow monitoring only considers a few of the first packets in the flow, reducing the computational complexity significantly. Sub-flow monitoring also allows real-time classification as there is no need to capture the full packet flow. Azab et al. [11] recognized detecting untrained versions as its own category because the other solutions did not consider previously unseen versions of different applications. The main finding by Azab et al. [11] was that the full flow and sub-flow monitoring can achieve good accuracy, but it heavily depends on the used training dataset. Detecting untrained versions decreased accuracy significantly.

In contrast to supervised learning methods, unsupervised learning does not need a labeled dataset. Clustering is a widely used technique in unsupervised network traffic classification [11]. Clustering can discover hidden or unknown patterns by clustering data points that are on some metric similar to each other. This reduces the effort required to either obtain a labeled dataset or manually label an existing one. However, unsupervised learning tends to deliver less accurate results compared to the supervised approach.

The semi-supervised approach combines the advantages of both supervised and unsupervised methods to perform classification [11]. It produces highly accurate results by using unsupervised algorithms for tasks like feature selection or labeling the dataset while relying on supervised algorithms for classifying network flows. However, because it merges the two techniques, it requires more computational resources when

building the model.

Zhao et al. [61] summarise the statistics-based machine learning methods with the following remarks. The most frequent features used include the number of packets, packet size statistics, packet inter-arrival time, and total bytes transmitted. These are preferred because they are straightforward to collect and process, while also offering stability and robustness. When comparing the accuracy of various methods based on statistical information, there is significant variation, with some achieving up to 98% accuracy, while others reach only around 80%. The level of classification detail also differs greatly between methods.

The literature is focused on machine learning when considering statistics-based classification, but it can be approached with classical methods as well, i.e., methods that do not use machine learning [46]. In the classical approach, network traffic patterns are modeled using statistical distributions to understand different network states, such as reachability in communications. Statistical metrics like the number of packets and failed flows are used to create distributions, such as negative exponential or Gaussian, for traffic analysis. Additionally, text classification can be applied to packet headers, with statistical models detecting new patterns. However, these models often face limitations due to their static nature, struggling to adapt to evolving internet traffic patterns. Machine learning techniques have been introduced to improve and enhance the adaptability of these statistical approaches.

Statistics-based methods are lightweight and can be applied to encrypted traffic. However, the disadvantage is reduced accuracy compared to payload-based methods [56].

3.1.4 Behavior-based classification

Behavior-based classification examines the traffic from a broader point of view, focusing on how a host interacts with other entities on the network [61]. By monitoring the traffic, behavioral classifiers aim to identify applications based on their unique communication patterns, such as the number of hosts contacted, the transport layer protocols used, and the distribution of ports.

This method is grounded in the assumption that different applications generate distinct network behaviors. For example, peer-to-peer (P2P) applications often communicate with many different peers, typically using a limited number of ports for each peer [56]. In contrast, web servers are often contacted by multiple clients using numerous parallel connections. By examining these behaviors, it becomes possible to classify the traffic and infer the type of application running on a host without directly inspecting the content of the traffic.

The strength of behavior-based traffic classification lies in its ability to observe long-term patterns rather than individual packets or flows [56]. It provides a more holistic view of network behavior, which can be effective when traditional methods fail, such as when traffic is encrypted or when ports are dynamically assigned. Similar to statistics-based classification, behavior-based classification is also lightweight. In addition, behavioral classifiers can achieve the same level of accuracy with less information. However, the granularity of its results may be limited, meaning that

it can sometimes struggle to differentiate between closely related applications or services.

Behavior-based classification has mostly been implemented with machine learning, especially in the last ten years. The machine learning approaches can be further divided into supervised and unsupervised learning methods. Similar to statistics-based classification, supervised behavioral classifiers have received the most attention. [61]

One example of a behavior-based classifier is an empirical study by Kohout et al. [38]. They used snapshots of user activities to perform context simulation on HTTPS traffic. The snapshot is a 5-minute interval containing all the requests issued by the same user when establishing a TLS tunnel. Based on the snapshot Kohout et al. [38] classified the traffic as normal or abnormal by using three different machine learning algorithms (Neural Networks, Random Forest, and Extreme Gradient Boosting). They were able to achieve up to 90% precision when classifying normal and abnormal traffic.

3.1.5 Correlation-based classification

The correlation-based classification is a method used to identify and categorize types of network traffic by finding patterns or relationships (correlations) between different flows of data [60]. Instead of just analyzing each packet or data flow on its own, this method groups related flows together and classifies them based on their similarities.

For example, if a set of network flows are all coming from the same source and behaving similarly, they are likely related to the same type of application or a service group (such as a video stream or a web browsing session) [60]. By analyzing these connections, the system can more accurately predict what kind of traffic it is, even when there is very little data available to train the system. This helps improve accuracy, especially when limited information is available.

Zhang et al. [60] were the first to experiment with the idea of correlation-based classification. They implemented a framework that improves network traffic classification by using a Bag of Flows (BoF) approach and integrating it into a nearest neighbor (NN) classifier. The goal was to use the correlation information between flows to improve classification accuracy, especially when the amount of training data is limited. The BoF consists of groups of flows in which the flows belonging to a group share certain similarities. Zhang et al. [60] grouped all the flows together that had the same destination IP, destination port, and protocol. In the classification process, BoFs are classified instead of individual flows, i.e., all the individual flows in a bag receive the same class as the bag. Zhang et al. [60] classified the BoF with three different nearest neighbor (NN) methods. NN methods calculate the distance of BoF to the data used in training. For example, the average nearest neighbor method takes the average distance of all the flows in a bag and then classifies the bag based on the smallest average distance. Zhang et al. [60] were able to show that correlation-based classification can improve classification accuracy, especially when a limited amount of data is available.

Zhao et al. [61] covered correlation-based classification in their survey, especially

from the perspective of machine learning. They discovered that all the correlation-based classifiers use flow-level data. This approach makes sense, as the methods categorize flows within clusters that exhibit similar attributes, such as three-tuples and temporal correlations. The features highlighted by these methods primarily include header information—such as IP addresses, port numbers, and protocols—as well as statistical data about packets, including packet sizes and inter-arrival times. They also discovered that many of these methods leverage supervised machine learning algorithms, which contribute to their impressive classification accuracy.

However, it appears that there is room for improvement [61]. Most of these methods struggle to meet important criteria, such as robustness, real-time classification capabilities, and the identification of unknown traffic simultaneously. While it's noteworthy that many of the correlation-based methods report accuracy rates above 90%, with some exceeding 99%, it seems that they typically classify traffic primarily at the protocol granularity. This focus on classification granularity suggests that further refinement could enhance their effectiveness.

3.1.6 Deep learning

In recent years, researchers have started incorporating deep learning techniques to classify network traffic based on its underlying application or service group [11]. Typically, raw network traffic data is fed into deep learning algorithms to capture the spatial or temporal characteristics of the traffic. However, deep learning methods have also utilized features that are used in payload-based and statistics-based classification, such as flow-level statistics and packet-level data. A key advantage of deep learning is that it eliminates the need for feature engineering, unlike earlier methods, making it easier to implement without requiring specialized knowledge in the field. Automatic feature selection allows it to handle the dynamic nature of network traffic where new classes and patterns emerge constantly.

Deep learning has demonstrated high accuracy in classifying network traffic with precise detection [11]. However, the technique requires a large amount of data for training and demands significant computational resources, particularly during the training phase. Deep learning models also result in a "black box" where it is hard for one to understand where its results were derived from.

Deep learning has also achieved high accuracy in encrypted traffic classification [1, 40, 50]. However, the performance is highly dependent on the target dataset. Malekghaini et al. [42] explained this phenomenon through data drift. Data drift occurs in dynamic data where the distribution of input data changes over time. Malekghaini et al. [42] tested the data drift performance of two state-of-the-art encrypted network traffic classifiers and found that on average the accuracy dropped by approximately 39% percent. These findings indicate that the existing deep learning methods to classify encrypted traffic are not viable in real-world dynamic networks.

3.1.7 Summary

The field of network traffic classification has been evolving towards machine learning based classifiers to reduce computational complexity and overcome issues related to

Table 3: Strengths and Limitations of Traffic Classification Techniques

Technique	**Strengths**	**Limitations**
Port-based	Simple and efficient to implement	Increasing use of dynamic ports and encrypted traffic makes it unreliable
	Works well for legacy protocols	Fails with applications using non-standard ports
Payload-based	Highly accurate when payload data is available	Limitations with encrypted or compressed payloads
	Can detect specific applications based on signature matching	High computational cost and potential privacy issues
Statistical	Can handle encrypted traffic	May suffer from reduced accuracy for similar traffic types and compared to payload-based techniques
	Low computational cost compared to payload-based techniques	Requires extensive training data
Behavioral	Can identify applications based on communication patterns and flow behavior	Requires long observation windows and is computationally intensive
	Adaptable to encrypted traffic	Difficult to handle new, previously unseen behavior
Correlation-based	Good accuracy, especially with limited information	Classification granularity is lower than with statistics-based methods
	Low computational cost compared to behavior-based techniques	Reduced accuracy compared to payload-based techniques
Deep Learning	Highly accurate	Requires a large amount of labeled data and significant computational resources
	Capable of automatic feature extraction	Lack of interpretability (black-box model)
	Adaptable to complex patterns, including encrypted traffic	Data drift

data complexity such as encryption and obfuscation. Of the six reviewed techniques, the most common techniques are statistical, behavioral, and deep learning. Port-based methods have not received much attention in recent years. Payload-based techniques have limited use cases. Table 3 summarizes the strengths and limitations of the covered traffic classification techniques.

While the field has made substantial progress, significant challenges remain. The following list describes the common challenges faced by the majority or all the traffic classification techniques:

- **Data availability & collection** There exist multiple issues when it comes to the datasets being used. One of the biggest issues is the lack of public datasets [11, 46]. Even though there exist several public datasets, there is a demand for more [11]. Many of the used datasets are relatively old which poses a problem in dynamic and fast-developing network environments. Furthermore, there exists a great number of different kinds of networks, applications, and users. To efficiently classify each unique instance, one also needs training data for those specific instances. Public datasets also cannot expose critical information related to the network and its users due to privacy and security concerns.

  Collecting and producing new datasets is time-consuming and complex [11]. In general, datasets should cover a long period of time. This could range from days to years depending on the task. In other words, the data should be plentiful as the classifiers are more accurate the more data there is available. This applies especially to the machine learning classifiers. Furthermore, datasets should not contain imbalances, i.e., bias towards a certain class [11]. Rather the datasets should contain a balanced representation of each class.

  In general, the datasets should also contain labels [11]. ML models rely on large datasets with labeled traffic. Labeling often requires manual annotation, which is labor-intensive and time-consuming. Labeling can also be performed with toolings such as port-based labeling and DPI-based labeling but these methods cannot guarantee 100% accuracy [61]. Oliveira et al. [45] suggest that the research should focus on building classifiers that can tolerate some imperfections in the labeling accuracy as achieving a perfect ground truth can be challenging or even impossible. Labeling can be avoided by using unsupervised learning algorithms. However, it has reduced accuracy when compared to supervised methods. Furthermore, it can be difficult to interpret what type of clusters have been created as the labels are missing [11].

- **Unknown traffic** Most of the traffic classification studies ignore the classification of unknown traffic [52, 61]. This means that the classifier can only classify the traffic it already has information on. In the dynamic and evolving network environment, this creates a problem as the classifiers need to be trained continuously with new information. At present, accurate and efficient classification is still an open problem [61].

- **Granularity** Fine-grained traffic classification lacks accuracy and efficiency [61]. Most existing methods classify traffic either into protocols or services. The existing methods can achieve good accuracy but for certain tasks and increased visibility, more fine-grained traffic classification is needed. There exist only a few methods that can classify traffic to specific application names or further into specific traffic inside a specific application. Furthermore, the accuracy of the more fine-grained classification needs to be improved.

3.2 Version classification

In the existing traffic classification literature, there is very little discussion on classifying different versions of applications. Azab et al. [11] mention briefly that classifying untrained versions is challenging when there are noticeable changes between the version numbers, such as a change in protocol. However, their remarks were in the context of service and application identification, not focused on classifying individual versions. A few studies have then focused on tackling the issue of classifying changing applications accurately [12, 16]. To the best of the author's knowledge, there has not been any research on classifying different versions of applications by observing network packets. Based on these findings, this chapter introduces a new version-sensitive approach that focuses on capturing changes between version increments.

Classifying different application versions includes collecting network traffic traces from multiple versions of the same application and classifying the network traffic correctly onto the corresponding application version. The version classification approach is even more granular than service or application-level classification. Version changes can be extremely small, for example, configuration value change or change in one line of code that only executes in a specific scenario. Therefore, detecting the versions also poses a great challenge.

Version classification is not only about classifying the different versions of an application but more generally about detecting small changes. Detecting small changes is important for network administrators to understand and secure the network. Vulnerabilities can be introduced by even the smallest changes in code or configurations [21]. Also, the attacks carried over networks can be small. In some cases, the attacks can only differ by one character [20]. Furthermore, detecting small changes is not only about detecting attacks and vulnerabilities but also about understanding the network and what has changed. This can be effective, for example, when trying to find bugs or understand user behavior.

There exist a few challenges as to why there is a lack of research on version classification. The lack of public datasets is one of the biggest reasons for this deficit. There are no network traffic datasets that would focus on different application versions. This means that researchers need to create their own datasets but that will introduce previously mentioned challenges about data collection and labeling. Furthermore, it can be difficult to generate datasets where the different application versions behave similarly enough to have a fair comparison. For example, if the goal is to detect WhatsApp version 1 from version 2 but on the dataset version 1 has traffic containing text messages and version 2 has voice calls, it would result in classifying the versions correctly, but the reason would have been most likely the different nature of the traffic generated on that instance. The correct approach would be to record the same type of actions on each version, if possible. In other words, the dataset should only reflect the changes that define the versions and nothing more. Furthermore, capturing small changes in the observed network traffic can often include inspecting the payloads which poses another problem as exposing the payloads publicly is a security and privacy risk.

4 Traffic classification framework

This thesis introduces a traffic classification framework designed to provide a fine-grained traffic classification as well as to address some of the key issues identified in the field. The framework is designed to detect subtle changes or anomalies in the observed network traffic. This is accomplished by distinguishing application versions apart. Small increments in application versions can be hard to detect on the network level. Therefore, if the framework succeeds in this task, it is a strong implication that it can detect subtle changes.

The other issues relate to data collection and availability. The framework is designed to be able to capture traffic from Kubernetes environments, providing a new dataset that contains traffic from real-life applications. Furthermore, as the data is collected in a controlled environment, it allows for full inspection of the packet payloads without privacy concerns. The framework employs SPI to be able to achieve the highest accuracy possible required for fine-grained classification. In the related work, packet inspection was recognized to provide the best accuracy, at least with relatively small datasets.

The framework is split into four steps. The first step is collecting network traffic from different versions of an application. The next step is creating a fingerprint based on the captured traffic produced by each application version. The fingerprint is used to determine if a particular set of network packets originates from that version or belongs to some other version. In the third step, the fingerprint is compared against a PCAP file, and the comparison yields the packets that differ between the fingerprint and the input PCAP file. Finally, these differences are given to a machine learning algorithm that classifies the PCAP file into that version or not, based on the packet differences. Figure 8 illustrates the framework in a simple manner.

Figure 8: Overview of the traffic classification framework

The framework is designed to be flexible and easily modifiable. This is reflected in each step. The traffic can be captured from any application or network. In this thesis, the framework is implemented to capture traffic in the Kubernetes environment, but this could be replaced with any environment. Furthermore, any application could be recorded. The fingerprint creation process works with any PCAP file and is not concerned about the domain or origin of the file. This is also true in the fingerprint comparison process. The classification step, distinguishing the version apart, is also not concerned about versions specifically. It can be set to classify any differences between a fingerprint and a PCAP file. With only slight modifications to the source code, users can run the framework in different environments. Also, the fingerprint creation process can be modified to work with other formats than PCAP.

4.1 Software requirements

The framework is implemented with multiple tools and Python scripts. To guarantee that the framework will run, the following requirements should be met:

- **Minikube** 1.34.0
- **Docker** 27.3.1
- **Helm** 3.16.2
- **Kubectl** 1.31.2
- **Python** 3.10.12
- **A Linux-based operating system**

The source code and details on how to run the framework can be found in the author's Github [27].

4.2 Data collection

The first step of the framework is to collect data. This step must be executed well as problems at this stage will cascade downwards. Issues can be caused by, for example, corrupt or misclassified data, which will in turn mislead the decision process. The data collection phase aims to collect data that is non-synthetic in the sense that assumptions and derivations from the data can be applied to real use cases as well. Furthermore, the data collection process aims to be as pure as possible, meaning that the data is recorded as is and no modifications are made to the data.

The collected data consists of network packets generated by an application. The network traffic is divided into startup and idle traffic, with no user-generated application traffic captured. This approach is employed to prevent the data from being classified as synthetic. Both the startup and idle traffic are naturally generated by applications. Generating natural application user interactions is a challenging task, and as a result, it has not been explored in this thesis. Despite this choice, the framework could be used to capture, for example, a live application in a production environment. However, within the context of this thesis, access to such environments was not possible.

4.2.1 Process

Before the data collection phase can begin, the user should have an application that can be deployed in Kubernetes, and the application should have multiple versions available. Note that the user can deviate from this plan as mentioned in Section 4. However, if the user wants to use the framework as is, a Kubernetes deployment is required.

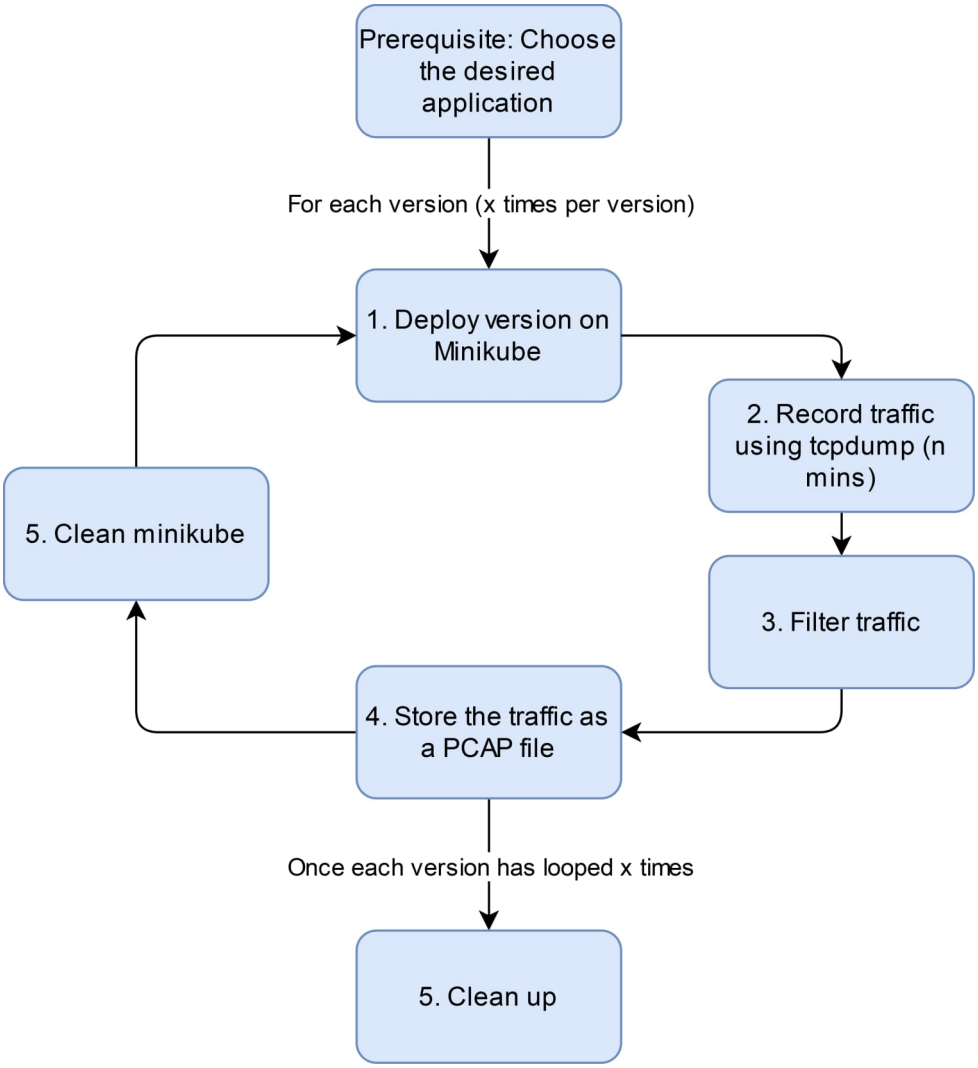

Figure 9: Overview of the data collection process

The following list describes the steps that are taken to collect data from a singular application. Figure 9 illustrates an overview of these steps and how the steps relate to each other.

1. Deploying the version: The versions are deployed on Minikube one version at a time. Each version needs to be deployed multiple (x) times. This is required as later the created fingerprint is an "average" of each version deployment. If versions are deployed only once, the framework cannot understand what packets define the version. However, it is important the framework alternates between versions and does not deploy a specific version multiple times in a row. This leads to a problem where a specific version can be defined by a timestamp. For example, app version 1 is deployed, and it takes place on Monday at 11 PM, it takes roughly an hour to deploy and capture the version 30 times. When the version 2 capture starts, it is already Tuesday. If the app version one sends packets that contain timestamps, the fingerprinting tool will mistake that Monday in the timestamp relates somehow to

version 1.

2. Listening on the network interface: Once the version has been deployed, the network traffic produced by the version is collected. In practice, the data is collected using *tcpdump* and listening for n minutes on the deployed Pods. The framework does not set any criteria for choosing n. After n minutes, the data is stored in a PCAP file.

3. Filtering the traffic: The PCAP file is filtered to only contain packets that either originate or are going to the application Pods. This is the type of traffic that allows for the framework to create a fingerprint. The traffic in question are the Service and Pod-To-Pod communication, health checks and probes, Ingress and Egress traffic, and telemetry and monitoring traffic (see 2.3.3). The other types of traffic are more related to Kubernetes itself and the infrastructure.

4. Storing the captured traffic: Finally, the PCAP file is stored in the filesystem.

5. Next steps: After every version has been deployed x times, Minikube will shut down. Otherwise, the Minikube environment is cleaned and the process repeats.

4.2.2 Example

This section provides an example application to illustrate the type of traffic that can be captured during the data collection process. The example uses the WordPress Helm chart available on Artifact Hub [15]. The application includes two components:

- A **WordPress server**, which hosts the website and editor.
- A **MariaDB database**, which stores website content such as images and text.

As of October 14, 2024, the latest Helm chart version for WordPress is **23.1.21**, which uses WordPress server version **6.6.2** and MariaDB version **11.4.3**.

When the WordPress application is deployed, two Pods are created: a server Pod, which interacts with the website, and a database Pod, which stores the data. The server Pod communicates with the database Pod using MySQL messages. This interaction includes:

- Initial greetings and log in.
- Creation of SQL tables.
- Insertion of example data.

Additionally, health check probes target the server at the path `"/"`. Other observed traffic includes TCP retransmissions, DNS packets, and slight variations in messages depending on the application version.

4.3 Fingerprint creation

The purpose of a fingerprint is to identify a specific application version. The framework builds up a fingerprint based on selected features from the network packets. The fingerprint should be able to capture the network packets that define the version, i.e., make it distinguishable from other versions and applications. The framework's goal is to create a fingerprinting mechanism that is application-agnostic. It should be able to create fingerprints without the need to understand the underlying network environment and traffic. To achieve the goal of making the fingerprinting process application agnostic, the fingerprint is essentially a summary of all the deployments for that version.

4.3.1 Packet similarity

Before covering the fingerprinting process, it is important to understand the concept of packet similarity. The goal is to be able to categorize packets as similar if their nature and intentions are the same. For example, requesting a page from a web server can result in slightly different packets depending on the browser, timestamps, etc. but the purpose of the packets is still the same. Ideally, these packets are deemed similar.

The framework defines packets as similar if they have the same protocol-length pair. This is a simple but naive solution and as such should be noted that it contains some limitations. The first limitation relates to falsely classifying two packets as similar. This can occur if two packets, with completely different intentions, happen to share the same protocol and length values. The second issue relates to missing packets that are similar. For example, a messaging server can receive different-sized packets depending on the message length. If it is deemed that sending a message is a similar enough action regardless of the content, the framework will fail to determine these packets as similar due to the length not matching.

Finally, it is important to understand the **protocol** definition. In the context of this thesis, protocol does not map one-to-one with the existing network communication protocols. The packets are handled by *pyshark*, [23] which is a wrapper for *tshark* [59]. The actual protocol definition comes from the *tshark* library and how it dissects the packets. The packets are dissected into layers, and each layer is viewed as a protocol. For example, an HTTP packet can be divided into the link, IP, TCP, HTTP, and DATA-TEXT-LINES layers. The first four layers are familiar from the OSI layer [34] but the DATA-TEXT-LINES layer is not a recognized communication protocol, but rather a unique layer dissected by *tshark*. There are many more examples of these unique protocols that further split the OSI application layer, for example, JSON and MEDIA. *Pyshark* was used to detect the protocols as manually crafting rules to detect each protocol would be extremely time-consuming. Furthermore, it would require extensive domain knowledge of different network protocols. In this thesis, the protocol of the packet means the highest layer protocol that *pyshark* can recognize. The packet length refers to the highest layer's payload length. The payload is the content that is included in the highest layer.

The protocol parsing process can be better understood through an example.

Consider an HTTP POST request captured in a TCP stream. Using *pyshark*, the highest protocol layer for this request is identified as DATA-TEXT-LINES. This protocol encapsulates both the HTTP headers and the payload together in a single unit. Below is a simplified representation of such a request:

```
POST /example HTTP/1.1
Host: www.example.com
Content-Type: application/json
Content-Length: 27

{"key":"value","num":123}
```

If multiple application-level protocols map to DATA-TEXT-LINES, potential ambiguities arise. These are partly addressed by ensuring the payload length matches exactly. An improved solution would involve matching the entire sequence of layers, from the transport layer (e.g., TCP) up to the application layer (e.g., HTTP). For an HTTP GET request, the *pyshark* library identifies the HTTP protocol as the deepest layer. The header information serves as the payload in this case.

4.3.2 Process

The fingerprint creation process includes the feature selection and extraction steps. The feature selection includes selecting which packet information is used to create the fingerprint. The extraction step is transforming the packet information to form the fingerprint. The selected features are the packet's protocol, destination port, payload length, and the payload itself. IP addresses were considered as a feature but it is too difficult to derive any similarity metrics between different IPs. Matching IPs is also not feasible as the IPs change between deployments. The extracted features are described by introducing the fingerprint in the steps below.

Step 1: The fingerprinting process starts by selecting $x\%$ of the PCAP files for a specific version. These files are used to generate the fingerprints. The rest, 100% - $x\%$, are left for testing, i.e., for comparing against the fingerprint. Each PCAP file is then iterated over, going through all the packets in the file.

Multiple PCAP files (network traffic captures on different deployments) are needed as the fingerprint cannot be simply created from a single application version deployment. This is because multiple factors contribute to the observed network packets, such as timing and network conditions. For example, when running the WordPress version 21.0.0 three times, the number of packets differs. The first run had a total of 7521 packets, the second had 6049, and the third had 6611 packets. The almost 1500 packet difference could already indicate that there are at least 1500 packets that do not define the version. It could also indicate that there are at least 7521 packets that all define the version but, for some reason, the other deployments could not emit as many packets.

Step 2: For each packet, the packet's protocol and length are extracted. The packet's protocol is the highest layer protocol as explained 4.3.1. The length is the length of the payload (in binary) on the highest layer, for example, the HTTP

payload. The payload itself is converted to ASCII for better readability (to help users debug and understand the fingerprint better).

Step 3: The protocol and length pair are used as a key and the payload is the value. The key-value pair is added to the fingerprint.

Step 3.1: If there already exists a key, the payload is simply added to a set of payloads for which the key points to.

Step 3.2 If there is no existing key, the key is added to the fingerprint and a set is created as a value, containing the one payload. The goal is to group similar packets under the same key.

Step 3.3: While iterating over the packets, the framework also keeps track of the protocol-length pairs that have occurred at least once in each PCAP file. This is done to figure out all the similar packets that exist in all the version deployments. The set of common protocol-length pairs is stored in the fingerprint.

Figure 10 illustrates the steps from 1 to 3.3

Step 4: Once all the packets have been iterated, the framework starts iterating over all the key-value pairs in the fingerprint. For each pair, the value (set of payloads) is used to determine all the character indices that are common in the payload strings, across similar packets. The common indices are then stored under the key. Step 4 is illustrated in Figure 11.

It is essential to compare the payloads as just relying on statistical data is not enough. The small changes that happen between version increments do not necessarily affect the statistical data much. Therefore, the payloads need to be inspected to find changes in keywords and individual characters. There were some observable statistical patterns when it came to certain applications and comparing certain versions. The number of packets tended to decrease when the version number was incremented. However, this trend was weak, and, in most cases, a minor version increment did not affect the number.

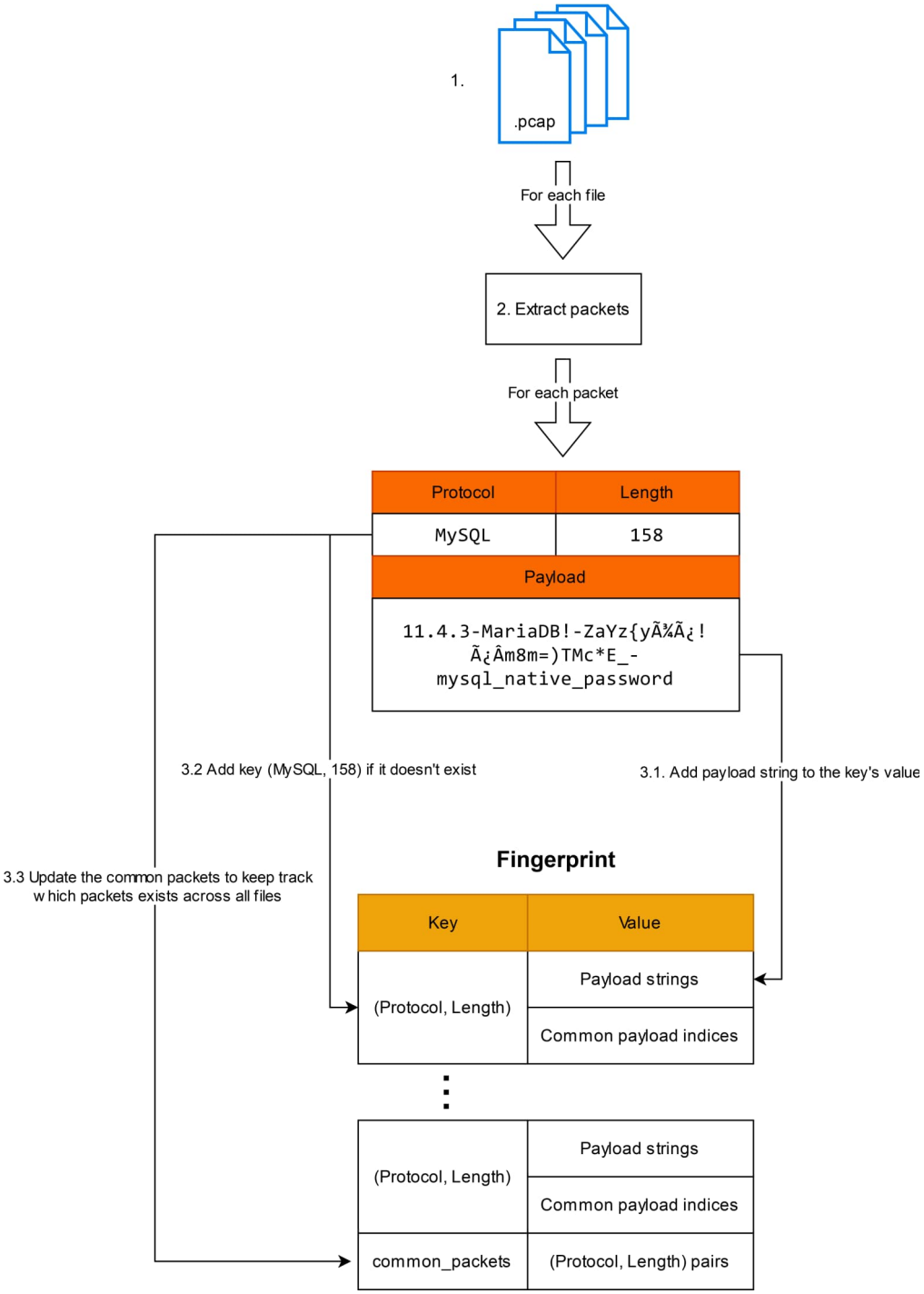

Figure 10: Overview of the file parsing process. Figure 11 describes the final step.

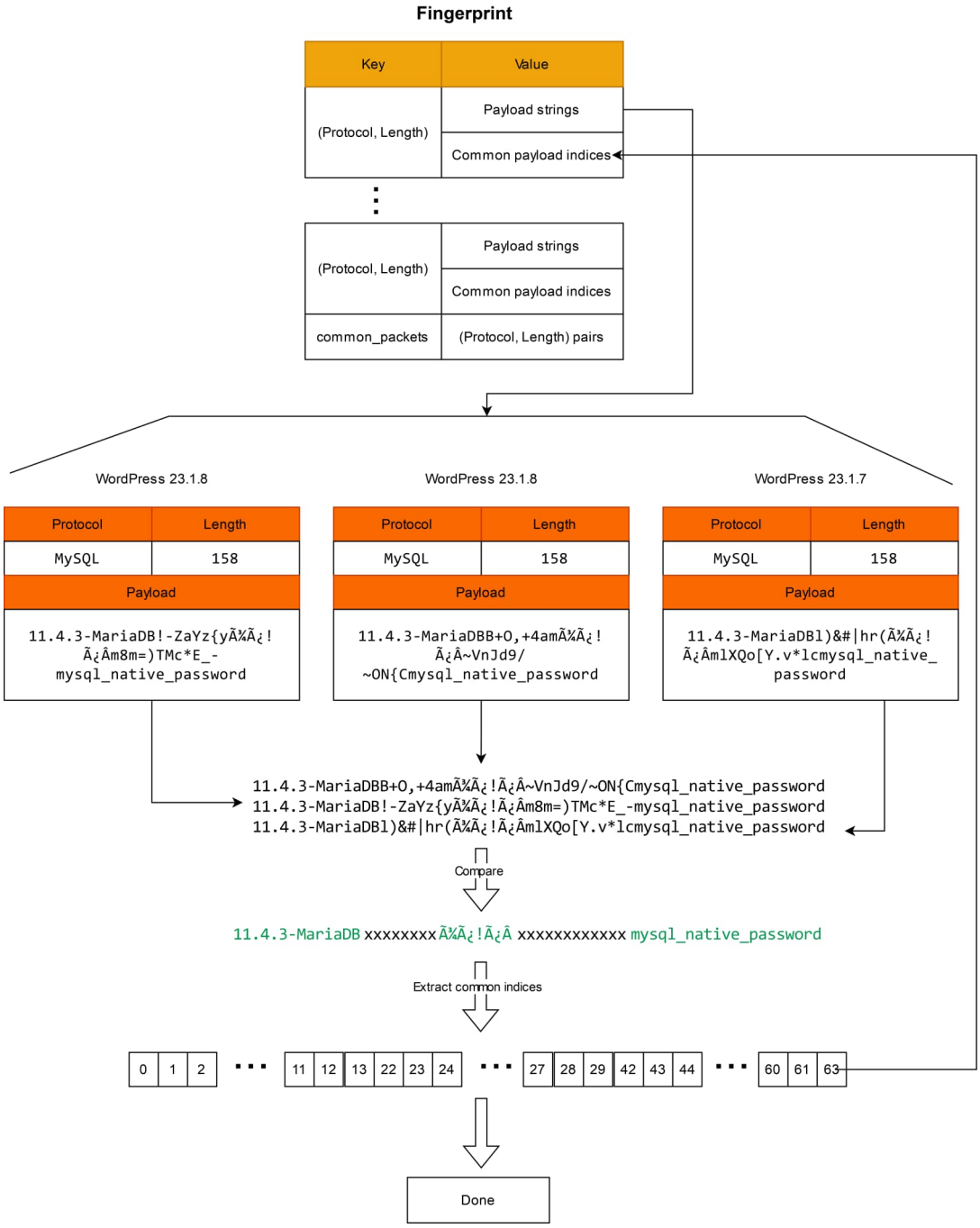

Figure 11: Overview of the payload comparison process (Final step 4)

4.3.3 Computational complexity

In big O notation, the computational complexity of the fingerprint creation process is as follows:

$$O(n * m) + O(a * b * c)$$

n = number of PCAP files used to create the fingerprint
m = number of packets in a PCAP file
a = number of protocol-length pairs
b = maximum number of unique payloads within a protocol-length pair
c = maximum length of the payload under the protocol-length pair

The big O notation here does not contain all the details of the fingerprinting process, only the most relevant steps which take up most of the time. Key observations are that there are multiple variables that affect the computational complexity, and the computational complexity grows linearly with each variable. Note that m and a are closely related as m grows it can be expected that a grows as well.

4.3.4 Limitations

The payload comparison process has some notable limitations. First, this only works well with unencrypted payloads. Conveniently, a lot of internal cluster traffic is not encrypted. Note that a service mesh can be applied to encrypt the internal cluster traffic. However, this does not mean that the fingerprint cannot be created for completely encrypted traffic as the fingerprint uses the packet protocol and payload length which are available for all packets. Even though some traffic is encrypted, the framework will apply to it as well. This simple approach is more convenient as there is no need then to determine if the packet is encrypted or not. Furthermore, if the traffic is only partially encrypted or it happens to contain some patterns, the framework will process those as well. When encrypted traffic is processed, the framework simply will not find any similarities between the payloads.

Besides dealing with the encrypted traffic, another limitation is that the payload comparison assumes that the payloads can be compared by looking at the index 0 and checking if the characters match, then moving to index 1 and checking if the characters match, and so on. This works as is later demonstrated by the accuracy of the framework. However, it is crucial to keep this in mind as this limits the use cases and sometimes can limit the accuracy as well. Even if the one has otherwise identical packets but the first packet is shifted by one character, the packets are deemed not to have any similarities. This limitation is illustrated in Figure 12.

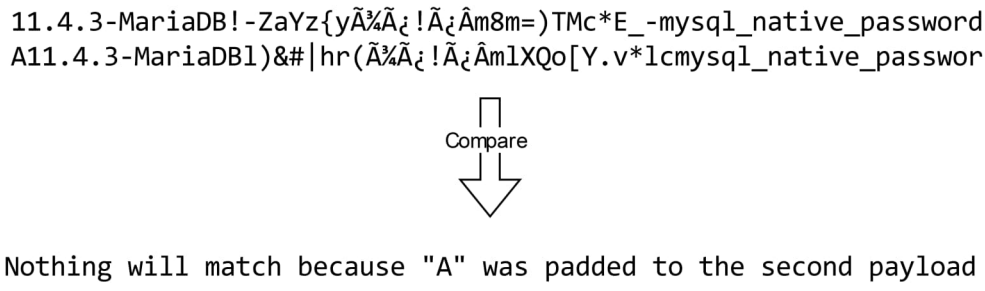

Figure 12: Limitation of the string comparison method

The resulting fingerprint is an object that contains the key-value pairs and

a "common_packets" key that points to the set of common protocol-length pairs. Each key-value pair consists of a protocol-length pair as the key, and the value contains two parts: a "payload" key, which points to a set of payload strings, and a "common_indices" key, which refers to a set of indices. These indices mark the positions of characters that are the same across all the payload strings.

The framework is proof of concept and therefore it should be noted that the fingerprinting mechanism cannot capture everything due to the aforementioned limitations like the naive string comparison. The goal is to capture enough defining packets so that it can be demonstrated that the framework can distinguish versions from each other.

4.4 Fingerprint comparison

Once the fingerprint is created, the framework can start comparing PCAP files from application version deployments with the fingerprint. The goal is to produce statistical differences between the PCAP file and the fingerprint. The differences include deviation from the payload, missing packets, and new packets.

4.4.1 Process

The comparison process starts by extracting the packets from the PCAP file using *pyshark*. The comparison process is illustrated in Figure 13. Each packet is iterated over and categorized into four categories based on its relation with the fingerprint:

1. **Different packet**: If the packet has a protocol-length pair that is also a key in the fingerprint, the key is used to fetch the payload and common indices values (from the fingerprint packets). The common indices are a set of indices that indicate which payload characters should be the same. The packet's payload is then compared with one of the fingerprint payloads and if there are any characters on the common indices that are different, the packet is marked different.

   The payload difference is strict. If only one character is different from the fingerprint, the packet is deemed different. This issue is greatly mitigated by the fact that the fingerprint is an average of all (except those that are used for testing) version deployments. If some characters are common across all deployments and suddenly there is a new packet that does not match those characters, it is a good indicator that the packet might not be from the same application version. Furthermore, it would be difficult to define a threshold for the packet payload difference. If it is established that 50% of the indices should match, how can it be verified that this threshold is sufficient across all applications? Therefore, it is more straightforward to keep a strict comparison method.

2. **New packet**: If the packet's protocol-length pair is not a key in the fingerprint, the packet is deemed new. This means that this specific packet has not been observed while creating the fingerprint.

3. **Missing packet**: For each packet, the fingerprint's common packets set is checked. If the packet's protocol-length pair is present in the set, it is removed from the set. Once all packets have been iterated, there is a set of common fingerprint packets that were not present in the PCAP file. All these packets are deemed missing.

4. **Trivial packet**: In this case, the framework cannot determine anything important about the packet. The packet is present in the fingerprint, and it matches the existing payloads.

The different, new, and missing packets form a set of **relevant packets**. Trivial packets are excluded from the set as they do not provide information that can be used to classify the version.

The *relevant packets* are collected into two formats: a CSV file and a PCAP file. The CSV file contains information about the packets that are specific to the framework and not present in the PCAP format. The following information is gathered: packet number, protocol, length, is it a new packet, is it a missing packet, payload, and payload difference. The payload difference consists of the characters that are different between the fingerprint's payload and the compared packet payload. There are some additional fields as well which are only relevant in debugging the process. The PCAP file contains the packets as they were captured by *tcpdump*. The PCAP file is used to debug and gain an understanding of the nature of the packets.

Finally, the previously mentioned CSV file is used to create a final CSV file that contains statistical information about the *relevant packets* (rather than listing each *relevant packet* on its own row). Appendix A1 lists the fields present in the final CSV file.

4.4.2 Computational complexity

In big O notation, the computational complexity of the fingerprint comparison process is as follows:

$O(n)$, n = number of packets in the compared PCAP file

The computational complexity grows linearly with the variable. The fingerprint comparison is more straightforward and efficient when compared to the computational complexity of creating the fingerprint.

4.4.3 Limitations

As in section 4.3.1, there is a potential limitation when comparing the fingerprints. The assumption on the packet similarity is that packets are deemed similar if they share the same protocol-length pair. As stated before, this might not be the case for all packets. When comparing the fingerprint, the definition of similar packets greatly affects the outcome. If there were other criteria to determine the similarity, the payload comparison would be different as the payloads are compared only across packets that are deemed similar.

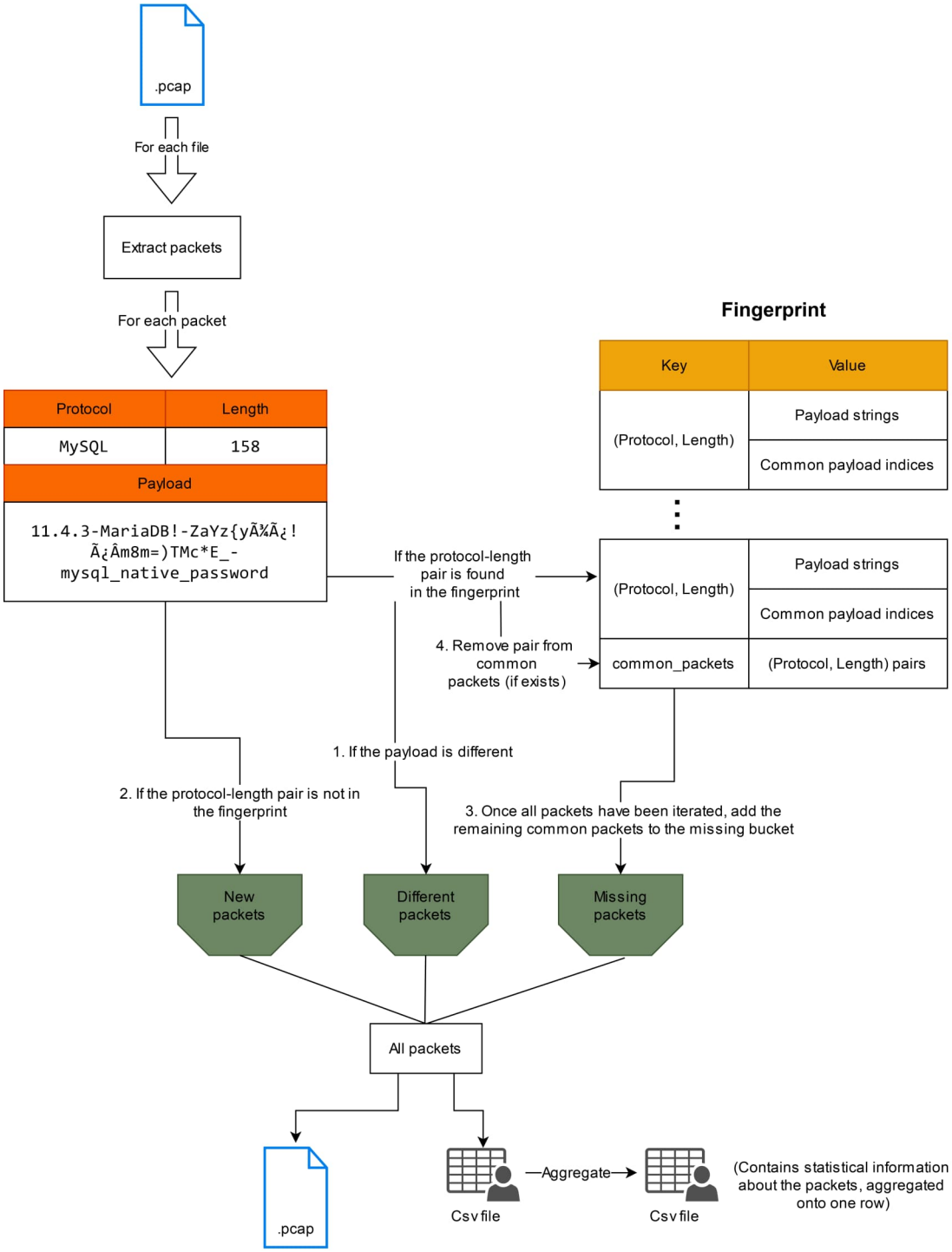

Figure 13: The process of comparing a PCAP file to a fingerprint

4.5 Classification

After a PCAP file is compared against the fingerprint, a classifier is created to classify the PCAP file as either originating from the same application version as the fingerprint or not. The classifier could be either made with machine learning

methods or classical methods. Classical methods could be, for example, setting certain threshold values to indicate if the traffic in the PCAP file is from the same version or not. For example, if the number of new packets is over 10, the PCAP file is from a different application version.

In the framework, machine learning has been chosen to implement the classifier. It is convenient as there is no need for human intervention, for example, no need to specify the exact threshold values by hand. Furthermore, the application might differ from each other which in classical methods would lead the user to tune the values for each application.

The classifier is implemented with Random Forest. The classifier takes the CSV file as input as described in Appendix A1. The training and classification processes are described in Figure 14.

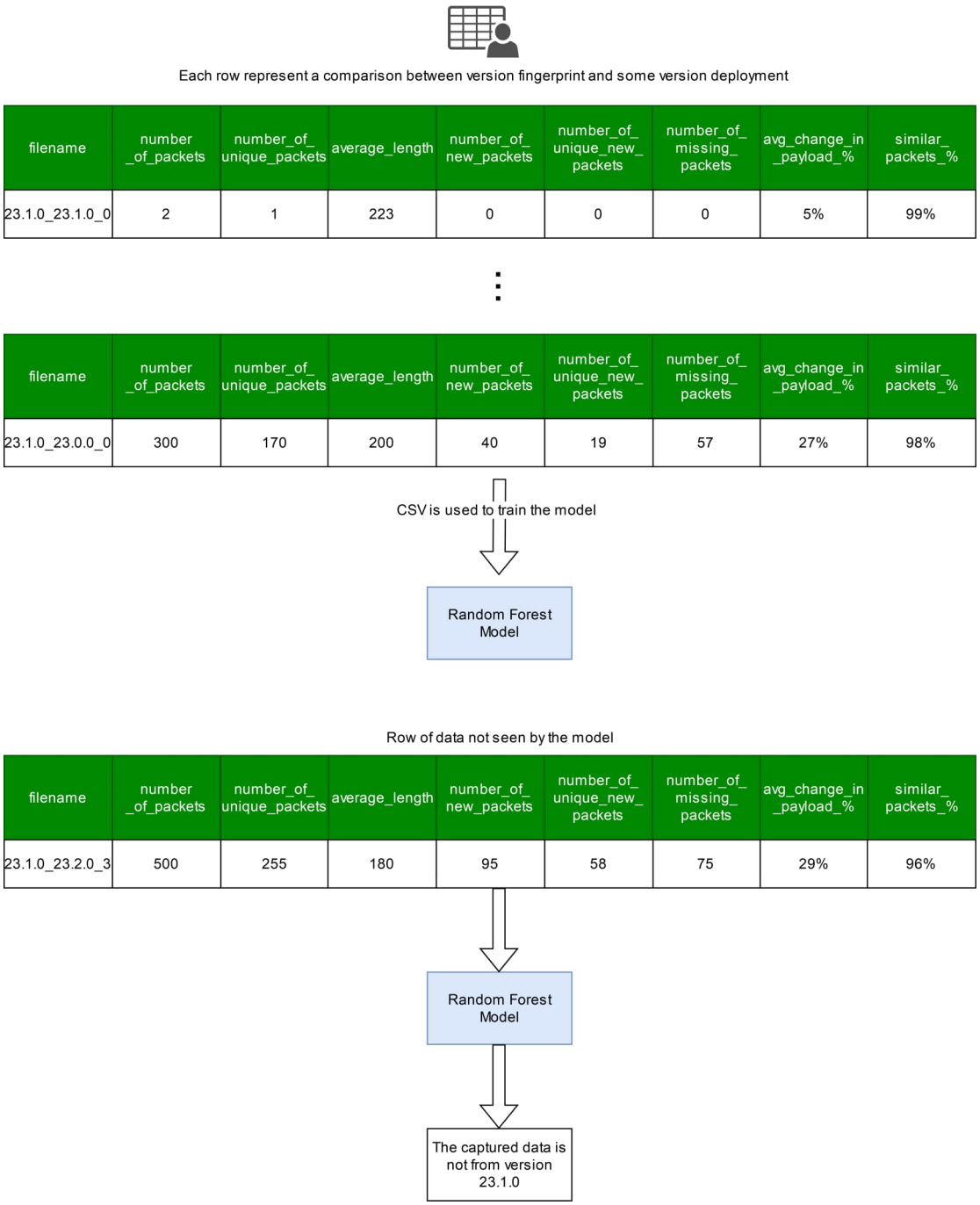

Figure 14: Overview of the training and classification process

5 The experiment setup

The framework gives the user the responsibility of choosing several parameters. The list below describes these parameters and what choices were made in this thesis. The choices are elaborated in the paragraphs following the list.

- **Source of Kubernetes applications**: Helm charts were chosen as the application templates and Artifact Hub was selected as the repository for collecting Helm charts.
- **Number of versions per application**: For each application, 10 distinct minor versions were chosen.
- **Number of PCAP files per fingerprint generation**: Each fingerprint was generated using 15 PCAP files.
- **Deployments per application version**: Each version of an application was deployed 30 times.
- **Duration of deployment recording**: Each deployment instance was recorded for 2 minutes.
- **Training and test data split**: The dataset was divided into 70% for training and 30% for testing.
- **Applications recorded**: A total of 20 applications were selected for recording. Detailed information regarding the selected applications is provided in Appendix B1.

Minor versions are used as they provide a good middle ground between major and patch versions. Patch versions often do not contain any changes that would reflect on the initial network traffic and major versions often contain such big updates that the framework would not be able to demonstrate its accuracy to its full potential. Few exceptions were made to this as initial inspection of the traffic indicated that it would be possible to classify patch versions consistently. The exceptions are Concourse, Harbor, and Jupyterhub. They contain some patch versions that can be classified, but this does not seem to apply in general. Also, it was impossible to keep the minor version increments consistent as for some Helm charts there were versions that did not work. This led to increasing the minor version by two on some occasions.

Ten Helm chart versions are chosen as a compromise. The perfect solution would be to take all the versions for each application but, in reality, deploying and recording the application version takes time. Ten versions per application was a reasonable choice in the given timeframe while providing confidence that the results are not attributed to pure coincidence.

An initial analysis was conducted to pick the number of PCAP files to generate the fingerprint. 10-15 files seemed to maximize the accuracy of the fingerprint. In some applications, the accuracy even decreased after 15 PCAP files. This can happen

as files used in the fingerprint are removed from files used to test and compare the fingerprint, which means less data for training the classifier. Based on these findings, 15 PCAP files out of 30 were deemed to be a sufficient fit for generating fingerprints. The analysis is showcased in Figure 15. The remaining files were used for comparison against the fingerprints. The more files, the better. However, there is a finite amount of time and therefore 30 PCAP files are recorded in total per version.

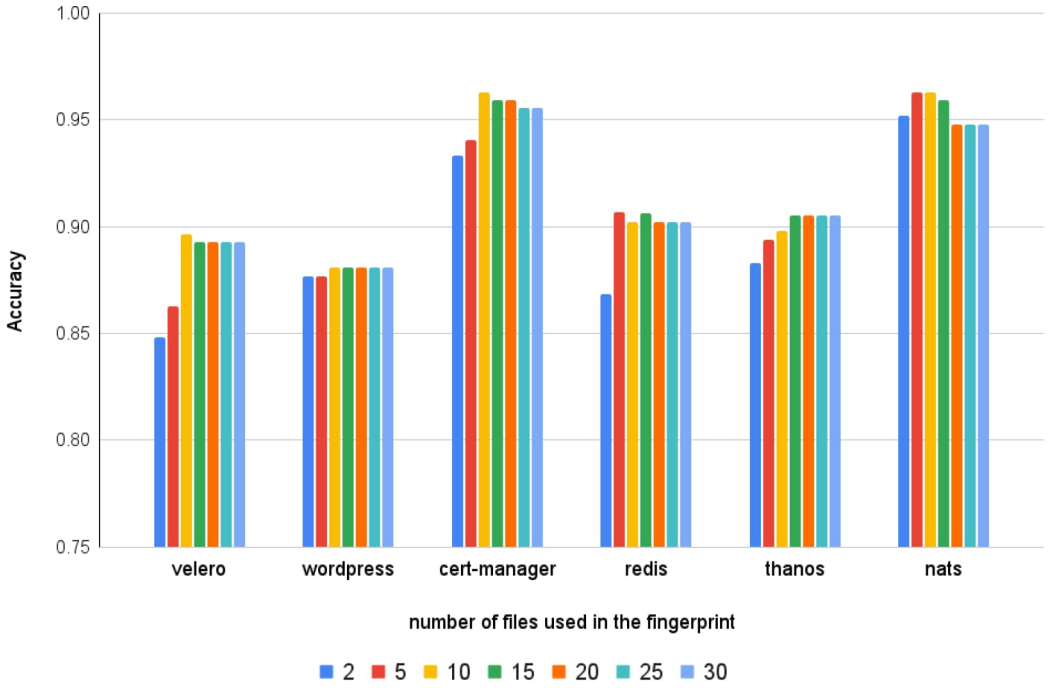

Figure 15: Initial analysis to decide the optimal number of files to use in a fingerprint.

To record each application deployment for two minutes was a compromise. To gain a complete understanding of the application it would be best to record for as long as possible. However, due to time constraints two minutes was deemed to be sufficient and in the initial analysis proved to provide accuracies of over 90% in some cases. This is indirectly visualized in Figure 15 as each created fingerprint consisted of several two-minute-long PCAP files.

The Helm charts are picked from Artifact Hub as it contains a wide variety of applications. 20 Helm charts were picked from the Artifact Hub repository. Appendix B1 lists all the chosen applications. To be able to suit this thesis, the applications should, in general, have at least two Pods so that some Pod-to-Pod or Service-to-Service communication can be observed. The only exception was Grafana which, despite only containing one Pod, produced network traffic that made the different versions distinguishable. Grafana was decided to be kept in the dataset to showcase that, at least in this case, it is possible to classify versions only by generating traffic from one Pod.

Each application needed to be first deployed, and some traffic needed to be collected to observe if it produced any meaningful network traffic. This increased the time to collect the data and to be able to complete the thesis in time, only 20 applications were chosen in the end. Considering each deployment is recorded for two minutes, it takes 60 minutes (30 x 2) to collect data for each version. This totals 10 hours for each application (10 versions x 1 hour per version). In practice, it takes more than 10 hours as the script needs to wait for the Pods to be ready. The resulting output is 300 PCAP files, each corresponding to a singular version deployment.

The dataset is published on Zenodo [28]. It contains an accurate list of used versions as well as allowing the reader to inspect the individual packets and results.

6 Results

This section starts by providing insights into both the collected data and the fingerprint comparison outputs. Afterward, the accuracy of the classifier is analyzed and validated. The classifier analysis is split into two parts: classifying all versions and classifying major versions only.

6.1 Data analysis

In this section, the statistical characteristics and protocol usage patterns derived from the application deployment data are analyzed. The analysis provides insights into the volume of network traffic, the diversity of communication protocols, and notable anomalies observed during deployments. Key metrics such as packet counts, byte volumes, and protocol distributions are summarized, highlighting variations between different applications and deployments.

Table 4 lists statistical insights from the dataset. On average, a single application deployment for 2 minutes resulted in around 11 636 packets and 3 481 070 bytes being transmitted across 8 different addresses. Maximum packets observed from a single application deployment were from Milvus. It produced 85480 packets in 2 minutes. Minimum packets observed from a single deployment came from Nats, where 0 packets were sent. This proved to be an anomaly, and no other application had any deployments that produced 0 packets.

Table 4: Summary of network traffic statistics from application deployments.

Statistic	**Value**
Average packets per deployment	11 636
Average bytes per deployment	3 481 070
Average addresses involved	8
Max packets in a deployment (Milvus)	85 480
Min packets in a deployment (Nats)	0

Table 5 provides an overview of the observed protocols and their distribution. Across all the application deployments, 22 different protocols were recorded. However, the definition of a protocol does not map 1:1 with the traditional network communication protocols. Refer to Section 4.3.1 for more details on the protocol definition. On average, 8.7 protocols were used per application. The minimum number of protocols observed in a single application deployment was 3, while the maximum was 16.

All applications produced HTTP and TCP traffic. By far, the most common protocol was TCP, with an average of approximately 6,050 packets per deployment. This high count reflects only the highest-level protocol detected by *pyshark*, as many application-layer protocols (e.g., HTTP) operate over TCP. HTTP, the second-most prevalent protocol, generated an average of 151 packets per deployment.

The "DATA-TEXT-LINES" protocol, often assigned to the unstructured text or undetected higher-layer protocols, was the next most common with an average of 51 packets, appearing in 70% of the deployments. DNS and TLS followed, observed

in 80% and 60% of deployments, producing an average of 744 and 457 packets respectively. Meanwhile, malformed packets, categorized as "WS.MALFORMED," were observed in 55% of deployments, with an average of 7 packets.

Rarer protocols included ESP, GRYPHON, SMPP, and SSH, each observed in only 5% of deployments (a single application). These produced an average of just 1 packet per deployment. Similarly, ENIP and CQL were less frequent, observed in 10% and 15% of deployments, respectively. Protocols like PGSQL and MYSQL, tied to specific applications, were observed in 30% and 20% of deployments, with average packet counts of 1,310 and 1,281. Other rare protocols, such as MEDIA and IRC, produced similarly low packet averages and appeared in fewer than 30% of deployments.

Table 5: Protocol statistics across application deployments. Each number is rounded up to the nearest integer.

Protocol	**Average Packets**	**Observation Rate (%)**
TCP	6 050	100
HTTP	151	100
DATA-TEXT-LINES	51	70
DNS	744	80
TLS	457	60
WS.MALFORMED	7	55
AMS	2	50
JSON	18	50
PMPROXY	3	45
ICMP	6	45
PGSQL	1 310	30
IRC	2	30
MYSQL	1 281	20
MEDIA	7	20
CQL	19	15
ECAT_MAILBOX	1	35
ENIP	1	10
ESP	1	5
GRYPHON	1	5
SMPP	1	5
SSH	1	5

6.2 Fingerprint comparison analysis

This section goes over some of the key details discovered when comparing PCAP files to a fingerprint from the same version and fingerprints from different versions. Furthermore, fingerprint comparison result bias is addressed.

When comparing a PCAP file to a fingerprint from the same version, the average number of *relevant packets* was 34. The minimum was 0, and most of the applications had 0 as the minimum value except for three applications (Hazelcast, Jaeger, and Prometheus). The maximum number of *relevant packets* from a single application deployment was recorded from Hazelcast where a single comparison resulted in 2034 *relevant packets*. The average percentage of trivial packets was 99.5%.

When comparing a PCAP file to a fingerprint from a different version, the average number of *relevant packets* was 870. The minimum was 0 and all the applications, except four, had at least one comparison where the minimum was 0. The four applications in question were Hazelcast, Jaeger, Prometheus, and Dagster. The maximum number of *relevant packets* was recorded in Netbox where 17 303 packets were marked as relevant. The average percentage of trivial packets was 89.5%.

There were a lot of the same packets within the *relevant packets*, i.e., packets where the payload matched exactly with one another. If only unique packets were considered, the number of average *relevant packets* decreased to 19 for the same version comparison and 317 for different version comparison.

Table 6 lists the above-mentioned statistics.

Table 6: Summary of *relevant packets* and trivial packet percentages when comparing PCAP files to fingerprints.

Comparison Type	**Same Version**	**Different Version**
Average Relevant Packets	34	870
Average Relevant Packets	19	371
Minimum Relevant Packets	0	0
Applications with Minimum = 0	All except 3	All except 4
Maximum Relevant Packets	2034	17,303
Average Trivial Packet Percentage	99.5%	89.5%

Note that if comparing a fingerprint to a PCAP file that originated from a different version and the result is 0 *relevant packets*, it is impossible to say if the PCAP file is from the same or different version. This means that there were no new, missing, or different packets.

Each application version fingerprint is compared to ten versions. One of the versions is the same as the fingerprint's version and the other nine are different. Each version has 15 PCAPs from different deployment instances, and the fingerprint is compared against all of them individually. Note that 30 PCAPs were recorded per version but 15 of those were used to create the fingerprint. The ones that were used to create the fingerprint need to be omitted from the comparison. As a result, the fingerprint comparison yields 150 rows of input to the classifier (per version fingerprint). Out of the 150 rows, 135 are comparison results between different versions and 15 are with the same versions as the fingerprint. That is there are 135 (90%) negative labels and 15 (10%) positive labels. This is a significant bias towards the negative labels, which may influence the classifier's performance.

6.3 Version detection results (all versions)

The overall accuracy is the total correct guesses divided by the total guesses. It gives a good overview of the performance of the classifier. Figure 16 shows the accuracy for each application. The average accuracy is around 95.3%, and the accuracy does not go below 90%. The lowest accuracy is from ArgoCD with an accuracy of 90.2%, and the highest accuracy is from Dagster with an accuracy of 99.8%.

$$Accuracy = \frac{CorrectGuesses}{TotalGuesses}$$

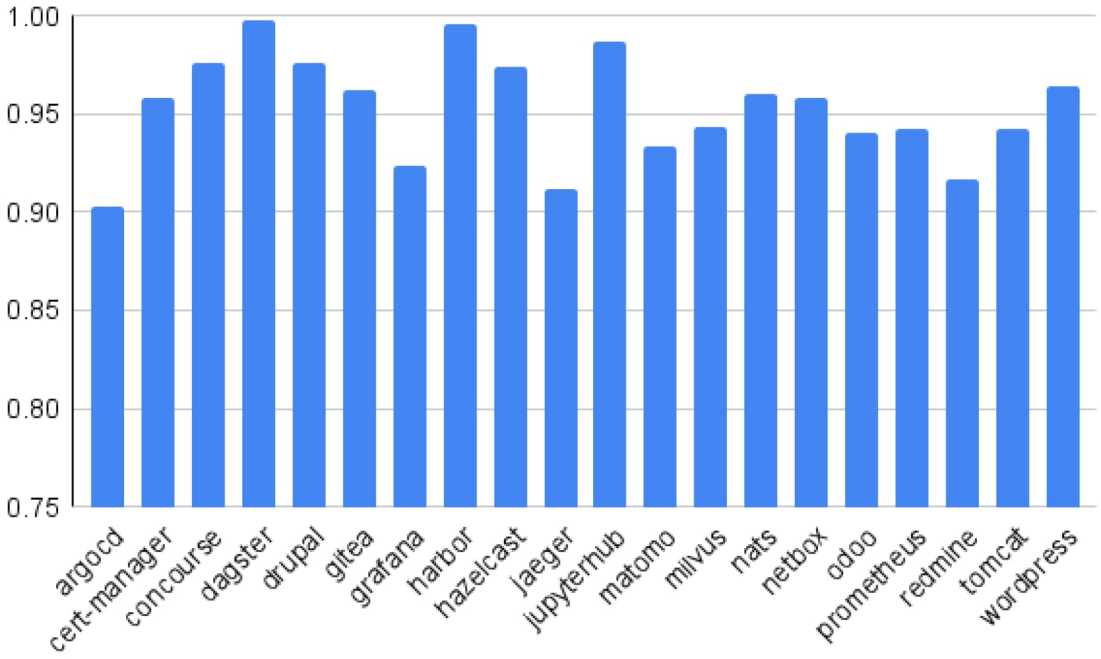

Figure 16: Version classification accuracy on each application (all versions).

However, to evaluate the accuracy in more detail, especially with a biased dataset, one needs to observe the true precision and recall as well as false precision and recall. The true precision, in this context, describes how precisely a PCAP file that originated from the same version, was classified correctly. The false precision describes how precisely a PCAP file that was not from the same version, was classified correctly. If both the true and false precision are high, it is a good indication that the classifier was able to find a distinction between the true and false labels. In this thesis, there are more false labels than positive ones. The classifier could easily achieve relatively high false precision just by guessing each entry as false and therefore achieve high overall accuracy. Therefore, a good classifier needs to have a high precision for both true and false labels. However, if the classifier makes only a single prediction on a positive label and gets it correctly, it effectively gets a 100% true precision. Therefore, to complement the analysis, one needs to observe the recall values as well.

$$TruePrecision = \frac{TruePositives(TP)}{TruePositives(TP) + FalsePositives(FP)}$$

The true recall, in this context, describes how many PCAP files from the same version were classified correctly divided by the number of all positive labels. The false recall, in this context, describes how many PCAP files from the different versions were classified correctly divided by the number of negative labels. In summary, the recall describes how completely the classifier was able to find the label in question.

$$TrueRecall = \frac{TP}{TP + FN}$$

Figure 17 shows the true precision and recall as well as the false precision and recall for each application. The false precision and recall are relatively high. The average false precision is 96.4%, and the average false recall is 98.5%. The true precision and recall are relatively low. The average true precision is 69.7%, and the average true recall is 66.7%. Some applications achieve high precision and recall, the highest being 100% true precision for Dagster and 98.3% true recall for Harbor. However, some applications achieve low precision and recall. The lowest observed true precision was 19.4% in Redmine, and the lowest observed true recall was 12.5% in Jaeger.

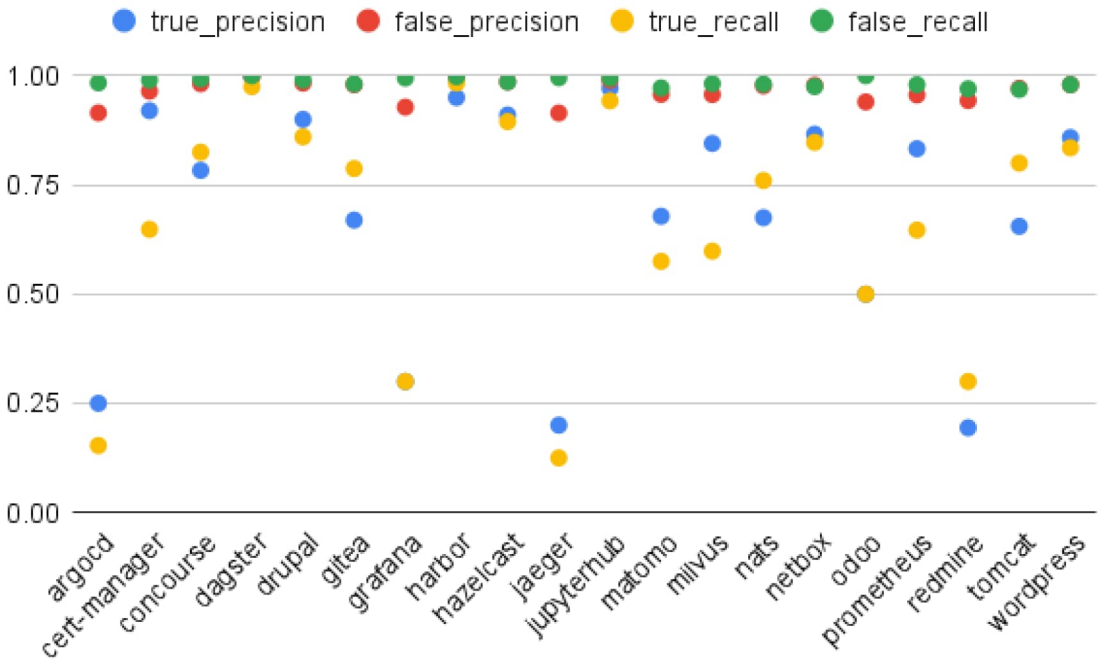

Figure 17: True and false recall and precision values per application (all versions).

The f1-score combines the recall and precision values. It is calculated by multiplying the precision and recall together and then further multiplying it by two. This

value is then divided by the sum of the precision and recall. The f1-score is simply not an average of the precision and recall but a "harmonic mean" which weights the score to the lower value. Otherwise, "cheating" in either recall or precision values by maximizing them at 100% would still result in at least 50% f1-score.

$$TrueF_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

Figure 18 illustrates the f1-scores per application. This is in line with the precision and recall values observed previously. The false f1-score is relatively high, on average 97.1%. The lowest value is 94.7% in ArgoCD. The true f1-score is relatively low, on average 61.2%. The lowest value is 14% in Jaeger, and the highest value is 95.5% in Dagster.

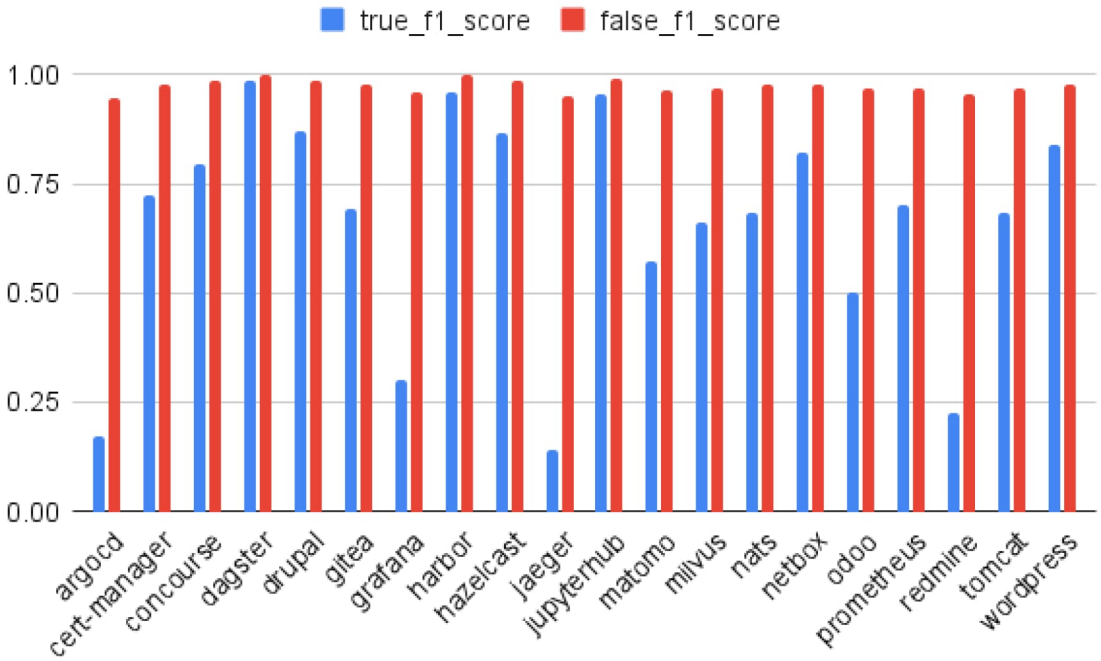

Figure 18: True and false f1-scores per application (all versions).

The results of the version detection analysis reveal important insights into the performance and limitations of the classifier. Overall, the results imply that the framework can distinguish versions apart. However, the accuracy is inconsistent especially when considering the true f1-scores. The framework can classify negative labels accurately, but true f1-scores vary greatly between applications. This imbalance can be attributed to the dataset bias as described in Section 6.2.

6.4 Version detection results (major versions)

The previous results are achieved when trying to classify all the recorded versions, including major, minor, and patch versions. To improve the accuracy, in this section,

the classifier tries to only classify traffic between major versions. That is classify if a PCAP file originates from the same major version as the fingerprint or not. Note that Cert-manager, Dagster, Harbor, and Prometheus were omitted from this analysis as they were recorded within a single major version.

Figure 19 shows the overall accuracy of each application. The average accuracy is relatively the same at 95.9%. However, certain applications have increased accuracy whereas others have decreased accuracy. Noticeably four applications reached 100% accuracy (Concourse, Drupal, Jupyterhub, and Odoo). On the other hand, ArgoCD and Jaeger lost around 4% accuracy. However, the difference in accuracy is minuscule and can be attributed to the fact that the training set is slightly different. Training the classifier, especially with a relatively small dataset, will lead to slightly different results based on the input given to the classifier.

Tables 7 and 8 represent the confusion matrices for all ArgoCD versions and major ArgoCD versions, respectively. Noticeably in all versions, the number of positive labels is bigger by 5 labels and the number of negative labels is larger by 57 labels. Due to these differences, the overall difference in accuracy (4%) is not deemed relevant.

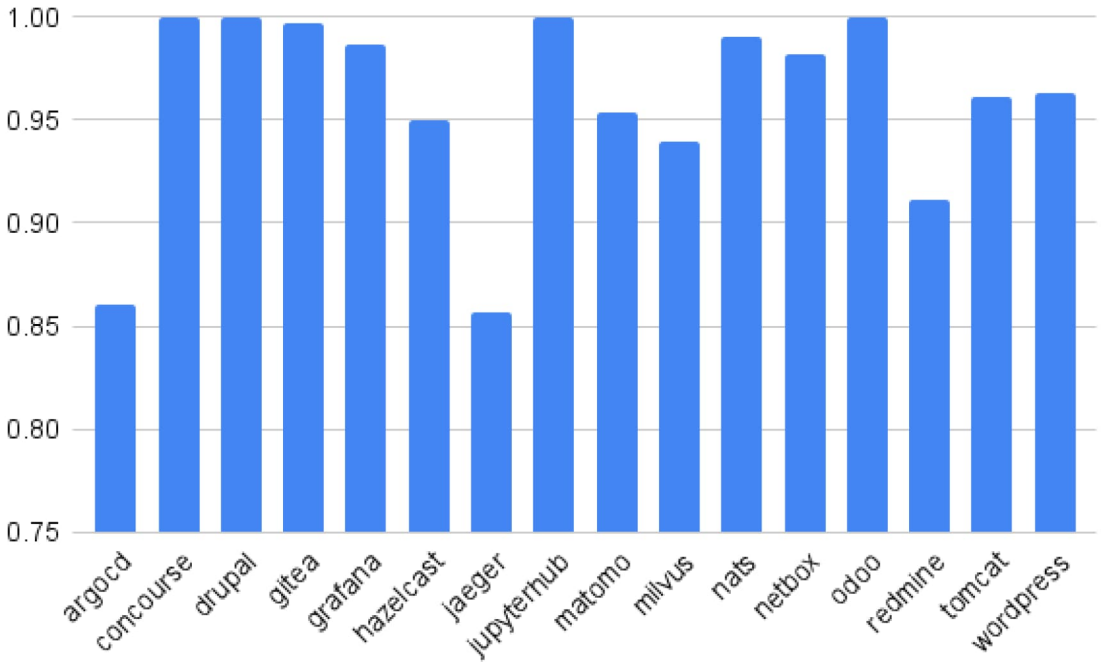

Figure 19: Accuracy per application (major versions only).

Table 7: ArgoCD all versions

	Predicted Positive	**Predicted Negative**
Actual Positive	7	34
Actual Negative	6	367

Table 8: ArgoCD major versions only

	Predicted Positive	**Predicted Negative**
Actual Positive	10	26
Actual Negative	23	293

Figure 20 showcases the true and false f1-scores per application when only considering major versions. While observing the f1-scores for major versions only, it can be noticed that the average true f1-score has increased noticeably to 79.8%. While the average false f1-score has stayed relatively the same at 97.6%. The same four applications that achieved 100% overall accuracy, naturally achieve 100% f1-scores as well. In general, applications that had over 50% true f1-scores with all versions, seem to now converge towards 100% accuracy. The applications that had relatively low true f1-scores have increased slightly but still remain relatively low.

There is a clear division between the true f1-scores, the score is either above 76% or below 33%. There is no middle ground. This division implies that the results yielded when comparing PCAP to a fingerprint define the accuracy more greatly than the model itself. If there is a relatively small number of *relevant packets* when comparing a PCAP to a fingerprint, it is difficult for the model to classify the version correctly.

The above is illustrated in Figure 21. The blue bar indicates the average true f1-score per application and the lines are the *relevant packets* (different packets, new packets, and missing packets) in relation to the average *relevant packets* across all applications. There is a noticeable trend where the number of *relevant packets* is low when the true f1-score is low. It can be seen, for example, with ArgoCD. ArgoCD's true f1-score is 17.3% and the number of *relevant packets* is only around 14% of the average amount. It should be noted that some applications still have relatively high true f1-score even though the *relevant packet* amount is below average. For example, Prometheus has a true f1-score of 70% but the *relevant packet* amount is only around 15% of the average. This indicates that in the right circumstances, there are individual packets that define a specific version well. Therefore, in some instances, only a small number of packets are needed to classify the versions correctly.

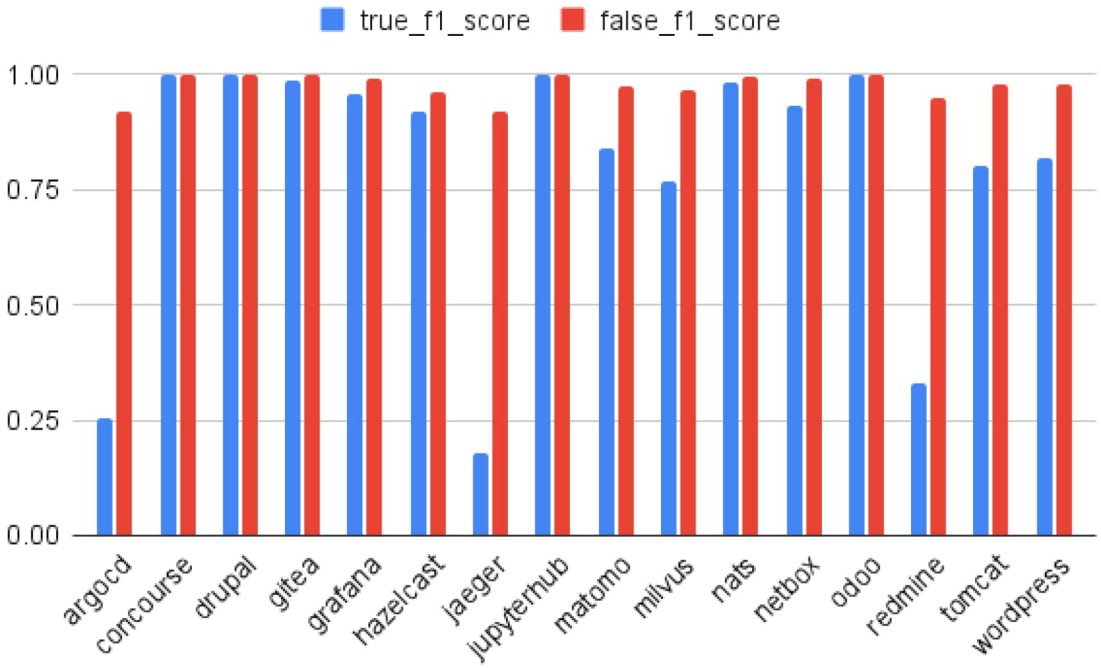

Figure 20: True and false f1-scores per application (major versions only).

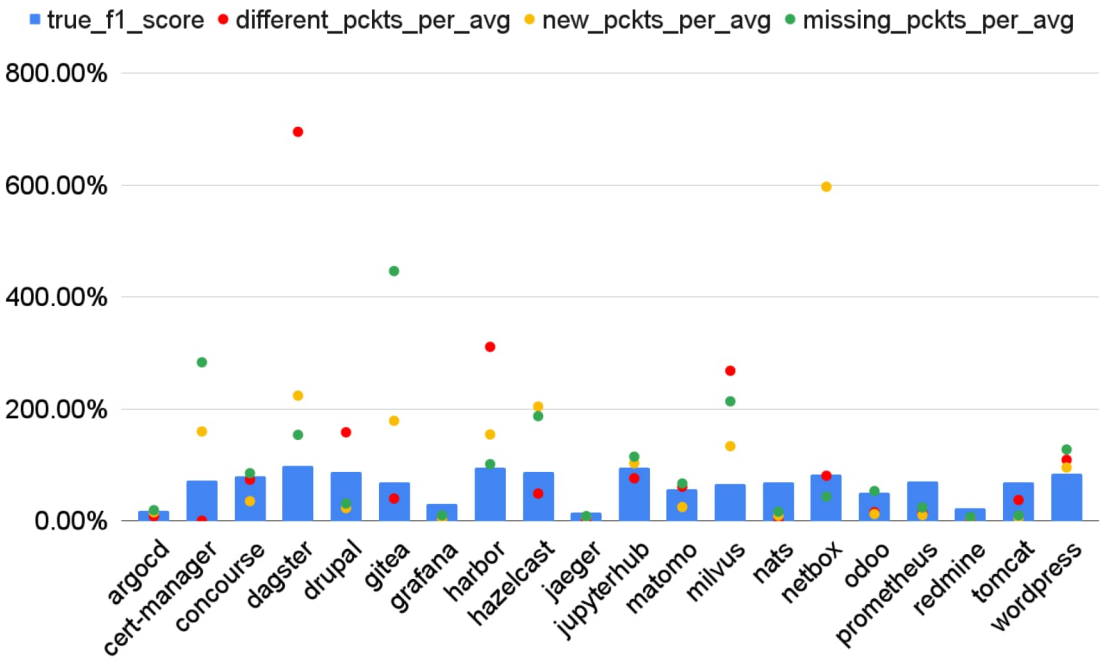

Figure 21: Average fingerprint comparison output and true f1-scores per each application (all versions).

7 Discussion

The findings of this research demonstrate that a version-sensitive approach to network traffic classification is both feasible and effective, at least within Kubernetes environments. By employing machine learning techniques and fingerprinting mechanisms, the framework successfully distinguished between application versions based on their unique network traffic patterns. This capability is especially important in environments where precise version tracking can significantly enhance security and operational oversight. Furthermore, there are strong implications that the capability extends beyond version classification and more towards detecting minute anomalies within network traffic. To be able to detect different versions requires being able to detect minor changes in the observed patterns.

The version classification accuracy proved to be relatively high with a relatively low amount of information. Only the first two minutes of each application were recorded which, in some cases, provided only around 300 packets per version. This is a strong contrast to flow-level statistical classification where a singular flow itself can contain 300 packets while achieving lower accuracy in general. Furthermore, the classification accuracy was relatively accurate even with the dataset containing significant bias.

The framework operates in a passive, application-agnostic mode providing several benefits. Achieving good classification accuracy without having to understand the application or the underlying network traffic is a strong tool for network administrators and analysts. Gaining knowledge in a certain domain can require significant effort. In a complex network environment, this task can be exceptionally difficult. Furthermore, the framework being passive means it can simply capture the network traffic and does not interfere with the running applications by, for example, probing the applications or generating user traffic. Taking an active role would require more domain knowledge and time to implement the probes.

The fingerprinting mechanism is the main component in distinguishing the versions apart. The machine learning classifier is implemented to free the users from making decisions on how to split the data and what thresholds should be set. The fingerprinting mechanism can exist without the classifier but not vice versa. The classifier could be implemented with classical methods like setting the thresholds manually. Furthermore, if the fingerprint is not effective at capturing the defining features of a specific version, the comparison cannot produce meaningful inputs for the classifier. This was further indicated in the results when the classifier performed relatively poorly on all versions as the fingerprint comparison's output lacked distinguishable features.

Several challenges emerged during this study. The main challenge was that in some cases there simply was not enough information to distinguish versions from each other. This was apparent when comparing all versions versus comparing major versions. The fingerprinting mechanism is strict and as such if no distinguishable information is received when comparing a PCAP file against the fingerprint, no information sets the versions apart. However, it does not mean that it is impossible to distinguish certain versions from each other, it only means that the fingerprinting

implementation was not able to achieve that.

The data collection process was straightforward, even though time-consuming, and its completion represents a meaningful contribution to the field. Providing the dataset is valuable in itself, as it enables further research and experimentation. However, it is difficult to evaluate the dataset. For example, how well does it match other real-world scenarios? On one hand, the dataset is collected from an environment that contains realistic Kubernetes deployments. On the other hand, the data is only collected from the beginning of the application's lifecycle and does not contain organic interactions from real users.

Even though making the framework application agnostic and relatively automated has strong advantages, it has its drawbacks as well. As mentioned before, this does not require much domain knowledge of network traffic and networked systems. However, understanding what packets make the differences between versions and debugging the framework requires domain knowledge. Furthermore, implementing and maintaining the framework requires some domain knowledge.

Despite these challenges, the implications of this research are important. It demonstrates how version-sensitive classification can contribute to network analysis by distinguishing versions apart.

8 Future work

While this research has provided valuable insights, it also opens up several prospects for further research. It is difficult to draw further conclusions about the dataset's effectiveness and how well it compares to other scenarios. Therefore, it would be crucial to evaluate the framework's performance using another dataset, possibly from a completely different environment as well. The framework describes the data collection process using Kubernetes but it is only one avenue for collecting the data. The fingerprinting process is not concerned about where the data originated as long as it is in PCAP format. Extending the research into different domains such as IoT or edge computing could provide insights into the framework's relevance in other networked environments.

Even though the framework is used in this thesis to distinguish versions apart, the fingerprinting technique can be used in other detection tasks as well. For example, recording 30 days of internal network traffic, generating a fingerprint from the first 15 days, and then comparing if the last 15 days differ notably from the first 15 days. The main idea behind this thesis is to differentiate subtle differences in network captures. Version detection is used as a means to realize and evaluate this idea.

Deploying the framework in a production environment would be another crucial step for evaluating its efficiency. Furthermore, analyzing the full lifecycle of an application would complement the evaluation. This thesis was only collecting data in a local development environment and only capturing the first two minutes of an application. Production applications might involve millions of users and the applications can run for months or years. This would also introduce the framework into a much more complex environment.

The framework should also be exposed to more encrypted data. The collected dataset in this thesis contained relatively low amounts of encrypted data and therefore made the payload inspection especially viable. However, modern network environments often use protocols with encryption and the data can even be encapsulated by VPN. If the framework is deemed to perform in more encrypted environments, the implications would be significant. Also, if the framework cannot perform under these conditions, the data could be collected at the point of the network where the packets are decrypted. This would allow the framework to access the decrypted packets. However, this approach might not be possible in some cases, and it might also violate user privacy.

It was considered to compare the framework to some existing research, but relevant research was not found. Therefore, any comparison with seemingly closely related tools would not have been meaningful. If similar research appears in the future, it would be beneficial for the whole network traffic classification field to provide meaningful comparisons between different version-sensitive frameworks. Alongside the framework, the dataset should also be evaluated and tested with other classification tools.

The main area of improvement involves the framework's performance. The framework has only been tested with a relatively small dataset. Even with the small dataset, the fingerprinting took, in some cases, an hour on laptop hardware. The

biggest bottleneck is the string comparison process. Comparing each character of each packet together is computationally intensive. To improve the performance, different optimizations could be considered. Also, other string comparison algorithms could be explored such as finding the largest common substring among all similar packets. Alternative algorithms might improve the performance or accuracy of the fingerprint.

In summary, the future work can be split into two parts: evaluating the framework and improving the framework. Evaluating the framework would include introducing the framework into more complex network environments such as a production environment with encrypted traffic. Improving the framework consists mainly of improving the performance but can also entail improving the accuracy of the framework by exploring alternative algorithms.

9 Concluding remarks

This thesis presented a novel framework for version-sensitive network traffic classification, addressing a gap in the field of network traffic classification. By focusing on application versions, the study highlights the potential for more granular and accurate traffic classification methods that can adapt to the evolving landscape of networked applications.

The findings indicate that a combination of machine learning and fingerprinting techniques can effectively differentiate between application versions, even in real-life environments like Kubernetes. However, the research also underscores the need for continued innovation to address challenges such as dataset limitations, scalability, and encrypted traffic handling.

In conclusion, this research contributes to the broader goal of enhancing network security and operational efficiency through improved traffic classification methods. Furthermore, the research provided a dataset to decrease the scarcity of public network traffic datasets. By laying the groundwork for future advancements in version-sensitive classification, this study paves the way for more robust and adaptable solutions in the ever-changing domain of networked systems.

References

[1] Iman Akbari et al. “A Look Behind the Curtain: Traffic Classification in an Increasingly Encrypted Web”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 5.1 (Feb. 2021). Association for Computing Machinery. DOI: 10.1145/3447382.

[2] Helm Authors. *Helm*. en-us. 2024. URL: https://helm.sh/ (visited on 09/05/2024).

[3] The Kubernetes Authors. *Cluster Architecture*. en. Oct. 2024. URL: https://kubernetes.io/docs/concepts/architecture/ (visited on 10/16/2024).

[4] The Kubernetes Authors. *Ingress*. en. Sept. 2024. URL: https://kubernetes.io/docs/concepts/services-networking/ingress/ (visited on 10/16/2024).

[5] The Kubernetes Authors. *Liveness, Readiness, and Startup Probes*. en. Nov. 2024. URL: https://kubernetes.io/docs/concepts/configuration/liveness-readiness-startup-probes/ (visited on 10/19/2024).

[6] The Kubernetes Authors. *Overview*. en. Sept. 2024. URL: https://kubernetes.io/docs/concepts/overview/ (visited on 10/16/2024).

[7] The Kubernetes Authors. *Pod Lifecycle*. en. Nov. 2024. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/ (visited on 10/17/2024).

[8] The Kubernetes Authors. *Pods*. en. Nov. 2024. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 10/17/2024).

[9] The Kubernetes Authors. *Service*. en. Nov. 2024. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 10/17/2024).

[10] The Kubernetes Authors. *Services, Load Balancing, and Networking*. en. Sept. 2024. URL: https://kubernetes.io/docs/concepts/services-networking/ (visited on 10/17/2024).

[11] Ahmad Azab et al. “Network traffic classification: Techniques, datasets, and challenges”. In: *Digital Communications and Networks* 10.3 (June 2024). Elsevier, pp. 676–692. DOI: 10.1016/j.dcan.2022.09.009.

[12] Ahmad Azab et al. “Skype Traffic Classification Using Cost Sensitive Algorithms”. In: *2013 Fourth Cybercrime and Trustworthy Computing Workshop*. IEEE, Mar. 2013, pp. 14–21. DOI: 10.1109/CTC.2013.11.

[13] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*. Vol. 1. MIT press, 2017.

[14] Gérard Biau and Erwan Scornet. “A random forest guided tour”. In: *Test* 25 (Apr. 2016). Springer Nature, pp. 197–227. DOI: 10.1007/s11749-016-0481-7.

[15] Bitnami. *WordPress 23.1.21*. en. Aug. 2024. URL: https://artifacthub.io/packages/helm/bitnami/wordpress (visited on 10/14/2024).

[16] Philip Branch and Jason But. “Rapid and generalized identification of packetized voice traffic flows”. In: *37th Annual IEEE Conference on Local Computer Networks*. IEEE, Feb. 2012, pp. 85–92. DOI: 10.1109/LCN.2012.6423690.

[17] Varun Chandola, Arindam Banerjee, and Vipin Kumar. “Anomaly detection: A survey”. en. In: *ACM Computing Surveys* 41.3 (July 2009). Association for Computing Machinery. DOI: 10.1145/1541880.1541882.

[18] F. Y. Edgeworth. “On Observations Relating to Several Quantities”. In: *Hermathena* 6.13 (1887). Trinity College Dublin, pp. 279–285. URL: https://www.jstor.org/stable/23036355 (visited on 09/03/2024).

[19] Alessandro Finamore et al. “KISS: Stochastic Packet Inspection Classifier for UDP Traffic”. In: *IEEE/ACM Transactions on Networking* 18.5 (Jan. 2010). IEEE, pp. 1505–1515. DOI: 10.1109/TNET.2010.2044046.

[20] The OWASP Foundation. *CRLF Injection*. en. May 2022. URL: https://owasp.org/www-community/vulnerabilities/CRLF_Injection (visited on 11/14/2024).

[21] The OWASP Foundation. *Doubly freeing memory*. en. May 2022. URL: https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory (visited on 11/14/2024).

[22] Mikel Galar et al. “A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.4 (Aug. 2012). IEEE, pp. 463–484. DOI: 10.1109/TSMCC.2011.2161285.

[23] Dor Green. *pyshark*. Dec. 2024. URL: https://github.com/KimiNewt/pyshark (visited on 10/23/2024).

[24] The Tcpdump Group. *libpcap*. Dec. 2024. URL: https://github.com/the-tcpdump-group/libpcap (visited on 09/04/2024).

[25] Patrick Haffner et al. “ACAS: automated construction of application signatures”. In: *Proceedings of the 2005 ACM SIGCOMM Workshop on Mining Network Data*. MineNet ’05. Association for Computing Machinery. Association for Computing Machinery, Aug. 2005, pp. 197–202. DOI: 10.1145/1080173.1080183.

[26] Guy Harris and Michael Richardson. *PCAP Capture File Format*. Internet Draft 04. Internet Engineering Task Force, Aug. 2024. URL: https://datatracker.ietf.org/doc/draft-ietf-opsawg-pcap/04/ (visited on 10/10/2024).

[27] Aleksi Hirvensalo. *Hirvensaloa/version-detection-framework: Version-sensitive network traffic classification framework*. en. Dec. 2024. URL: https://github.com/Hirvensaloa/version-detection-framework (visited on 12/10/2024).

[28] Aleksi Hirvensalo. *Kubernetes application PCAPs*. eng. Dec. 2024. DOI: 10.5281/zenodo.14338912.

[29] Yung-Fa Huang et al. “Traffic Classification of QoS Types Based on Machine Learning Combined with IP Query and Deep Packet Inspection”. In: *2020 14th International Conference on Signal Processing and Communication Systems (ICSPCS)*. IEEE. Feb. 2020. DOI: 10.1109/ICSPCS50536.2020.9310061.

[30] IANA. *Service Name and Transport Protocol Port Number Registry*. June 2024. URL: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml (visited on 09/23/2024).

[31] IBM. *What is a Decision Tree?* en. Nov. 2021. URL: https://www.ibm.com/topics/decision-trees (visited on 10/26/2024).

[32] IBM. *What Is Deep Learning?* en. June 2024. URL: https://www.ibm.com/topics/deep-learning (visited on 09/28/2024).

[33] IBM. *What Is Machine Learning (ML)?* en. Sept. 2021. URL: https://www.ibm.com/topics/machine-learning (visited on 09/19/2024).

[34] IBM. *What Is the OSI Model?* en. June 2024. URL: https://www.ibm.com/think/topics/osi-model (visited on 11/27/2024).

[35] Danial Javaheri et al. “Fuzzy logic-based DDoS attacks and network traffic anomaly detection methods: Classification, overview, and future perspectives”. In: *Information Sciences* 626 (May 2023). Elsevier, pp. 315–338. DOI: 10.1016/j.ins.2023.01.067.

[36] Pratibha Khandait, Neminath Hubballi, and Bodhisatwa Mazumdar. “Efficient Keyword Matching for Deep Packet Inspection based Network Traffic Classification”. In: *2020 International Conference on COMmunication Systems NETworkS (COMSNETS)*. IEEE, Mar. 2020, pp. 567–570. DOI: 10.1109/COMSNETS48256.2020.9027353.

[37] Chrissy Kidd and Muhammad Raza. *Virtual Machines (VMs) vs Containers: What's The Difference?* en-US. July 2020. URL: https://www.bmc.com/blogs/containers-vs-virtual-machines/ (visited on 10/17/2024).

[38] Jan Kohout et al. “Learning communication patterns for malware discovery in HTTPs data”. In: *Expert Systems with Applications* 101 (July 2018). Elsevier, pp. 129–142. DOI: https://doi.org/10.1016/j.eswa.2018.02.010.

[39] Ákos Kovács. “Comparison of different Linux containers”. In: *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, July 2017, pp. 47–51. DOI: 10.1109/TSP.2017.8075934.

[40] Mohammad Lotfollahi et al. “Deep packet: A novel approach for encrypted traffic classification using deep learning”. In: *Soft Computing* 24.3 (May 2020). Springer Nature, pp. 1999–2012. DOI: /10.1007/s00500-019-04030-2.

[41] Marko Lukša and Kevin Conner. *Kubernetes in action, Second Edition*. Simon and Schuster, Feb. 2020.

[42] Navid Malekghaini et al. “Deep learning for encrypted traffic classification in the face of data drift: An empirical study”. In: *Computer Networks* 225 (Apr. 2023). Elsevier. DOI: https://doi.org/10.1016/j.comnet.2023.109648.

[43] Ibomoiye Domor Mienye and Yanxia Sun. "A Survey of Ensemble Learning: Concepts, Algorithms, Applications, and Prospects". In: *IEEE Access* 10 (Sept. 2022). IEEE, pp. 99129–99149. DOI: 10.1109/ACCESS.2022.3207287.

[44] Andrew W. Moore and Konstantina Papagiannaki. "Toward the Accurate Identification of Network Applications". In: *Passive and Active Network Measurement*. Springer, 2005, pp. 41–54. DOI: 10.1007/978-3-540-31966-5_4.

[45] M. Rosario Oliveira et al. "Do we need a perfect ground-truth for benchmarking Internet traffic classifiers?" In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE. Aug. 2015, pp. 2452–2460. DOI: 10.1109/INFOCOM.2015.7218634.

[46] Fannia Pacheco et al. "Towards the Deployment of Machine Learning Solutions in Network Traffic Classification: A Systematic Survey". In: *IEEE Communications Surveys Tutorials* 21.2 (Nov. 2019). IEEE, pp. 1988–2014. DOI: 10.1109/COMST.2018.2883147.

[47] Tom Preston-Werner. *Semantic Versioning 2.0.0*. en. 2024. URL: https://semver.org/ (visited on 10/14/2024).

[48] Prometheus. *Prometheus - Monitoring system & time series database*. en. 2024. URL: https://prometheus.io/ (visited on 10/22/2024).

[49] Muhammad Shaheem Raza et al. "High Performance DPI Engine Design for Network Traffic Classification, Metadata Extraction and Data Visualization". In: *2024 5th International Conference on Advancements in Computational Sciences (ICACS)*. IEEE, Mar. 2024. DOI: 10.1109/ICACS60934.2024.10473274.

[50] Shahbaz Rezaei, Bryce Kroencke, and Xin Liu. "Large-Scale Mobile App Identification Using Deep Learning". In: *IEEE Access* 8 (Dec. 2020). IEEE, pp. 348–362. DOI: 10.1109/ACCESS.2019.2962018.

[51] Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. "Accurate, scalable in-network identification of p2p traffic using application signatures". In: *Proceedings of the 13th International Conference on World Wide Web*. WWW '04. Association for Computing Machinery, May 2004, pp. 512–521. DOI: 10.1145/988672.988742.

[52] Muhammad Sameer Sheikh and Yinqiao Peng. "Procedures, Criteria, and Machine Learning Techniques for Network Traffic Classification: A Survey". In: *IEEE Access* 10 (June 2022). IEEE, pp. 61135–61158. DOI: 10.1109/ACCESS.2022.3181135.

[53] Peter Sollich and Anders Krogh. "Learning with ensembles: How overfitting can be useful". In: *Advances in Neural Information Processing Systems*. Vol. 8. MIT Press, 1995. URL: https://proceedings.neurips.cc/paper_files/paper/1995/file/1019c8091693ef5c5f55970346633f92-Paper.pdf.

[54] Yan-yan Song and Ying Lu. "Decision tree methods: applications for classification and prediction". In: *Shanghai Archives of Psychiatry* 27.2 (Apr. 25, 2015). Shanghai Municipal Bureau of Publishing, pp. 130–135. DOI: 10.11919/j.issn.1002-0829.215044.

[55] The Artifact Hub Authors. *Artifact Hub*. en. 2024. URL: https://artifacthub.io/ (visited on 09/05/2024).

[56] Silvio Valenti et al. "Reviewing Traffic Classification". In: *Data Traffic Monitoring and Analysis: From Measurement, Classification, and Anomaly Detection to Quality of Experience*. Springer, 2013, pp. 123–147. DOI: 10.1007/978-3-642-36784-7_6.

[57] Barry de Ville. "Decision trees". In: *WIREs Computational Statistics* 5.6 (Oct. 2013). Wiley Online Library, pp. 448–455. DOI: 10.1002/wics.1278.

[58] Yu Wang, Yang Xiang, and Shunzheng Yu. "Internet Traffic Classification Using Machine Learning: A Token-based Approach". In: *2011 14th IEEE International Conference on Computational Science and Engineering*. IEEE, 2011, pp. 285–289. DOI: 10.1109/CSE.2011.58.

[59] Wireshark. *tshark(1) Manual Page*. 2024. URL: https://www.wireshark.org/docs/man-pages/tshark.html (visited on 10/23/2024).

[60] Jun Zhang et al. "Network Traffic Classification Using Correlation Information". In: *IEEE Transactions on Parallel and Distributed Systems* 24.1 (Mar. 2013), pp. 104–117. DOI: 10.1109/TPDS.2012.98.

[61] Jingjing Zhao et al. "Network traffic classification for data fusion: A survey". In: *Information Fusion* 72 (Aug. 2021). IEEE, pp. 22–47. DOI: https://doi.org/10.1016/j.inffus.2021.02.009.

[62] Zhi-Hua Zhou. *Machine learning*. Springer Nature, Aug. 2021.

A Fingerprint comparison CSV

Table A1: Aggregated CSV file fields. The CSV file is a result of a comparison between a PCAP file and a fingerprint.

Field	**Explanation**
filename	The name of the PCAP file that contains the *relevant packets*. (*<fingerprint version>_<PCAP version>_<deployment number>.pcap*)
number_of_packets	Total count of *relevant packets*.
number_of_unique_packets	Number of unique *relevant packets*. Often the list of packets contains multiple packets that are exactly the same.
average_length	The average payload length of the packets.
number_of_new_packets	The number of new packets. Packets that were in the PCAP file but not in the fingerprint.
number_of_unique_new_packets	The count of unique new packets.
number_of_missing_packets	The number of packets marked as missing. Common packets in the fingerprint but not found in the PCAP file.
avg_change_in_payload_%	The average percentage change in payload compared to the fingerprint.
benign_packets_%	The percentage of packets found in both the PCAP file and the fingerprint, and no differences were detected. (Packets / Total packets)

B Application table

Table B1: List of the captured applications. The description field is provided by Artifact Hub [55].

Name	**App description**	**From Helm chart version**	**To Helm chart version**
ArgoCD	ArgoCD bundle with ingress and app of apps configuration.	4.9.8	7.6.1
Cert-manager	Cert-manager is a Kubernetes addon to automate the management and issuance of TLS certificates from various issuing sources.	1.7.3	1.16.1
Concourse	Concourse is a simple and scalable CI system.	16.1.24	17.3.1
Dagster	The data orchestration platform built for productivity.	1.0.17	1.9.2
Drupal	Drupal is one of the most versatile open source content management systems in the world.	17.1.2	21.0.1
Gitea	Gitea is a community managed lightweight code hosting solution written in Go.	1.1.0	2.3.22
Grafana	The leading tool for querying and visualizing time series and metrics.	9.6.2	11.3.26
Harbor	An open source trusted cloud native registry that stores, signs, and scans content.	1.13.1	1.14.3
Hazelcast	A distributed in-memory data grid for high-performance computing and caching.	3.10.3	5.10.1
Jaeger	Jaeger is a distributed tracing system. It is used for monitoring and troubleshooting microservices-based distributed systems.	0.62.0	3.3.2
Jupyterhub	Multi-user Jupyter installation.	3.3.0	4.0.0
Matomo	Matomo, formerly known as Piwik, is a real time web analytics program.	5.2.0	8.0.13
Milvus	Milvus is an open-source vector database built to power AI applications and vector similarity search.	5.6.3	9.0.6

Name	App description	From Helm chart version	To Helm chart version
Nats	A lightweight, open-source messaging system for distributed systems.	0.10.0	1.2.4
NetBox	IP address management and data center infrastructure management tool	4.3.14	6.1.4
Odoo	Odoo is an open source ERP and CRM platform, formerly known as OpenERP, that can connect a wide variety of business operations such as sales, supply chain, finance, and project management.	25.1.0	28.0.1
Prometheus	Prometheus is a monitoring system and time series database.	25.22.0	25.30.0
Redmine	Redmine is an open source management application. It includes a tracking issue system, Gantt charts for a visual view of projects and deadlines, and supports SCM integration for version control.	26.5.2	32.0.1
Tomcat	Apache Tomcat is an open-source web server designed to host and run Java-based web applications.	10.5.20	11.3.1
WordPress	A popular open-source content management system for building websites and blogs.	19.2.6	24.0.0

"From Helm chart version"-field and "To Helm chart version"-field in Appendix B1 refers to the used Helm chart version range. The smallest version is the oldest version used, and the biggest version is the most recent version used. Inside the version range, some versions are not utilized, especially if they are only patch version changes.