University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

IDAPN: Intelligent Detection of Anomalies in Programmable Networks

Cédric Ivo Lūchinger
Zurich, Switzerland
Student ID: 19-918-085

Supervisor: Dr. Alberto Huertas Celdrán and Dr. Muriel Figueredo Franco
Date of Submission: March 09, 2025

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich,

Signature of student

Abstract

Abnormales Verhalten in Netzwerken zu erkennen mittels maschinellen Lernens (ML) oder tiefen Lernens (DL) gehört zu den vielversprechenden Methoden Cyberangriffe zu erkennen. Dies auch aus dem Grund, dass Zero-Day-Angriffe oft nicht entdeckt werden können, da wir das Verhalten solcher Angriffe noch nicht kennen. Im Moment fehlt es an einem Ansatz, der nur die Ressourcen von einem Switch anschaut, die direkt aus dem Kern stammen in einem programmierbaren Netzwerk. Ein solcher Ansatz würde einen schnellen und einfachen Weg eröffnen, Angriffe in einem Netzwerk zu erkennen und zugleich sehr viele Informationen über dem Switch und seinen Zustand preiszugeben. Aus diesem Grund will diese Arbeit mittels dem erweitertem Berkeley Paketfilter (eBPF) die Metriken vom Switch direkt extrahieren. EBPF wurde ausgewählt, da es ein schneller und effizienter Weg ist Ressourcen direkt aus dem Linux Kern zu extrahieren. Weiter gilt es, mittels der extrahierten Daten einen Fingerabdruck zu erstellen und damit ML-Algorithmen zu trainieren. Die Experimentierumgebung muss dafür realitätsnah sein, um überhaupt Informationen direkt aus den Linux Kern extrahieren zu können. Aus diesem Grund wurde eine Docker-Umgebung ausgewählt. Die Resultate zeigen, dass man mit genug Fleiss und Entschlossenheit in der Lage ist, die Docker Container Metriken mittels eBPF auszulesen. Somit werden für weitere Forschungen die Möglichkeit eröffnet, direkt mit realen Kern Metriken zu arbeiten. Die Resultate bezüglich des intelligenten Aufdeckens von abnormalem Verhalten, zeigten, dass es möglich ist, Anomalien zu erkennen und das besonders Algorithmen mit Labeln gut performen. Bei leicht schwankende Paketgrössen in einem Netzwerk wurde k-Nächste Nachbaren (KNN) als bester Algorithmus befunden. Bei einer grossen Fluktuation der Paketgrössen hingegen performte Zufälliger Wald (RF) am besten.

To detect anomalous behavior, the use of Machine Learning (ML) and Deep Learning (DL) is regarded one of the most promising forms of detecting cyberattacks. This is because Zero-Day-Attacks can often go undetected with other tools. Currently, there is a lack of approaches that detect anomalous behavior using only the switch resource metrics directly from the kernel in a programmable network. Such an approach offers a fast and efficient way to detect attacks within a network, as one can gather a lot of information about the state of the switches. For this reason, the extended Berkeley Paket Filter (eBPF) is used in this work, as it can extract resources from the Linux kernel quickly and efficiently. Furthermore, the extracted data is used to generate a fingerprint and then train ML algorithms. The chosen environment must be realistic, as the metrics must be extracted directly from the kernel. For this reason, a Docker environment was built. The results show that with effort and determination one is able to extract the resource metrics via eBPF. This enables new possibilities for future research. Regarding intelligent anomaly detection, it shows that the supervised algorithms perform well in these cases. For small deviating packet sizes, k-Nearest Neighbors (KNN) performs best. In contrast, Random Forest (RF) performs best with high fluctuating packet sizes.

Acknowledgments

My sincerest gratitude goes to the supervisors, Dr. Alberto Huertas Celdrán and Dr. Muriel Figueredo Franco. Their support, time, and knowledge invested were of upmost importance to this thesis. I would also like to thank Matheus Saueressig for his support and expertise on the topic.

Further I am honored to have the opportunity to conduct my Bachelor's thesis at the University of Zurich with the Communication Systems Group (CSG), under the guidance of Prof. Dr. Burkhard Stiller. The skills and topics learned in this thesis will have a great impact on my future. I really appreciate the opportunity to work on such interesting topics. Additionally, I would like to sincerely thank all people who were part of this Bachelor Thesis.

Contents

Declaration of Independence **i**

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**

2 Fundamentals **3**

2.1 Programmable Networks 3

2.1.1 Software-defined Networking 4

2.2 Fingerprinting 4

2.2.1 Creating and detecting fingerprints 4

2.3 Anomaly detection 6

2.3.1 Types of Anomaly 7

2.4 Related Work 7

2.4.1 P4-enabled switches 7

2.4.2 Monitoring with eBPF 9

2.4.3 Docker Container Anomaly Monitoring 11

2.4.4 Intelligent and behavioral-based detection of malware in IoT spectrum sensors 12

3 Design & Implementation **15**

3.1 IDAPN Overview 15
3.2 Environment 16
3.3 Data Extraction 20
3.4 Simulation 22
3.5 Intelligent Anomaly Detection 22

4 Evaluation and Results **27**

4.1 Features Extraction 27
4.2 ML Evaluation 28
4.3 Discussion and Limitations 31

5 Final Considerations **35**

5.1 Summary 35
5.2 Conclusions 37
5.3 Future Work 37

Bibliography **38**

Abbreviations **43**

List of Figures **44**

List of Tables **45**

A Contents of the Repository **49**

A.1 Installation 49
A.2 Operation 51

Chapter 1

Introduction

Today OpenFlow is a very prominent protocol for Software-defined Networking (SDN). Nevertheless, it has some disadvantages regarding the complexity and programmability of its components. Programming Protocol-independent Packet Processors (P4) are crucial in circumventing those, since it allows to fully program a switch that is P4-enabled [21]. There have been some successful anomaly detection frameworks with P4 and a corresponding algorithm. However most efforts went into Distributed Denial of Service (DDoS) detection [21, 48] and Operating System (OS)-types classifications [2]. Furthermore to detect anomalous behavior, most common tools are perf or proc to extract metrics [8, 47, 48]. Lately there is a trend in using extended Berkeley Packet Filter (eBPF) as an extraction tool with promising results. For example, on anomaly detection in communication services [45, 55], container and kernel analysis [28, 40], and blockchain security [7].

The combination of eBPF and P4 still has room for improvement regarding the detection of abnormal behavior in programmable networks. Thus, miss out on the chance to observe the capabilities of the switch as a central control point in the network. This would offer a lean, scalable, and efficient way of detecting anomalies within a network.

The main goal of this thesis is to provide a framework that enables the extraction of resource metrics from P4-enabled switches using eBPF. To get more precise and fine-grained information from the switch through the usage of eBPF. Additionally, the goal is to train machine learning algorithms on said data to identify, what value eBPF brings to intelligent anomaly detection. To achieve this, the initial step is to get familiar with the topic and its state of the art. The assessment of a suitable environment will be the next step. From there on one must examine, what the best practices are to work with P4 and eBPF. In the next step, one must make themselves familiar with P4 and the evaluated environment. Further, the creation and deployment of an appropriate P4-enabled topology must be done. In addition to that, the switch must be made eBPF ready so that one can create and implement the extraction via eBPF. Additionally, the simulation scripts of normal and attack scenarios must be created. Finally, the simulations can be run to extract live resource metrics from the switch. The next objective is to clean the extracted metrics and train appropriate machine learning algorithms. For the last step one has to assess the results of the simulations and evaluate the key points regarding the performance and accuracy of the framework.

Regarding the methodology as a first step, plenteous of information was gathered via a bibliographic review. Furthermore, the current state of research in the field of intelligent anomaly detection in programmable networks was assessed. The machine learning algorithms, the appropriate environment as well as the extraction library were chosen through research and sometimes hands-on testing. As an example, the case of Mininet and P4Docker can be mentioned here. Additionally, further assessments were done to gain an understanding of machine learning and P4. The creation of eBPF scripts was created by looking at header files, man pages, documentation, and examples.

This thesis is organized as follows. First, the thesis explains the importance of the work to get the reader hooked and inspired. Then, the topic is introduced, where the reader gets an overview and a summary of the current research state. This counts as an introduction and explanation part of the design chapter. The design of the work is introduced with the implementation. This was done to explain the design together with the implementation. After the explanation of how the work was equipped the evaluation of the thesis is introduced. This counts as a logical step since the reader just learned how it was supplied. In the final review, the findings are summarized, possible conclusions are drawn and areas for future work are identified to round of the work.

Chapter 2

Fundamentals

In this chapter the fundamentals of this work are presented. The first section provides the underlining concepts of programmable networks. The second section presents a summary of fingerprinting. The following section presents anomaly detection. And the last section represents the literature review of related works.

2.1 Programmable Networks

As the name of programmable networks suggest, the networks become programmable. It separates the control from the data [1]. This promises a swift roll out of new network services and environments [5]. There are a lot of advantages that have emerged in deploying a programmable network. For example, a device's equipment can be upgraded remotely, allowing the reuse of hardware even for different national standards and regulations. Additionally, it enables more realistic research. Nevertheless, programmable networks have higher power consumption and ultimately process fewer packets per second. Supplementary, security can become an issue as malicious access can be done at a much lower level. Programmable networks can have a wide variety of applications [29]. The beginning of programmable networks started in the 1960s [1]. Charles G. Miller applied for a patent for moving the telephone network to a digital implementation in order to prevent eavesdropping in military communications [34]. Therefore, programmable networks originated from the telephone networks [1, 5]. Open Signaling (OPENSIG) and Active Networks (AN) are the two main opinions in programmable networks [5]. The Open Signaling Working Group in turn created a special purpose working group with Internet Engineering Task Force (IETF) [6]. It created the General Switch Management Protocol (GSMP), which allows to remotely manage switches [37]. Another way to work with switches is to make them programmable. P4 is one way to do this. It is applicable for high traffic loads and fast packet forwarding [21]. However, it requires special hardware such as a Barefoot Tofino processor to work with P4 [21].

AN is described as a programmable network infrastructure that has either separate channels or one channel for control and data transfer. Additionally, routers execute program

fragments. As there were some security and performance concerns, such as the unsafe execution of code, which diminished the application in practice [4, 37].

2.1.1 Software-defined Networking

Programmable networks got a breath of fresh air with the appearance of SDN [37]. Namely when OpenFlow was introduced. A tool that allows experimental protocols to be run in a real world environment without vendors having to uncover the internal secrets of their switches [32]. Among others OPENSIG and AN are considered the forerunners of today's OpenFlow [37].

SDN usually consists of three to four distinct components. The first is the compute machines, which in SDN are called the programmable edge elements. Despite their name, they are not fully programmable as they are managed by a controller. They are indirectly controlled by, for example, an OpenFlow enabled switch, the so-called programmable network element, which forwards the instructions of the SDN controller. The fourth element, the divisor, manages multiple controllers. This can also be referred to as the Southbound interface. On the other hand, the Northbound interface enables to add functionalities to the controller via Application Programming Interfaces (APIs). An example of an API would be a load balancer [29].

2.2 Fingerprinting

Fingerprinting models a digital profile or pattern of device behavior. It successfully identifies possible misbehaviors, such as malicious attempts, malfunctions within the hardware or software and improper setup [46]. To detect misbehavior, one must monitor traffic when it deviates from normal behavior. A different approach to assuming that misbehavior causes the data to look very distinctive is to look at the distribution instead. One successful way to do this is to look at the randomness and spread of the traffic [16].

The malfunction of a device can have drastic effects on our environment, for example the failure of the metro. For this reason, it is vital to successfully predict where the next fault may occur in order to prevent it. One can predict such an incident by devices behaving abnormally [30]. Another type of misbehavior detection is to detect malicious attacks. One approach is to model the basic behavior of devices within the network and then monitor their behavior based on a trust score. If the trust score is too low, the packets are rejected due to suspicious behavior [20]. Fingerprinting is also successfully used to identify devices. The devices are categorized by type based on their behavior. This can be very beneficial for network management, for example to enforce security policies [31].

2.2.1 Creating and detecting fingerprints

This section discusses how to work with the data retrieved from the device in order to create a behavior profile. There are several approaches to that.

- **Rule-based:** If a device is well known and therefore its behaviors. A set of rules can be composed that the device must be within that frame, otherwise it is considered an anomaly. Due to its simplicity, it is not applicable for complex scenarios [46]. One approach to specifying the normal behavior of an IoT device is to look at the Manufacturer Usage Descriptions (MUDs) [25]. Additionally, you can look at all software that a device executes and defines its capabilities and resource usage [46].

- **Statistical:** Are simple and do not require a lot of data for implementation, but it is not good for complex data. It uses statistics to determine whether a value should be labeled as normal or not.

- **Knowledge-based:** Is good for small amounts of data. With too much data, it can become very complex and inefficient because it builds rules based on the data fed into it.

- **Machine Learning/Deep Learning:** Both ML and DL are well suited to detect complex patterns. If you have a huge training set, DL is better suited. On the contrary, ML performs better on a smaller data set than DL. Such a trained model can be seen as a black box. It is therefore difficult to know what is going on inside. Another disadvantage is that such a model can consume a lot of resources during training. Working with ML/DL in fingerprinting is a promising and widely used approach. There are two distinct approaches for training ML/DL algorithms, one is supervised learning and the other is unsupervised learning. Supervised learning works with labeled data. Namely with the following two main algorithms:

  - **Classification Algorithms:** Predicts based on training data or never seen data. As the name suggests, it labels the data and states whether it is an anomaly or normal behavior. Decision Tree (DT), Random Forest, Logistic Regression (LR), Naive Bayes (NB) and Support Vector Machine (SVM) are widely used classification algorithms.
  - **Regression Algorithms:** Instead of clearly labeling the outcome, regression algorithms score the behavior based on whether it has an anomalous background or not. The most commonly used regression algorithms are Linear Regression and Polynomial Regression.

  Unsupervised learning uses unlabeled data as an input. It tries to model the density of the input data. Based on the density probabilistic patterns are detected. The algorithms make their decision based on the expected patterns.

  - **Dimensionality Reduction Algorithms:** These algorithms reduce the set of values under consideration for input data. This makes the data more efficient and fits well with statistical algorithms. Common dimensionality reduction algorithms are t-distributed Stochastic Neighbor Embedding (t-SNE) and Principal Component Analysis (PCA).
  - **Clustering Algorithms:** Cluster the input data into different groups regarding to their closeness. Frequently used algorithms are density-based spatial clustering of applications with noise (DBSCAN) and k-means.

  - **Anomaly Algorithms:** Assume that most of the training data is normal, but include some deviating data from the norm, which are considered to be the anomalies. This can be achieved with One-Class Support Vector Machine (OCSVM) and Isolation Forest (IF) algorithms.

- **Time Series:** Looks not only at the values, but also at when they occurred. Time series can be used for anomaly detection and device identification. Although time series has better performance, it needs lots of input data to be effective [46].

2.3 Anomaly detection

Anomaly detection does not only occur in cyber-security. It can occur as fraud detection, military observation, intrusion detection, medical detection, image processing, industrial detection and more. Anomaly detection is the detection of unusual patterns in data. Taking a closer look at the data can give critical insights that would otherwise remain hidden [10]. Usually, anomaly detection is done with traditional machine learning, which is still the norm for small data sets. However, if one has a large data set, deep neural networks, a subset of machine learning, are much more appropriate and popular because it shows the data in hierarchical layers [9].

Anomaly detection is not to confuse with novelty detection. A novelty is considered a newly detected pattern. It is important to mention that it is not an anomalous data point. The main goal of novelty detection is to discover if a system is off its initial baseline. [33].

Finding patterns that do not match normal behavior seems simple. In fact, it is not, there are whole host of challenges. Sometimes it is even hard to define what anomalous behavior is. The action may look normal but is actually a deviating value, or the attacker obfuscates its malicious code to look and behave like normal. Nonetheless, over time, everything changes, and so does what is an anomaly and what is not. This invalidates old normal behavior. Additionally, in the real world there is also noise, which can have a similar appearance to anomalies. Another challenge could be to receive enough valid data for training, especially for supervised learning [10].

There are three types of possible approaches when working with anomaly detection. Each depends on the available data set. Supervised anomaly detection has labeled data, which means one has a clearly labeled data set of anomalies and normal behaviors. A common issue with labeled anomaly detection is to have much more data from normal behaviors than from anomalous behavior [10]. Oversampling methods are a solution for dealing with imbalances. Here, the size of the class that is smaller is increased by a random replication of representatives of the corresponding class until it is equal in size as the other class. Oversampling is considered a good approach as no information is lost [17]. Semi-supervised anomaly detection is performed when only the data set of normal behavior is labeled available. As the last approach, unsupervised anomaly detection does not need any training data and therefore has many use-case scenarios. It is only effective when the normal behavior occurs more often than anomalous behavior [10].

The output of the anomaly detection is how the deviating behavior is reported. This can be done via scores, which shows the grade on an instance being an anomaly or via labeling, where an instance is either considered to be a normal behavior or anomalous behavior [10].

2.3.1 Types of Anomaly

Anomalies are split into different types. This classification serves to better detect the anomalies.

- **Point Anomalies:** This is considered the basic example. Here the anomaly is far away from the rest of the data.
- **Contextual Anomalies:** As the name suggest, the data is only an anomaly if it is in the correct context. Otherwise, they are considered normal. The contextual attributes of said anomaly define the context of the data, such as when or where. The behavioral attributes describe the actual values.
- **Collective Anomalies:** If multiple data appear in the same region, this is an anomaly. However, if single data appears in the same region, this is not considered an anomaly.

It is worth noting that there can only ever be one point anomaly. Additionally, a contextual anomaly and a point anomaly can simultaneously be a collective anomaly. One anomaly can therefore be transformed into another anomaly in order to change the detection problem [10].

2.4 Related Work

This section discusses different related works. The goal is to find the current state of research and technology used. In order to identify the delta this work can fill. In the Table 2.1 one can find the summary of the discussion.

2.4.1 P4-enabled switches

Vendors implement their data path hardware differently and SDN often relies on protocols such as OpenFlow. One approach to circumvent these dependencies is to use P4 [48]. P4 is a protocol that enables the individual configuration and programmability of a switch. It is vendor-independent, reconfigurable, protocol- and target-independent [3]. On top of that it can be used to monitor the metrics of the switch. Nevertheless, it lacks the ability to check the correctness of the P4 programs. Fortunately, there are efforts on building a verifier for P4 [53]. Thus [48] addresses the possible detection of faulty P4 programs and malicious behavior. They implemented a topology consisting of three switches and

three hosts. Used Mininet and Behavioral Model version 2 (BMv2) switches with a P4 Multi-Hop Route Inspection (MRI) program. MRI tracks the exact path a packet has taken and its queue. Additionally, iPerf was used to create distinct flow behaviors [48]. The framework created is called FEVER. It was built by them to detect anomalous behavior in P4-enabled networks. To do so, the authors used the metrics of resources and network telemetry extracted with perf, proc and P4. Further they created 6840 samples of all behaviors, 3240 of which were normal behavior. With those samples they trained two unsupervised machine learning algorithms, namely OCSVM and Local Outlier Factor (LOF). ML algorithms are often used to detect anomalies. The performance of the unsupervised algorithms was evaluated with the F1 score. FEVER is very successful in detecting malicious behaviors and identifying P4 programs. Nevertheless, the success of the ML algorithms was highly dependent on Resident Set Size (RSS) [48].

Active fingerprinting can in some cases convolute the network traffic, for example, with Nmap, as they create supplementary network load. Therefore, passive fingerprinting is more suitable for larger networks, as they are viewed from the outside. Additionally, passive fingerprinting does not miss hosts that create inbound traffic and hosts that appear or disappear quickly. As they monitor the network in real time and not just during a scan. Therefore, [2] built P40f an OS-based TCP SYN packet passive fingerprinter that can intervene directly with packets. For that BMv2 switches were used with a P4 program on them. Furthermore, they used Protocol Independent Switch Architecture (PISA) to deploy P4 programs in the campus network. With P4, the fingerprinter looks at the Operating System information of the packet sender and thus can easily detect vulnerable OS-types. This also works in encrypted networks, because it exclusively looks at the header and not at the encrypted payload. To do this and maintain an overview, P40f always stores the source address, source port and destination address as a triple in a Bloom filter with its sequence number. Each SYN packet is checked against the Bloom filter. If the filter contains a desired set, it duplicates the set from the filter and forwards it. Unfortunately, the Bloom filter can produce false positives due to the way it stores entries. Fortunately, it creates no false negatives. The authors found P40f good for outbound communications, but not for inbound communications. With inbound packets the passive fingerprinter achieves an accuracy of over 98%. Thus, adding great value for network administrators. In addition, one can track whether there is a vulnerable OS-type or whether a host is impersonating another OS [2].

Nevertheless, DDoS attacks are becoming ever more frequent and intense. There are already solutions for defending against DDoS-attacks. However, they have a high overhead, as such detection tools analyze every forwarded packet. While it should be effective in use. Usually packet sampling such as sFlow or forwarding protocols such as OpenFlow are used. Nevertheless, they do not fully eradicate the problem. Either they lack resource efficiency or accuracy. Additionally, there must always be a communication loop between the data and control plane, which leads to unnecessary inefficiencies. This can be circumvented with the help of P4. Since it lets one program the data plane directly. Euclid is an approach for fast and accurate DDoS-attack detection and mitigation using P4 [21]. To do so they used observation windows (OWs). They chop the incoming flow into fixed size groups. The segregation size must be increased for a higher throughput and a robust representation. In each OW the number of identical destination IPs and identical source IPs are being counted. Then those counted frequencies are fed into the entropy logic. The

entropy logic measures how unpredictable or different the values are within the input. The output of the entropy is used to create the statistical model and keep it up to date with normal behavior. Then possible anomalous behavior is detected using a traffic model from the entropies. If an attack is detected, the attack mitigation takes places. Each packet must pass through three stages. First, Euclid measures the occurrence variation of the source and destination addresses. After that a decision is made whether a packet is suspicious or not. Lastly, the packets are appropriately handled. Either dropped, throttled or sent to the appropriate destination [21].

For methodological purposes, [47] proposed a procedure for programmable switches to create behavioral fingerprints. This is important because enhanced anomalous behavior detection improves anomaly detection. Protocol-independent Packet Processors switches can be used for this purpose due to their programmable data plane. This allows metrics to be extracted from the data plane. The experiment is conducted on a Commercial-Off-The-Shelf (COTS) server in Mininet with BMv2 switches and a P4 program. The P4 program implements MRI. This is because MRI tracks the route taken by the packet and the length of the queue. The metrics which were extracted are Queue Depth and Switch ID from the In-band Network Telemetry (INT). CPU and RAM measurements are extracted with the Linux Table of Processes (top) command. For the identification of individual switches, Process Identifiers (PIDs) were used as unique identifiers. After one hour of normal data collection, abnormal behaviors were brought in. In the next step the data gets prepared and thus generating the fingerprint. Preferably, a fingerprint of the P4 program is also made to avoid mixing up anomalous behavior with P4 program bugs. After that the ML algorithms can be trained [47].

2.4.2 Monitoring with eBPF

As SDN and cloud computing are becoming increasingly more popular, routers have become more open. Due to this reason [45] suggests monitoring routers with eBPF, which promises a more insightful approach than regularly measured metrics. eBPF is a technology that enables safe and fast communication and alterations directly with the kernel at runtime. By the user defined code can be injected into the kernel and then executed [51]. The extended Berkeley Packet Filter is only executed in the kernel when certain hooks are triggered. This makes it very resource efficient. [45] wrote their eBPF program with C and compiled it to eBPF bytecode to load it into their six individual virtualized SONiC routers kernel. Each bystander router is connected via a virtual interface using Border Gateway Protocol (BGP) and Open Shortest Path First version 2 (OSPFv2) protocols [45]. BGP's main functionality is to inform other BGP systems what kind of networks they can reach [44]. With OSPFv2, each router maintains a database of the topology of the autonomous system [36]. In the case of [45] the authors use the obtained metrics from eBPF in the Operation Support System (OSS) to train the deep learning models. They used Long Short-Term Memory (LSTM) for anomaly- or BGP sessions detachment-detection and LSTM Autoencoder for human errors such as misconfigurations. In general, it is recommended to use an OSS to train models when specific resources such as GPU are being used. The LSTM autocoder was trained with 300 normal samples, each of which capturing 300 seconds of the sample. For the training with LSTM to detect anomalies,

[45] 150 normal and 150 anomalous datasets were used, each with a measurement time of 400 seconds. The results of the work were evaluated with the F1 score. The evidence indicates a high improvement in detecting human errors. However, there is no significant improvement in detecting anomalies. This could be due to the fact that they only used eBPF to measure routing table length. Therefore, [45] suggested that a future work should look at other metrics extracted from eBPF such as disk I/O speed.

The paper of [55] made a tool for predicting network situations with eBPF and LSTM. For this purpose, they used experimental hardware with an installed Linux OS. They were interested in the TCP data that takes place between the communication of the Linux host and a web server. To do so, they used eBPF to collect the size of tcp_sendmsg and the length of tcp_v4_rcv, tcp_v6_rcv. The tcp-rcv-kernel function is used to transmit the acknowledgment of the TCP connection, as send tcp_sendmsg() and tcp_v4_rcv() are not used exclusively for HTTP transmission. Therefore, you must filter by HTTP and HTTPS ports to extract the correct data via eBPF. Doing so, they managed to create an accurate prediction of the network situation [55].

The next work proposes an analysis system that enables anomaly detection with data collection using eBPF in a modern cloud-native infrastructure. To do so, a Kubernetes cluster with pods and containers was used. [28] run micro-services within the containers. Data was extracted from the containers to detect anomalous behavior. Thus, enabling them to locate the specific pod where the anomalous behavior occurred. To test their implementation, [28] modified weaveworks. For additional tests, Locust was installed and run on a distinct Kubernetes cluster to simulate traffic for eBPF measurements. Locust is a tool for load-testing. While Locust is running, logs are collected with the help of eBPF. Then distinct logs get united and sent to the SLS instances of the Alibaba Cloud log storage. This is for training and reviewing, which can detect anomalous behavior. Speaking of irregular behavior: Chaosmesh is used for creating anomalous behavior. Chaosmesh is a platform to let you simulate faults in a controlled manner in Kubernetes. [28] extracted TPC, UDP, HTTP and DNS metrics using socketfilter, tracepoint and kprobe/kretprobe as eBPF programs. This results in a network delay of around 1% during runtime. Additional metrics form the Kubernetes apiserver, such as pid, src_ip, src_port and many more were used for the analysis system [28].

[40] argued that the restriction regarding the unavailability of floating-point values in eBPF has disadvantages for complex algorithms. For example, libraries that use floating-points will fail. These kinds of security measurements have been implemented in eBPF to protect the Linux-kernel from non-deterministic behavior. Due to this reason, [40] has implemented dynamic fixed-point values as a substitute for the missing floating-point values. Dynamic fixed-point are fixed-point values with dynamic bit allocation. The distinction between fixed-point values and floating points is that the fixed-point values have a narrower range of possible values, since they convert real numbers into integers through bit shifts. A benefit over fixed-point values is being more precise. To validate the effectiveness of the dynamic fixed-point values, they were tested in an DDoS anomaly detection framework. A statistical approach is promised to make the framework faster than a neural network or machine learning approach, even though they are more flexible and accurate. Instead of a regular implementation with eBPF, in which the anomaly detection program is in the user space and the data extraction takes place in the kernel space. The anomaly

detection program was implemented directly in the kernel space. GothX [41] was used to create the dataset [40]. GothX is a generator for normal and anomalous IoT network traffic [41]. The environment was built in GNS3. The implemented framework had about the same accuracy but about 18% higher throughput as the user space anomaly detection. This is due to moving the workload to the kernel space. [40] recommends to do tasks high in memory usage inside eBPF programs [40].

2.4.3 Docker Container Anomaly Monitoring

[56] has created an anomaly monitoring system that pursues a special system architecture. It consists of a monitoring server and multiple host machines. The monitoring server is used to store monitoring data, the anomaly detection using Isolation Forest and the anomaly analysis. On each host machine there is a monitoring agent. It communicates with the monitoring data storage and the anomaly analysis. It collects resource metrics such as ID, CPU/memory usage, disk I/O rate and much more for the containers on the current machine. Based on the input from the anomaly analysis, the agent can adjust its monitoring period during runtime. This is needed since a small surveillance time is suitable to detect anomalous behavior but creates excessive resource consumption. On the other hand, a large monitor interval results in a slow detection. The monitoring data storage module, which runs on the monitoring server, receives the collected resource data from the monitoring agents. It processes the incoming data in a specific format and then stores the formatted container data in InfluxDB. Only the data is forwarded to the anomaly detection module [56].

At the anomaly detection module, the measurements first get cleaned from impure data such as missing values, duplicates or immediate extreme outlines. In the next step, weights are applied. Weights are a necessary key element to improve the performance of the Isolation Forest (IF). They should be applied according to their use. So, if an environment is very dependent on its CPU, the weight of the CPU metric should be increased. Then the IF is used. An Isolation Forest consists of multiple Isolation Trees (iTrees), which are a type of binary tree. The shorter a path is for a data, the less frequently it occurs. This signals a potential anomalous data. The anomaly analysis module receives the log from the anomaly detection module, filters it with a predefined threshold and identifies it as an anomaly if the value of the measurement is higher. If anomalous behavior is detected, the cause is determined based on the obtained log. [56] implemented the previously discussed structure in a simulated and a real cloud environment. The simulated cloud environment consisted of Linux machines, one for the monitoring server and the others for using monitored hosts with Docker containers. The real cloud environment was implemented using the Amazon EC2 service with Linux machines as server and hosts. To create benchmarks, [56] used CloudSuite. CloudSuite measures how well cloud services perform with the eight most common applications used in datacenters. Since there is no benchmark suite for anomalous behavior on container, anomalies were created with an infinite loop in the CPU, a memory leak, a Disk I/O fault and network bandwidth restrictions. 200 tests were conducted with an equal distribution of anomaly types. IF delivers similar or better results than Local Outlier Factor due to the use of weights. Especially in the optimized Web search IF shines compared to LOF regarding false alarms. The issue with the ML

algorithm LOF is that it is more sensitive to fluctuating metrics during normal runtime. It has been found that the perfect amount of iTrees for an iForest is 100 [56].

2.4.4 Intelligent and behavioral-based detection of malware in IoT spectrum sensors

The framework of [8] detects and classifies anomalous behavior in Internet-of-Things (IoT) devices, namely in a Raspberry Pi 4 equipped with a Software Defined Radio (SDR) kit. Various ML approaches were analyzed and the best algorithms for each scenario and their pain points were evaluated. The anomaly classification identifies botnets, rootkits, backdoors, ransomware and cryptojackers [8]. In order to train the data with the ML algorithms, it must first be collected. The data gathering module is responsible for this. It periodically surveils distinct kernel events. Apart from the data collecting, the data sent to the central server takes place within the sensor. The central or decentralized server then receives the data within the corresponding module. The measurements received are also pre-processed. This means that faulty data is filtered out and then aggregated and normalized [8].

The ML algorithms of the classifier and anomaly detection can now work with the received metrics. It is known from literature that classifiers generally perform better than intelligent anomaly detectors. Distinct supervised algorithms were used for classifiers. Whereas for anomaly detection, semi-supervised and unsupervised algorithms were used, which were only trained with normal behavior to be able to detect zero-day attacks. Two distinct experiments were conducted, one in a Fiber to the Home (FTTH) setting and the other in a 4G mobile network. In the FTTH experiment, OCSVM stood out with the best performance for anomaly detection and XGBoost (XGB) with Random Forest for malicious software classification. In the 4G mobile experiment, XGB performed best in the classification of malware and OCSVM performed best in anomalous behavior detection with LOF delivering comparable statistics. Ultimately, in both experiments, the harmless attacks were the ones that slipped under the radar in terms of classification and detection of anomalies. Nevertheless, they delivered promising results. In general, normal behavior was rarely flagged as anomalous behavior [8].

Table 2.1: Literature Review

Work	Impl.	Detection	Metrics	Extraction	Misbhvr.	Samples
Fever [48]	Mininet & BMv2 switches with P4 program	OCSVM, LOF (unsupervised)	Cycles, Instructions, Branches, Branch Misses, Task Clock, Context Switches, CPU Migrations, CPU Usage, Page Faults, RSS	Proc, perf	UDP flood, P4 altercation, P4 program replacement	Total: 6'840 Normal: 3'240
P40f [2]	PISA on campus network & BMv2 switches with P4 program	Knowledge-based	Packet headers	P4	Only fingerprinting OS-types	Total: 286'215 SYN packets
Router [45]	SONiC-routers	LSTM autoencoder / LSTM (deep learning)	Length of routing table, memory utilization rate, CPU utilization rate	eBPF	human errors & detection before issue	Total: 300 datasets each 300 seconds & Total: 300 datasets Normal: 150 datasets each 400 seconds
Prediction Model [55]	Experimental hardware with Linux OS	LSTM	Size of tcp_sendmsg, length of tcp_v4_rcv, tcp_v6_rcv	eBPF	Predict network situation	NaN
Alibaba Cloud [28]	Kubernetes with Pods and Containers on Aliyun ECS ecs.c6.xlarge servers	Alibaba Cloud log storage SLS (includes ML)	TCP, UDP, HTTP, DNS metrics and Kubernetes apiserver data	eBPF & Kubernetes apiserver	Anomaly detection	NaN
Fingerprint [47]	Mininet & BMv2 switches with P4 program	None	Queue Depth, Switch ID, CPU and RAM	P4 & Linux	Behavioral fingerprinting	Normal: One hour normal behavior
iForest [56]	Monitoring server and host machines	Optimized Isolation Forest	ID, CPU and memory usage, disk I/O rate, network receive rate, network transmission rate	-	Anomaly monitoring system	Total: 200 tests
Euclid [21]	P4 enabled switches	Statistical	Source IP, destination IP	P4	DDoS detection and mitigation	CAIDA Anonym. Internet Traces dataset & CAIDA DDoS Attack dataset
IoT spectrum [8]	Raspberry Pi 4 Model B with RTL-SDR Blog V3 R820T2 RTL2832U	RF, SVM, KNN, GNB, XGBoost, DT (classification) & OCSVM, IF, LOF (anomaly detection)	Network, Virtual Memory, File System, Scheduler, CPU, Device Drivers, Random Number Generation	Perf	Class. and anomaly detection	Total: 25 datasets per experiment
Dynamic Fixed-point Values [40]	GNS3 with Docker containers and virtual machines	Statistical	-	eBPF	DDoS	Total: 36M MQTT packets

Chapter 3

Design & Implementation

This chapter first provides an overview of the design and implementation. Later, each section introduces the corresponding topic with the reasoning and implementation. First, the environment section is demonstrated. Then the data extraction, next the simulation and lastly the intelligent anomaly detection are presented in detail.

3.1 IDAPN Overview

This thesis proposes the IDAPN approach, which serves the intelligent detection of anomalies in programmable networks. It enables to extract huge amounts of resource metrics directly form the kernel and detect anomalies using only the switch's system measurements. Figure 3.1 visualizes the IDAPN approach. The figure and the design are inspired by [15]. To start with the IDAPN proposal, an environment consisting out of one P4-enabled switch with a basic P4 code is needed. It additionally requires two identical Linux machines, which are deployed in individual Docker containers. The switch has an edge to each host, but the hosts are not directly connected to each other, as seen in the Figure 3.1. For more details on the configuration of the testbed, see Figure 3.2. All base images for the devices were extracted from P4Docker's Virtual Machine, including the startup and clean script for said topology. Nevertheless, the evacuated image of the switch was made eBPF ready using a custom Dockerfile. This enables direct data extraction from the kernel of the containerized switch. The experiment is conducted on a Lenovo ThinkStation S20 with an Ubuntu 20.04 Operating System. Figure 3.2 shows the environment set up for this thesis.

The data extraction for the intelligent anomaly detection is done using eBPF, more precisely with the libbpf library for C. This is a novel approach on extracting metrics directly from the kernel using hooks. With libbpf there are at least two C-scripts, one for the user space and the other for the kernel space. The user program handles the interaction and export of the data, while the kernel script extracts the needed statistics. Communication from the kernel to the user code usually takes place via a ring buffer, as is the case in this work. In the end, 1405 different metrics were derived directly from the kernel.

To extract the metrics, three different types of simulations were run using the traffic generator. The two Linux hosts pinged each other via the switch. The normal simulation took 4h and consisted of a mean, normally distributed packet size of 300 bytes and an interval mean of one second, also normally distributed. The flooding attack is simulated for 4 hours with a close to the maximum as possible packet size of 65500 bytes. These packets are sent as fast as possible from both hosts to each other, simulating for example the traffic load of a DDoS attack. Finally, various delay attacks were simulated, all of which had a mean interval of 5 seconds. The distinction lies in the packet size. It starts at 300 bytes and is then incremented in steps of 300 bytes until 1800 bytes for each simulation run. Examples of delay attacks can be a Man in the Middle (MitM), Denial of Service (DoS) via delay attack and many more.

The test manager, as the name suggests, manages the tests. First, the P4Docker Topology Editor is used to create the environment set up scripts. These act as the Deployer. After a successful data extraction via eBPF, the data sets must be cleaned by the Data Cleaner. This consists of two distinct Python scripts, one for cleaning the normal data and the other for cleaning the same features unloaded in the normal data file. These are then later fed into the ML Trainer, which again is a Python script. The code of this work is publicly available at https://github.com/dattes/IDAPN.

3.2 Environment

To achieve a representative result, this thesis wanted to be as realistic as possible in every aspect. The best way is to have a real environment. However, due to resource restrictions, one is forced to use something different. Therefore, an analysis on which environment is best for the IDAPN case was made. The Table 3.1 shows the considerations. There are some very realistic options such as Proxmox, Vmware ESXi and Kubernetes, which are frequently used in the real world [18, 19, 28]. Proxmox has the advantage of doing free full virtualization, although it offers paid support. This is due to its nature of being an open source. Vmware ESXi, on the other hand, demands payment for the license. Both have extensive documentation and no actual P4 support, although one could use the Open vSwitch (OVS) and enable P4 in such a scenario. Kubernetes takes a slightly different approach to full virtualization that uses and manages containers. Scalability, good documentation and free use are other advantages of Kubernetes. Nevertheless, there is no need for scalability for this work. Unfortunately, all the previously discussed frameworks come with a big overhead, especially for scientific purposes. Many use Mininet for scientific purposes as it is lightweight and can be deployed quickly. Nevertheless, it is run in a virtual environment, which means it may not cover all aspects of a real world example [23]. As already presented by Kubernetes, another way to implement an environment is to containerize it with Docker. For this approach, one can either use ContainerNet, P4Docker [49] or work directly with Docker [14]. ContainerNet is free to use, has limited scalability and is designed for experiments. It is also a fork of Mininet that allows Docker containers as hosts. The last two options that are considered for the environment are either manual implementation with Docker or the use of P4Docker. P4Docker has a GUI that allows to create the own environment and compile the P4

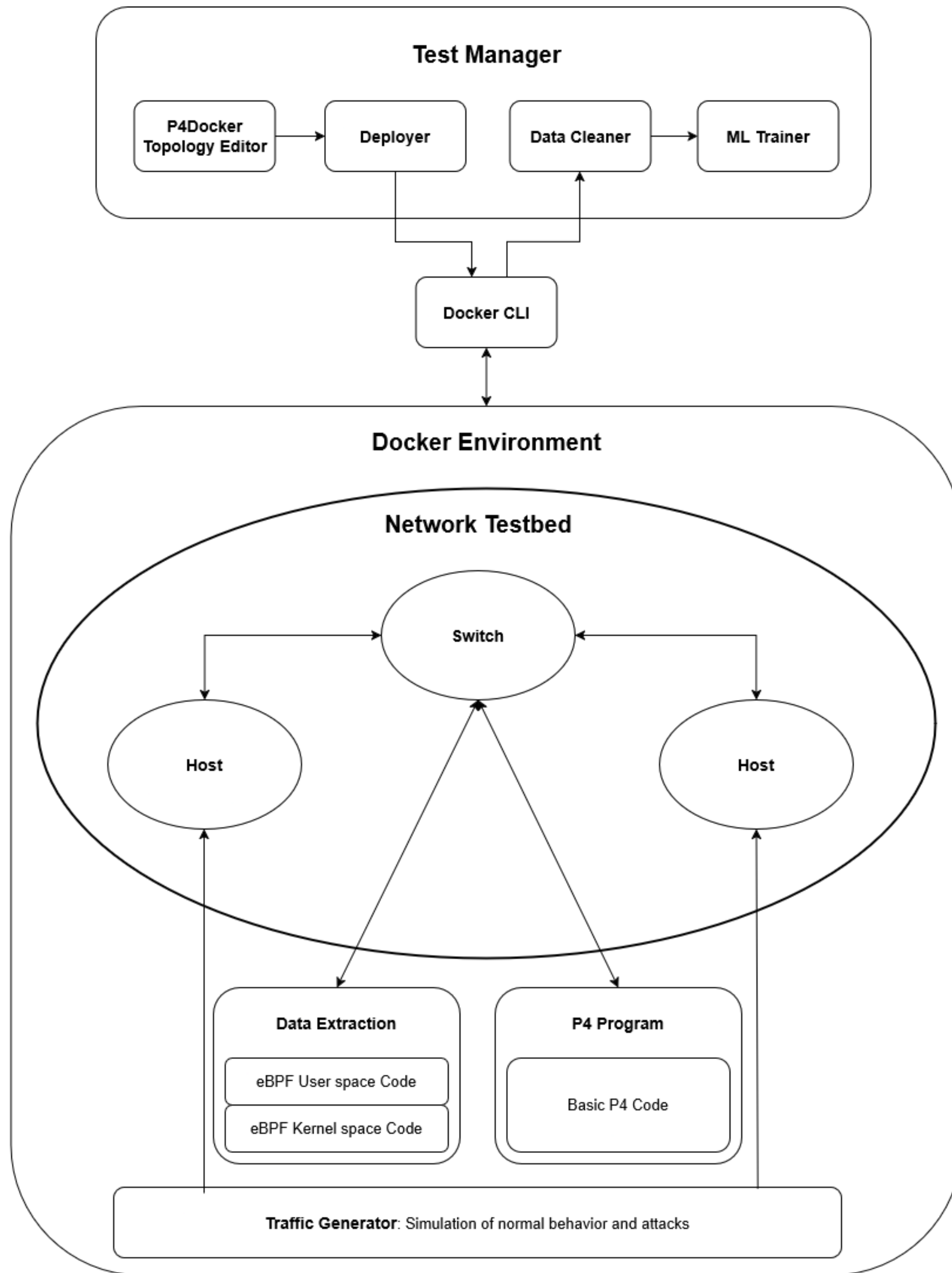

Figure 3.1: IDAPN Overview

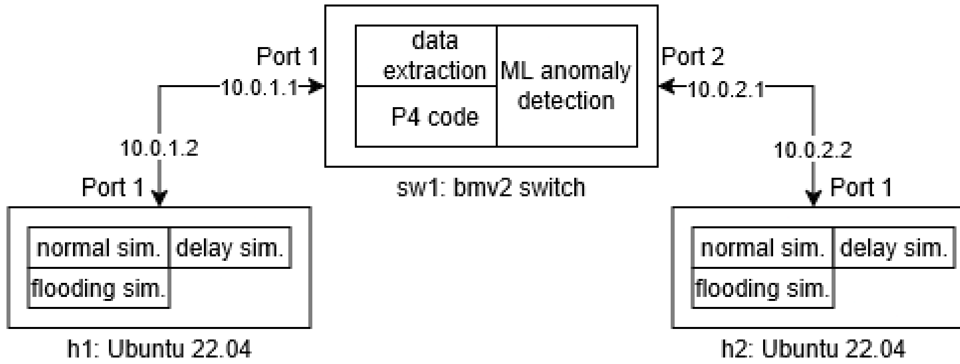

Figure 3.2: Topology of the Environment Built to Support the Thesis Development

code, once being satisfied with the configuration. P4Docker can automatically create a script for building and destroying the configured topology with Docker containers and the desired switch with the P4 code. Since P4Docker is new, the documentation is not very extensive and there is no active eBPF support. All the automatic script creation would mean manual labor in creating Docker environments by hand.

Because the real world environments Proxmox, VMware ESXi and Kubernetes have too big overhead for this works purpose. Likewise, ContainerNet with Mininet is an emulation that could later cause problem when extracting metrics directly from the kernel. Therefore, the ultimate decision on the choice of environment is P4Docker. Due to its nature, it works with P4-enabled switches and creates a real implementation. In the Table 3.1 the summary of the above discussion can be found.

P4Docker is being delivered as an image. It was originally used in VirtualBox [39] as a Virtual Machine (VM). Nevertheless, problems quickly arise when trying to read from the switch's container kernel, because the P4Docker image has the kernel version 6.8.0-49-generic and the BMv2 switch has version 5.15.0-131-generic. This results in miscommunications between the host and the containerized switch. For this reason, the scripts and images were extracted from P4Docker and used on a Lenovo ThinkStation S20 with Ubuntu 20.04 as the Operating System with the exact same kernel version as the switch. The comparison of the Operating Systems can be found in the Table 3.2. The kernel versions were extracted with the command "uname -r" and the OS versions were obtained using "cat /etc/os-release".

The files detached from P4Docker were a JavaScript Object Notation (JSON) file and two shell scripts. The "basicConfig.sh" is intended for deploying the designed topology with the Docker containers. It first starts up the needed containers with the desired images. For this thesis, the desired images were changed to the eBPF-enabled customized image. It is worth noting that the switch must have the sys folder of the ThinkStation mounted, the memory lock limit removed and run in privileged mode to work properly with eBPF. This was done over the Docker run command. The configuration script then creates virtual Ethernet (vEth) interfaces and sets the namespaces for the vEths. In the next step, the IP and MAC addresses of the interface are assigned, and the interface is activated. Finally, the default gateway of the hosts is set to the switch, the P4 switch is started with the compiled P4 code as a JSON file (basic.json) and forwarding rules are added to

Table 3.1: Environment Review

Name	Pro	Cons	Technical
Proxmox [43]	- Full virtualization - Open source - Web-based management - Extensive documentation - Free to use - Suitable for small & medium cases	- Heavyweight - Complex setup - No P4 support - No eBPF support - Paid support	- Server management platform - Enterprise virtualization - VMs & Containers - OVS for P4 - Ansible, Terraform, Prometheus
VMware ESXi [52]	- Enterprise-grade - Full virtualization - Extensive documentation	- No P4 support - Paid license required	- Bare-metal hypervisor - Architecture virtualization - OVS for P4
P4Docker [49]	- GUI platform - P4 enabled - High productivity - Lower entry barrier - Free to use	- No large topologies - New technology - No active execution or administration - Limited documentation - No eBPF support	- P4-enabled switches - Automates Docker configuration
Mininet [35]	- Support for SDN - Low resource usage - Lower entry barrier - Free to use	- No large topologies - No eBPF support	- Emulator - Designed for experiments
ContainerNet [12]	- Mininet-based topology - Uses Docker containers as hosts - Supports dynamic topology changes - Good documentation - Free to use	- Limited scalability - Not enterprise-grade	- Fork of Mininet - Used for experiments - Managed through Python API
Custom Dockers [14]	- Realistic network behavior - Flexible in design - Free to use	- Requires manual setup - Complex configuration	- P4-switch images available - Network setup on a single machine
Kubernetes [42]	- Scalable - Well-documented - Free to use - Native networking via CNI	- Overhead for small-scale experiments - Paid enterprise support	- Native eBPF support - Suitable for large-scale deployments - Uses Cilium [11] for eBPF - Uses Grafana [24] for visualization

Table 3.2: Operating System Comparison

Name	OS-Version	Kernel-Version
P4Docker	Ubuntu 22.04.5 LTS	6.8.0-49-generic
Bmv2 switch	Ubuntu 20.04.3 LTS	5.15.0-131-generic
Lenovo ThinkStation S20	Ubuntu 20.04.6 LTS	5.15.0-131-generic

Table 3.3: Topology Review

Topology	Pro	Cons
Star	- Setup - Robust - Fault identification and isolation	- Single point of failure
Hierarchical	- Scalability - Centralized control - Overhead - High cost	- Single point of failure
Spine-Leaf [38]	- Highly scalable - Low latency and high bandwidth - High fault tolerance - Most modern and prominent	- Design and management - High cost

the switch. The ”cleanbasic.sh” script stops and removes the containers and discards the interfaces. The compiled code called ”basic.p4” and done in the exercise from P4. Further information can be found on https://github.com/p4lang/tutorials/tree/master/exercises.

Nevertheless, the topology of the simulation still needs to be defined. The first and simplest possible topology is a star-like structure with one switch and two hosts. It has an easy setup, a simple fault identification and is robust. If one link fails, there is no effect on the other links. However, there is a single point of failure. With a hierarchical structure, there is great scalability and centralized control. Although, there is a small overhead because, for example, the root switch is connected to intermediate switches. The most advanced and prominent topology considered is the spine-leaf architecture. It is highly scalable and offers high bandwidth, high fault tolerance and consistent performance. The downsides are the high complexity in design, management and the associated high costs. Due to the nature of P4, the enabled switch does not care about the surroundings, as all incoming packets are processed according to the P4 code. It is not necessary to have a complicated topology. Therefore, a simple star-like structure was chosen, which is best suited for this work. In the Table 3.3 the comparison of the topology can be reviewed.

3.3 Data Extraction

An important part of intelligent anomalies detection is the extraction of resources. The most prominent case of extracting such metrics is the use of perf or proc [8, 47, 48], due to the rise of eBPF and its encouraging evidence in combination with artificial intelligence [28, 40, 45, 55]. The decision was made to go with eBPF. Nevertheless, there are different ways to implement eBPF. The most prominent and therefore oldest library for working with eBPF is bcc [22]. The Python library has many predefined functions and tools that can be used. In addition, bcc is mainly used with Python as its language in which one can get great visualizations. Nevertheless, it is considered outdated and has the overhead of Python. Then there is bpftool [26]. It uses the command line instead of creating its custom code. As a result, it has many predefined commands, which makes it easier to use.

Table 3.4: Data Extraction Review

Library	Purpose	Pro	Cons	Language
libbpf [27]	- Custom files in Linux kernel - Open, Load, Attach, Close	- Hooks - Compile Once-Run Everywhere - Skeleton files -Maps support	- Low level - Limited documentation	- C
bcc [22]	- Initial tool to work with eBPF	- Predefined commands - Easy to use	- Python overhead -Limited low-level access -Considered outdated	- Python
bpftool [26]	- Working with commands	- Predefined commands - Easy to use	- Limited low-level access - Depended on commands	- Command line

Lastly, consider libbpf [27]. This is a low-level C library. It works with hooking functions within the kernel and consists of four different steps in each C script, namely opening, loading, attaching and finally closing the compiled BPF object file. It is worth noting that this only works for Linux kernels. A disadvantage of libbpf is the limited documentation, even though it is frequently used [28, 40, 45, 55]. Table 3.4 summarizes the discussion.

After assessing the possibilities, the decision was made in favor of libbpf due to the code customizations and frequent usages. The switch's Docker image must be prepared for working with libbpf. For this, various packages must be installed via Dockerfile. The list of the specific packages can be found in Annex A.1. After updating and installing all needed packages, the repository of libbpf must be downloaded from the git repository. Additionally, libbpf-bootstrap repository can be downloaded optionally, but it is useful to verify the successful installation of libbpf and to use pre-built tools. After doing so the switch is ready for the code.

The metrics extraction in this work is carried out with two header files and two C files using libbpf. One of the header files is called "vmlinux.h". It entails the generated code of the currently running Linux kernel. The purpose of this file in our case is to retrieve the correct definitions of the task_struct. The other header file with the name "many_metrics.h" only entails a struct that is used as a ring buffer in the other two C scripts. The kernel script is one of the C codes mentioned. Its purpose is to extract the data directly from the kernel and thus to be opened, loaded, attached and closed by the user space script, the other specified C code. The kernel script is attached to the subsystem "perf_event" and is created by manually going through the metrics and nested structs of "task_struct". The data extraction is mostly done with the function "BPF_CORE_READ()". In the end, 1405 distinct metrics were extracted. This was just short of the stack limit of 512 bytes.

If we take a closer look at the sequence diagram visualized in the Figure 3.3, we find the typical libbpf pattern of open, load, attach and destroy. However, there are a few things that are worth noting. Firstly, the names of the functions before the double underlines often correspond to the file name. This is also the case with "many_metrics". In the case of the attachment for perf event's, however, it is not. There "bpf_program" is written for the correct function. Nevertheless, to get the codes to run, the kernel space code needs to be compiled manually from C into an object file and then into a header file with the help of bpftool. In our case, this is handled by the script "compile.sh", including the compilation of the user space code and creation of the "vimlinux.h". Additionally, the displayed function "perf_event_open()" is a custom function that uses a system call to retrieve the file descriptor for reading the desired data. This file descriptor and with the

skeleton received from "many_metrics__open()" are parameters for the perf event attachment. To keep the metric extraction running, one has to write a while true loop around the "ring_buffer__poll()" that checks if there is new data in the ring buffer submitted by the kernel code. As we wanted to have exact simulation runs, we implemented a timer here, in such a way that it jumps out of the loop after 4 hours. Then, the BPF program is detached from the kernel and cleaned up. With the space of the ring buffer being freed before.

If one wants to create its own code with libbpf, it is recommended to take a template and then adjust the code according to the needs. In our case, this is done with "profile.c" and "profile.bpf.c" from the libbpf-bootstrap GitHub repository https://github.com/libbpf/libbpf-bootstrap/tree/master/examples/c.

3.4 Simulation

The traffic of a network is normal distributed [13, 50]. During the simulation run, both hosts execute the scripts simultaneously and ping each other. Therefore, three different types of scripts were created, each simulating a desired scenario. A normal simulation that runs for 4 hours with a standard deviation of 30 and 300, and a normal distributed interval of 1 second with a standard deviation of 0.5.

The flooding simulation uses a fixed packet size of 65500 bytes with an interval of zero. This is used to replicate a DDoS attack, for example. 65500 bytes were chosen because they are very close to the maximum ping size of 65535 bytes in order to achieve maximal traffic load [54].

Six different delay simulations were run, starting from 300 bytes in the mean with increasing steps of 300 bytes to 1800 bytes, with the same standard deviation as the normal simulation, namely 30 and 300. Each simulation lasted 4 hours. The scripts running the Gaussian distributed simulations always check that the packet size is bigger than 64 bytes, as this is the minimum frame size of Ethernet. Otherwise, the normal distributed random number is recalculated. The same applies to an interval where the number is lower than zero. It gets redetermined. The simulation runs are summarized in the Table 3.5.

In order to have a suitable extraction rate, one would normally set a timer and then extract the data every second, for example. This is not possible when using libbpf. The extraction event gets executed according to the parameter "sample_period" in the perf_event_attr struct. During probing, it turned out that 305280 triggers a sampling event roughly every second when the switch is idle. This means that every 305280 perf_event is read from the kernel.

3.5 Intelligent Anomaly Detection

Each individual simulation run writes the captured data to its own CSV file, which must be cleaned up before being used for machine learning. A Python script is created for cleaning

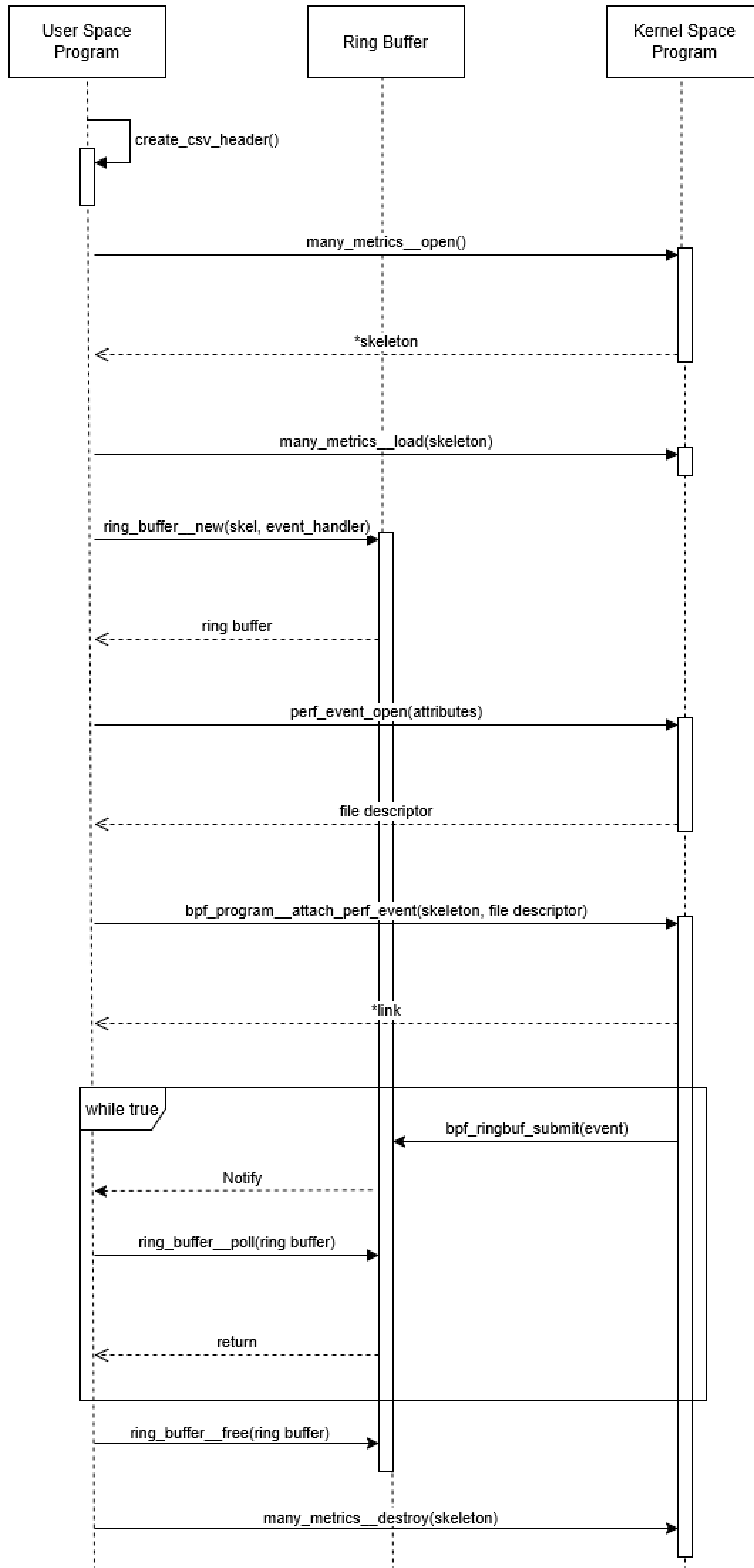

Figure 3.3: Sequence Diagram of eBPF Code

Table 3.5: Simulation Runs

Name	Packet size (PS)	PS std. dev.	Interval (Int)	Int std. dev.	Time	Samples
Normal	300 bytes	30	1 second	0.5	4h	81907
Delay	300 bytes 600 bytes 900 bytes 1200 bytes 1500 bytes 1800 bytes	30	5 seconds	0.5	4h	155970 166683 179993 181644 175384 162656
Flooding	65500 bytes	0	0 seconds	0	4h	9060
Normal	300 bytes	300	1 second	0.5	4h	176423
Delay	300 bytes 600 bytes 900 bytes 1200 bytes 1500 bytes 1800 bytes	300	5 seconds	0.5	4h	184679 84643 74810 87586 84524 85372
Flooding	65500 bytes	0	0 seconds	0	4h	8481

Table 3.6: Features Deleted form Both Simulation Runs

Metric	Number	Reasoning	Simulation Run
Temporal	146	'time' in the name	Packet size std. dev. 30 run
Constant	659	No change in value	Packet size std. dev. 30 run
Correlating	297	Correlation > 0.97	Packet size std. dev. 30 run
Total Removed	1102	-	Packet size std. dev. 30 run
Temporal	146	'time' in the name	Packet size std. dev. 300 run
Constant	677	No change in value	Packet size std. dev. 300 run
Correlating	294	Correlation > 0.97	Packet size std. dev. 300 run
Total Removed	1117	-	Packet size std. dev. 300 run

the data of the normal simulation run. It uses the pandas library and reads the normal CSV file. In the next step, the code removes the temporal features, meaning it removes all columns in the data set whose feature name contains "time". Further, all columns with strictly constant values were detached. Additionally, all features with a Pearson correlation higher than the threshold value of 0.97 were evacuated. The Python code then stores the cleaned DataFrame with the name "cleaned_normal.csv". Each removed feature is written in a text file called "bad_features.txt". In the end, the code creates a heatmap of the cleaned DataFrame using the Pearson correlation method and stores the image in the folder images with a representable name. Once the data set from the normal reproduction run has been cleaned, all other data sets from the same columns as the normal data set have been cleaned. This is done with the help of a separate Python code and the previously mentioned "bad_features.txt". Of the original 1405 features, only 303 features remained after the cleanup in the case of the standard deviation of 30 package sizes and 288 features in the case of the 300 standard deviation. Most of the columns were removed in both cases. As they only contained constant values during the simulations, namely 659 constant features. Further numbers are given in the Table 3.6.

The next step is to train the actual algorithms with another Python script. Therefore, the cleaned normal data set is split into a 90% training set and 10% holdout set. Then outliners with a higher Z-Score than three were removed from the trainings set. To improve the performance of the machine learning algorithms, all DataFrames of the distinct cleaned data sets are normalized with the standard scaler. Finally, unsupervised/semi-unsupervised algorithms such as LOF, IF and OCSVM were trained with a contamination factor of 0.05. Additionally, KNN and RF were also trained as a supervised machine learning algorithms.

Chapter 4

Evaluation and Results

This chapter presents the evaluation and results of the IDAPN approach. In the first section, we will discuss the extracted features. This is followed by the evaluation of the machine learning. In the end, we will conclude with a discussion about the evaluation and its limitations. For practical reasons, the simulation run with a packet size standard deviation of 300 is referred to as "Run1". The simulation with the packet size standard deviation of 30 is now introduced as "Run2".

4.1 Features Extraction

Of the 1405 extracted metrics, only 303 or 288 were used for training the machine learning algorithms. Therefore, the majority of the features were not useful for our incentive. Nevertheless, each simulation run has its own Pearson correlation heatmap, which was created after the machine learning preparation. What catches the eye first is the dominant color in the big picture, indicating that the extracted data has a correlation around zero. All heatmaps show a similar general color pattern, with the top left corner, left side and top hand being selectively more correlated. This means that the core relationship is stable, indicating its correctness. Slightly more than half of the maps have horizontal or vertical white lines. Sometimes they are really dominant, as shown in Figure 4.2. Sometimes, however, they are only slight, as shown in Figure 4.1. Sometimes there are no visual appearances as in Figure 4.3. All other generated heatmaps can be found in the Annex A.2.

Such white horizontal and vertical lines represent constant values in our case. In Run2, all attack simulations have such white lines. Since there seem to be only a few in Run1, this could be due to high diversification of packet sizes. Nevertheless, in both scenarios the flooding attack is clearly visible on the maps and shows the strongest deviations. If you look at the white lines of Run2, one can detect a pattern in the delay attacks with a packet size of more than 600 bytes, starting at the bottom. The vertical lines always appear near the features serial, count and stats_ac_exitcode and for the horizontal patterns len, my_q_throttle_count and stats_cpu_run_real_total. However, whether there is

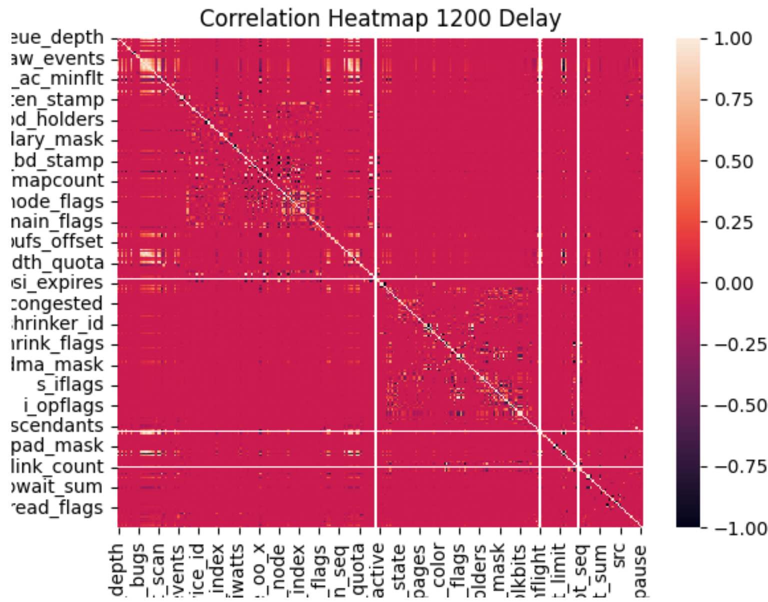

Figure 4.1: Heatmap 1200 bytes Delay Run1

something vital close to these features needs to be verified. Strangely enough, serial, len, stats_ac_exitcode and stats_cpu_run_real_total are exclusively contained in the Run2 data set, as can be seen in Table 4.1. Therefore, these features had to be discarded during the data preparation of the normal simulation Run1. This means that they had either temporal, constant or correlating values. A look at the stored names of the deleted features. It turns out that serial, stats_ac_exitcode and stats_cpu_run_real_total were deleted due to high correlation. However, the preparation script discarded len due to constant values. Nevertheless, if one writes a Pyhton script to compare the features of the delay scenarios that actually causes the white line patterns in Run2, you get the following results: tskgrp_nr_spread_over, audit_pid, stats_ac_ppid, len, numa_group_gid, stats_hiwater_rss, stats_blkio_count. If we also consider all attack simulations and analyze them for intersecting constant values, we obtain a single feature name: tskgrp_nr_spread_over. No intersecting constant values were found in Run1.

On the contrary, as one might think, there are far fewer samples created by attacks, which means a higher stress level on the switch. Even though, the metrics should be extracted once per 305280 events. This can be explained by the fact that the switch is under more stress so that it can no longer monitor every event. The Table 3.5 displays the number of samples for each simulation run.

4.2 ML Evaluation

In general, we used five different algorithms to detect anomalies in the created dataset. Isolation Forest and Local Outlier Factor are the unsupervised algorithms. The third algorithm is the semi-supervised One-Class Support Vector Machines. The fourth and fifth

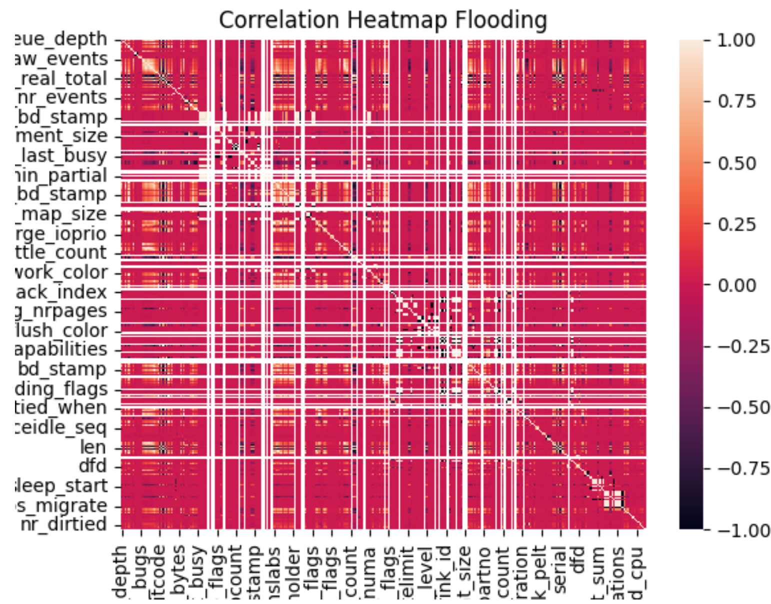

Figure 4.2: Heatmap Flooding Run2

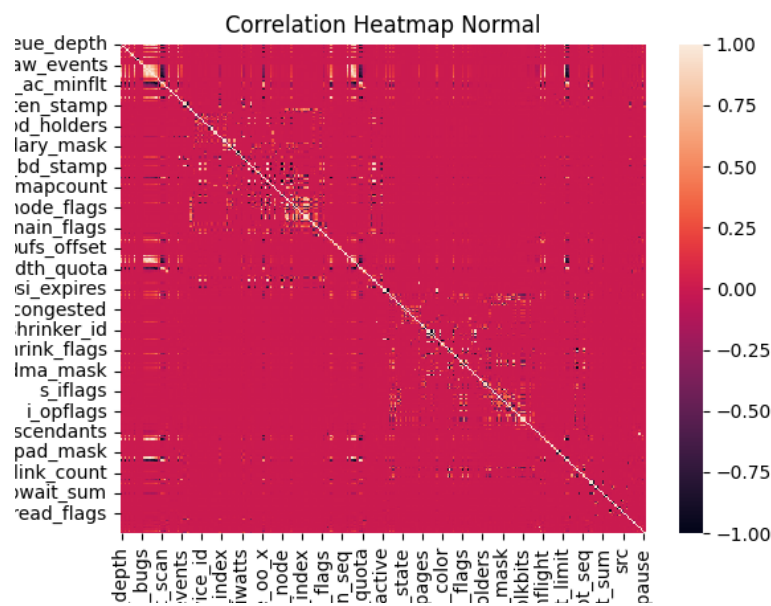

Figure 4.3: Heatmap Normal Run1

Table 4.1: Cleaned Features Differences

Features	Quantity	Names
Only Run1	24	base_cpu_active_bases_st, bio_autosuspend_delay biotail_io_tlb_mem_late_alloc, biotail_io_tlb_mem_nslabs biotail_io_tlb_mem_used, biotail_io_tlb_memfor_alloc cfs_bandwidth_quota, dev_io_tlb_mem_froce_bounce device_bus_dma_limit, ele_kmem_cache_cpu_tid, ele_oo_x gendisk_major, i_wb_dirtied_stamp, in_syscall nr_wakeups_sync , oom_mm_numa_next_scan owner_device_bus_dma_limit, s_bdi_id, s_fsnotify_mask s_iflags, s_inode_lru_memcg_aware, s_max_links s_readonly_remount, stats_swapin_count
Only Run2	39	audit_pid , base_cpu_clock_was_set_seq_st base_cpu_cpu_st , bio_device_bus_dma_limit, block_max cfs_bandwidth_burst , dev_bd_fsfreeze_count dev_bd_partno , dev_device_bus_dma_limit dev_device_id , dev_nr_perf_states dma_parms_max_segment_size , ele_kmem_cache_usersize fscrypt_operations_max_namelen , io_pages iowait_count , len , memcg_aware , nr_failed_migrations_hot nr_migrations_cold , nr_wakeups , numa_group_gid owner_dma_parms_max_segment_size , s_bdi_capabilities s_bdi_io_pages , s_bdi_max_prop_frac , s_bdi_max_ratio s_shrink_batch , s_writers_frozen , serial , slice_max stats_ac_exitcode , stats_ac_ppid , stats_blkio_count stats_cpu_run_real_total , stats_hiwater_rss tskgrp_nr_spread_over , workqueue_struct_nr_drainers workqueue_struct_saved_max_active

are the supervised algorithms, K-Nearest Neighbor and Random Forest. The first paragraph evaluates the results of Run2 and the second paragraph evaluates the performance of Run1.

Regarding Run2, as one can see in the Table 4.2, OCSVM achieves 100% in the detection of all anomalous behaviors. Unfortunately, the algorithm does not perform well in detecting normal behavior. 4339 out of 8191 cases were correctly detected. That is just over 50 percent. A relatively similar detection accuracy is achieved with LOF. Slightly more than half of the normal traffic is correctly categorized. However, it had more trouble detecting the delay attack with the same packet size as in the normal simulation. Out of 155970 samples, only 85499 were correctly identified, making LOF less effective than OCSVM. A look at IF shows that it performs well in detecting normal behavior, which is not the case for LOF and OCSVM. However, it is poor in detecting anomalous behavior. It is worth noting that the performance on the 300 bytes delay attacks is above 50%, although the other delay simulations do not work and detect a lot of normal behavior where no normal behavior should be. The supervised algorithm KNN performs very good. Detects anomalous behavior perfectly and normal behavior almost perfect with an accuracy of 90.27%. However, the other supervised algorithm trained RF is even better in identifying normal behavior but struggles on detecting 600 bytes and 1800 bytes delay attacks. No connections to the heatmaps could be established.

Moving no to Run1 a similar pattern is observed regarding OCSVM and LOF. Both perfectly detect anomalous behavior, but in a similar fashion as in Run2 even lack more in detecting normal behavior. Their performance is nearly identical, the only difference is that OCSVM performs a tiny bit better in detecting normal behavior then LOF, although neglectable. However, IF detects 15780 normal behaviors out of 17643 samples tested. This results in an accuracy of 89.44 percent. 5942 out of 8481 flooding samples were correctly identified. Then IF starts to perform poorly for example it detects only about a third of the 300 bytes delay attacks, half of the 900 bytes delay simulation and, in the case of 1200, bytes delay simulation 39834 out of 87586 samples were correctly identified. Nevertheless, in the case of 1500 bytes delay attack 67749 anomalies out of 84524 samples were correctly detected. In the case of 600 bytes, a performance of 68.68%. On the other hand, with the 1800 bytes delay run only 43.33% of the anomalies were correctly identified. As it was the case in Run2 the supervised algorithms perform better than the unsupervised algorithms. For example, RF has a flawless anomaly detection and a very close to 100% correct detection rate of normal behavior. The case with KNN on Run1 is very similar, they have the exact same performance for normal behavior. The only distinction to RF is, that KNN does not do well in detecting anomalous behavior in the case of 300 bytes delay attack. Only 0.29% of the samples were correctly identified.

4.3 Discussion and Limitations

The first paragraph gives a brief summary of the key results. In the second part the discussion and limitations of the metrics extraction are provided. In the third section the outcome and its limitations of the machine learning training are being concluded.

Table 4.2: Machine Learning Results

Algorithm	Run	Simulation	Normal detected	Anomalies detected
OCSVM	Run 2	Normal	4339	3852
OCSVM	Run 2	Flooding	0	9060
OCSVM	Run 2	Delay 300 bytes	0	155970
OCSVM	Run 2	Delay 600 bytes	0	166683
OCSVM	Run 2	Delay 900 bytes	0	179993
OCSVM	Run 2	Delay 1200 bytes	0	181644
OCSVM	Run 2	Delay 1500 bytes	0	175384
OCSVM	Run 2	Delay 1800 bytes	0	162656
LOF	Run 2	Normal	4357	3834
LOF	Run 2	Flooding	0	9060
LOF	Run 2	Delay 300 bytes	70471	85499
LOF	Run 2	Delay 600 bytes	0	166683
LOF	Run 2	Delay 900 bytes	0	179993
LOF	Run 2	Delay 1200 bytes	0	181644
LOF	Run 2	Delay 1500 bytes	0	175384
LOF	Run 2	Delay 1800 bytes	0	162656
IF	Run 2	Normal	7183	1008
IF	Run 2	Flooding	5545	3515
IF	Run 2	Delay 300 bytes	70471	85499
IF	Run 2	Delay 600 bytes	134803	31880
IF	Run 2	Delay 900 bytes	150428	29565
IF	Run 2	Delay 1200 bytes	155635	26009
IF	Run 2	Delay 1500 bytes	141380	34004
IF	Run 2	Delay 1800 bytes	129881	32775
KNN	Run 2	Normal	7394	797
KNN	Run 2	Flooding	0	9060
KNN	Run 2	Delay 300 bytes	0	155970
KNN	Run 2	Delay 600 bytes	0	166683
KNN	Run 2	Delay 900 bytes	0	179993
KNN	Run 2	Delay 1200 bytes	0	181644
KNN	Run 2	Delay 1500 bytes	0	175384
KNN	Run 2	Delay 1800 bytes	0	162656
RF	Run 2	Normal	8112	79
RF	Run 2	Flooding	0	9060
RF	Run 2	Delay 300 bytes	0	155970
RF	Run 2	Delay 600 bytes	163828	2855
RF	Run 2	Delay 900 bytes	0	179993
RF	Run 2	Delay 1200 bytes	0	181644
RF	Run 2	Delay 1500 bytes	0	175384
RF	Run 2	Delay 1800 bytes	160931	1725
OCSVM	Run 1	Normal	4337	13306
OCSVM	Run 1	Flooding	0	8481
OCSVM	Run 1	Delay 300 bytes	0	184679
OCSVM	Run 1	Delay 600 bytes	0	84643
OCSVM	Run 1	Delay 900 bytes	0	74810
OCSVM	Run 1	Delay 1200 bytes	0	87586
OCSVM	Run 1	Delay 1500 bytes	0	84524
OCSVM	Run 1	Delay 1800 bytes	0	85372
LOF	Run 1	Normal	4324	13319
LOF	Run 1	Flooding	0	8481
LOF	Run 1	Delay 300 bytes	0	184679
LOF	Run 1	Delay 600 bytes	0	84643
LOF	Run 1	Delay 900 bytes	0	74810
LOF	Run 1	Delay 1200 bytes	0	87586
LOF	Run 1	Delay 1500 bytes	0	84524
LOF	Run 1	Delay 1800 bytes	0	85372
IF	Run 1	Normal	15780	1863
IF	Run 1	Flooding	2539	5942
IF	Run 1	Delay 300 bytes	119734	64945
IF	Run 1	Delay 600 bytes	26596	58047
IF	Run 1	Delay 900 bytes	35733	39077
IF	Run 1	Delay 1200 bytes	47752	39834
IF	Run 1	Delay 1500 bytes	16775	67749
IF	Run 1	Delay 1800 bytes	48378	36994
KNN	Run 1	Normal	17590	53
KNN	Run 1	Flooding	0	8481
KNN	Run 1	Delay 300 bytes	184148	531
KNN	Run 1	Delay 600 bytes	0	84643
KNN	Run 1	Delay 900 bytes	0	74810
KNN	Run 1	Delay 1200 bytes	0	87586
KNN	Run 1	Delay 1500 bytes	0	84524
KNN	Run 1	Delay 1800 bytes	0	85372
RF	Run 1	Normal	17590	53
RF	Run 1	Flooding	0	8481
RF	Run 1	Delay 300 bytes	0	184679
RF	Run 1	Delay 600 bytes	0	84643
RF	Run 1	Delay 900 bytes	0	74810
RF	Run 1	Delay 1200 bytes	0	87586
RF	Run 1	Delay 1500 bytes	0	84524
RF	Run 1	Delay 1800 bytes	0	85732

A high standard deviation in the packet size in a normal behavior seems to have great effects on the intelligent detection of anomalous behavior for unsupervised algorithms and heatmaps. There are great visual distinctions from a small standard deviation and a higher standard deviation of packet sizes.

The heatmaps of the simulation runs show promising results if the standard deviation of the packet size is not too big. If one chooses to have a big standard deviation the data tends to get scatter all over the place and thus obfuscating anomalous behavior. There is also an indication that one can survey packet size increases of 300 bytes with heatmaps alone. This would mean to survey the traffic load of the switch in a very nuanced way. Nevertheless, the correlation maps of the flooding attack drastically deviate from the norm. Thus, should make it easy for the artificial intelligence to detect such types of attacks. Regarding the white lines in the heatmap there could not be a pattern detected in Run1 as in Run2. Therefore, if the standard deviation of the packet sizes changes different types of features get used for the ML algorithms. Regarding the detected pattern, the features who participate in the visual distinctions are mostly deleted in Run1 because they showed a high correlation. The intersecting constant feature of all attack simulations in Run2 is tskgrp_nr_spread_over. Nevertheless, the stack limit of 512 bytes makes it hard to extract every single resource metric from the switch. Thus, disabling to get a clear overview of all metrics together, how they behave and relate to eachother.

As one can see in the Table 4.2, flooding always has the least number of samples. Even though, it should technically have the most metrics events, due to the higher activity of the switch. Quick reminder, the data gets extracted every 305280 event. Since it was found to be extracting roughly every second in idling state. A decline in the number of samples starting from 1500 bytes delay simulation until flooding has been found. Indicating that the switch reduces the extraction while being under stress. This can have negative effects on detecting anomalies. Since in our example we are creating only heavy loads with two hosts. With lots of attacking devices the switches metric extraction may get disabled. Thus, potentially resulting in no data for the ML algorithm. If no precautionary protections were made, a DDoS attack might go unnoticed. Also, our results show that RF delivers the best performance in a network with highly deviating packet sizes and KNN is best to use in an environment with less fluctuating packet sizes. However, both supervised algorithms outperformed the unsupervised/semi-supervised algorithms in case of detecting normal behavior. Where overall similar perfect performance in detecting anomalous behavior was detected. Nevertheless, IF does not seem to work in this works settings. In some cases high packet sizes fluctuations seem to level out some subtle attacks, as it is the case with KNN in Run1.

Chapter 5

Final Considerations

This chapter first provides the reader with a summary of the main factors considered in this work. The next section describes the findings and considerations in more detail. Suggestions for future work are made in the final sections.

5.1 Summary

The first paragraph describes how the objectives of this thesis were achieved. The next paragraph summarizes the main considerations and conclusions. The last section summarizes the main difficulties and challenges.

The incentive was to create a framework for realistic extraction of resource metrics with eBPF in a programmable network. This was successfully completed. Nevertheless, the first objective was to get an overview of the state of the art. To this end, a background and literature analysis was done by reading topic-related papers. In order to fulfill the next objective of choosing an appropriate environment, a review table was created. The outcome was to go with P4Docker, because it has a Graphical User Interface (GUI) for the creation of an P4-enabled environment with Docker containers. This is exactly what we were looking for. To familiarize ourselves with P4, the exercises provided by P4 were performed. The exercises can found on https://github.com/p4lang/tutorials/tree/master/exercises. Then a review table was created on which tools are best to use. The decision was made in favor of the libbpf library, due to it being the state of the art. For the next objective, a P4-enabled BMv2 switch base image and two Docker containers with the same Ubuntu base image were created in a star-like topology using P4Docker. Next the switch was made eBPF ready.

After manually testing everything in the switch's container, a custom Dockerfile was created. To be able to extract metrics directly from the Linux kernel, C scripts were written using the libbpf library. Once the data could be extracted from the kernel, three different types of simulation scripts were created. The two Ubuntu hosts ping each other through the host with different packet sizes and intervals to demonstrate different scenarios. However, a little quirk was implemented. Instead of always sending with the same parameters,

the variables were made Gaussian distributed using the GNU Scientific Library, as the network traffic is normal distributed. The next objective was to run the simulation and extract live metrics. One normal and two attack reproduction were run. All previously achieved objectives were combined. Then the data from the normal run was cleaned for temporal, constant and highly correlated features. The exact same features were deleted on the data sets of the other attack simulation runs. This was done to train the machine learning algorithms in the next step. The chosen algorithms were IF, LOF, OCSVM, KNN and RF. The mixture of supervised and unsupervised algorithms was done to see if any of them perform better. Thus, the last objective of this work has been successfully completed.

The work shows that it is possible to draw conclusions by only looking at the resource metrics. For example, you can clearly see the stress level of the switch in the heatmaps. Nevertheless, the load of the switch can be obfuscated by sending many different packet sizes, so that the attack remains undetected in visualizations for longer. Reducing the frequency of extraction events can be a valid alternative for detecting a high stress inside the switch. This should be considered whether all types of flooding attacks should be detected. Such high stress could be placed on the switch that it is no longer able to extract metrics. This would make intelligent detection obsolete. On the other hand it could be used to recognize stress levels. Additionally, the found pattern of features within the delay attack could be a promising feature set for a lean monitoring system. For now, the intelligent anomalies detection perfectly detects anomalous behavior but struggles on identifying normal behavior. OCSVM is found to be most promising among the unsupervised algorithms, it perfectly detects anomalous behavior but struggles on identifying normal behavior. The supervised algorithms were found to perform better in general. KNN detects normal and anomalous behavior almost perfectly. The only drawback it has is in detecting 300 bytes delay attacks. Meaning the slowdown in receiving packets could not be detected in the case of highly fluctuating packet sizes. RF has also almost a flawless performance detecting with high fluctuating packet sizes. On the other hand, RF has problems detecting 600 and 1800 bytes delay attacks other than that its performance can be considered flawless.

The main difficulty was to write the extraction script with libbpf. After countless hours invested, it is best to write a C script with the libbpf library. This is by getting a code that does something similar to what you intend to do. You should start from there. It is recommended to use the code from libbpf-bootstrap for such purposes. Nevertheless, it was also quite a challenge to get the correct packages on the Docker container. One can find all needed packages in the Annex A.1. If the intention is to do metrics extraction in a Docker container, it must be remembered that the Docker image deployed in the container and the host machine run the same kernel version. Additionally, before running the extracting container the memory lock limit has to be removed, the system folder of the host machine has to be mounted, and it has to be run in privileged mode. Another challenge in this thesis is that the IPv4 forwarding is disabled by default in the P4Docker configuration script. It is best to open the shell script to create the environment and change the variable of IPv4 forwarding from zero to one.

5.2 Conclusions

The framework provides an approach for reading resource metrics directly from the Linux kernel. This allows many distinct data points to be read. Nevertheless, only a fraction of these can be used for training machine learning algorithms. The fraction even consists out of 303 or 288 features. This enables machine learning with many more features than usual. It is nice to see that the heatmaps change when the workload of the switch changes. However, the visible effect on the heatmaps gets damped if the packet size is too scattering. A flooding attack is always very visible, and a delay attack tends to be more visible the more workload it causes. Nevertheless, among the delay attacks, the 1200 bytes packet size was found to be the most visible on the heatmaps and also created the most samples during the simulation. This is because the switch reduces the metric extraction via eBPF when the workload gets too big. This results in fewer samples and fewer representation values.

Since it was originally set that the values get extracted every 305280 perf event, but under stress this number might be not accurate. This can be surveilled for the flooding attacks and a bigger packet size than 1200 bytes in the case of delay simulation. 1200 bytes was the tipping point of the switch in reducing the metrics extraction in our case.

Nevertheless, care must be taken as an attacker could disable the metric extraction if they manage to drastically increases the load on the switch and thus disable the intelligent anomaly detection. Additionally, a pattern was detected when running on the small packet size deviation. However, due to high correlation, these features were removed in the run with the higher deviation of the packet size. The above findings maybe also apply to the stress of other Linux machines not exclusively to a switch. Since libbpf can be used on most Linux kernels.

The unsupervised machine learning algorithms had similar results regardless of the standard deviation of the packet size. OCSVM and LOF detect anomalous behavior with 100% accuracy. Nevertheless, they still lack the ability to detect normal behavior, correctly detecting about half of normal behavior with a low standard deviation in packet size. The high standard deviation run performs worse. IF lacks in detecting anomalous behavior, whereas it is better at detecting normal behavior. The supervised algorithms do have the best performance. KNN shines in a network where the packet sizes to not vary much and RF is found to perform best in a network with high fluctuating packet sizes.

5.3 Future Work

The only limitations of the metrics extraction program are the stack limit of 512 bytes, and it must have a Linux kernel. Due to this reason, it would be interesting to see, where else intelligent anomalies detection can be performed and how their performance is related to each other. This, to ultimately find the best components of a network to detect incidents for different scenarios. Additionally, one could do an analysis of all possible

metrics to extract and analyze which ones perform best for machine learning training. Nevertheless, instead of linking the Docker container to an environment, one could also use Docker compose and change the image of the BMv2 switch to an OVS. To find if there is a distinction in using a switch that is used in real life networks.

Ultimately, it is important to mimic the traffic load of a network in order to achieve a representative traffic load. Therefore, future work could analyze different types of network loads, focusing on packet size, interval and their standard deviations. It would make sense to make an additional distinction between a small network and an enterprise network. Nevertheless, an important step for the future work is the improvement and fine tuning of the ML algorithms for better results. Any further developed product or framework should also take into account, that all kinds of attacks are possible. Meaning, other network traffic instead of TCP SYN should be considered.

Bibliography

[1] Nikos Anerousis et al. "The Origin and Evolution of Open Programmable Networks and SDN". In: *IEEE Communications Surveys & Tutorials* 23.3 (2021), pp. 1956–1971.

[2] Sherry Bai, Hyojoon Kim, and Jennifer Rexford. "Passive OS fingerprinting on commodity switches". In: *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*. 2022, pp. 264–268.

[3] Pat Bosshart et al. "P4: programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95.

[4] K.L. Calvert et al. "Directions in active networks". In: *IEEE Communications Magazine* 36.10 (Oct. 1998), pp. 72–78.

[5] Andrew T. Campbell et al. "A survey of programmable networks". In: *ACM SIGCOMM Computer Communication Review* 29.2 (Apr. 1999), pp. 7–23.

[6] Andrew T Campbell et al. "Open signaling for ATM, internet and mobile networks (OPENSIG'98)". In: *Computer Communication Review* (Oct. 1998).

[7] Jeison C. Caroly, Eder J. Scheid, and Lisandro Z. Granville Muriel F. Franco. "Securing Blockchain Wallet Files Using eBPF". In: *Global Communications Conference (GLOBECOM 2024)*. 2024, pp. 1–6.

[8] Alberto Huertas Celdrán et al. "Intelligent and behavioral-based detection of malware in IoT spectrum sensors". In: *International Journal of Information Security* 22.3 (June 2023), pp. 541–561.

[9] Raghavendra Chalapathy and Sanjay Chawla. *Deep Learning for Anomaly Detection: A Survey*. Jan. 2019. URL: `http://arxiv.org/abs/1901.03407` (visited on Sept. 25, 2024).

[10] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey". In: *ACM Computing Surveys* 41.3 (July 2009), pp. 1–58.

[11] *Cilium - Cloud Native, eBPF-based Networking, Observability, and Security*. URL: `https://cilium.io` (visited on Feb. 19, 2025).

[12] *Containernet*. URL: `https://containernet.github.io/` (visited on Feb. 18, 2025).

[13] Remco Van De Meent, Michel Mandjes, and Aiko Pras. "Gaussian traffic everywhere?" In: *2006 IEEE International Conference on Communications*. Vol. 2. June 2006, pp. 573–578.

[14] *Docker: Accelerated Container Application Development*. May 2022. URL: `https://www.docker.com/` (visited on Feb. 18, 2025).

[15] E. J. Scheid L. Z. Granville E. V. Oliveira M. F. Franco. *P4thTest: an Approach for the Automation of Tests for Programmable Networks and P4 Programs*. 2025.

[16] Roman Ferrando and Paul Stacey. "Classification of device behaviour in internet of things infrastructures: towards distinguishing the abnormal from security threats". In: *Proceedings of the 1st International Conference on Internet of Things and Machine Learning*. Oct. 2017, pp. 1–7.

[17] Vaishali Ganganwar. "An overview of classification algorithms for imbalanced datasets". In: *International Journal of Emerging Technology and Advanced Engineering* 2.4 (2012).

[18] Rik Goldman. *Learning Proxmox VE*. Mar. 2016.

[19] Ajay Gulati et al. "Vmware distributed resource management: Design, implementation, and lessons learned". In: *VMware Technical Journal* 1.1 (2012), pp. 45–64.

[20] Kyle Haefner and Indrakshi Ray. "ComplexIoT: Behavior-based trust for IoT networks". In: *2019 First IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. 2019, pp. 56–65.

[21] Alexandre da Silveira Ilha et al. "Euclid: A Fully In-Network, P4-Based Approach for Real-Time DDoS Attack Detection and Mitigation". In: *IEEE Transactions on Network and Service Management* 18.3 (Sept. 2021), pp. 3121–3139.

[22] *iovisor/bcc*. Feb. 2025. URL: `https://github.com/iovisor/bcc` (visited on Feb. 18, 2025).

[23] Karamjeet Kaur, Japinder Singh, and Navtej Singh Ghumman. "Mininet as software defined networking testing platform". In: *International conference on communication, computing & systems (ICCCS)*. 2014, pp. 139–42.

[24] *Kubernetes Monitoring with Grafana*. URL: `https://grafana.com/solutions/kubernetes/` (visited on Feb. 19, 2025).

[25] Eliot Lear, Ralph Droms, and Dan Romascanu. *Manufacturer usage description specification*. Tech. rep. RFC Editor, 2019.

[26] *libbpf/bpftool*. Feb. 2025. URL: `https://github.com/libbpf/bpftool` (visited on Feb. 18, 2025).

[27] *libbpf/libbpf*. Feb. 2025. URL: `https://github.com/libbpf/libbpf` (visited on Feb. 18, 2025).

[28] Chang Liu et al. "A protocol-independent container network observability analysis system based on eBPF". In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2020, pp. 697–702.

[29] Daniel F. Macedo et al. "Programmable networks-From software-defined radio to software-defined networking". In: *IEEE communications surveys & tutorials* 17.2 (2015), pp. 1102–1125.

[30] Giuseppe Manco et al. "Fault detection and explanation through big data analysis on sensor streams". In: *Expert Systems with Applications* 87 (2017), pp. 141–156.

[31] Samuel Marchal et al. "Audi: Toward autonomous iot device-type identification using periodic communication". In: *IEEE Journal on Selected Areas in Communications* 37.6 (2019), pp. 1402–1412.

[32] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (Mar. 2008), pp. 69–74.

[33] Dubravko Miljković. "Review of novelty detection methods". In: *The 33rd International Convention MIPRO*. 2010, pp. 593–598.

[34] Charles G. Miller. *Telephone switching system*. 1965. URL: `https://scholar.google.com/scholar?hl=de&as_sdt=0%2C5&q=C.+G.+Miller%2C+%E2%80%`

9CTelephone+switching+system%2C%E2%80%9D+U.S.+Patent+3+189+687+A%2C+Jun.+15%2C+1965&btnG= (visited on Nov. 11, 2024).

[35] *Mininet: An Instant Virtual Network on Your Laptop (or Other PC) - Mininet.* URL: http://mininet.org/ (visited on Feb. 19, 2025).

[36] John Moy. *OSPF version 2.* Tech. rep. IETF, 1997.

[37] Bruno Astuto A. Nunes et al. "A survey of software-defined networking: Past, present, and future of programmable networks". In: *IEEE Communications surveys & tutorials* 16.3 (2014), pp. 1617–1634.

[38] K. C. Okafor et al. "Leveraging Fog Computing for Scalable IoT Datacenter Using Spine-Leaf Network Topology". In: *Journal of Electrical and Computer Engineering* 2017 (2017), pp. 1–11.

[39] *Oracle VirtualBox.* URL: https://www.virtualbox.org/ (visited on Feb. 19, 2025).

[40] Atsuya Osaki et al. "Dynamic Fixed-point Values in eBPF: a Case for Fully In-kernel Anomaly Detection". In: *Proceedings of the Asian Internet Engineering Conference 2024.* 2024, pp. 46–54.

[41] Manuel Poisson, Rodrigo Carnier, and Kensuke Fukuda. "GothX: a generator of customizable, legitimate and malicious IoT network traffic". In: *Proceedings of the 17th Cyber Security Experimentation and Test Workshop.* Aug. 2024, pp. 65–73.

[42] *Production-Grade Container Orchestration.* URL: https://kubernetes.io/ (visited on Feb. 18, 2025).

[43] *Proxmox Server Solutions.* URL: https://www.proxmox.com/en/ (visited on Feb. 18, 2025).

[44] Yakov Rekhter, Tony Li, and Susan Hares. *A border gateway protocol 4 (BGP-4).* Tech. rep. IETF, 2006.

[45] Minato Sakuraba et al. "An Anomaly Detection Approach by AIML in IP Networks with eBPF-Based Observability". In: *2023 24st Asia-Pacific Network Operations and Management Symposium (APNOMS).* Sept. 2023, pp. 171–176.

[46] Pedro Miguel Sánchez Sánchez et al. "A survey on device behavior fingerprinting: Data sources, techniques, application scenarios, and datasets". In: *IEEE Communications Surveys & Tutorials* 23.2 (2021), pp. 1048–1077.

[47] Matheus Saueressig et al. "An Approach for Behavioral Fingerprinting of P4 Programmable Switches". In: *Anais da XX Escola Regional de Redes de Computadores (ERRC 2023).* Oct. 2023, pp. 55–60.

[48] Matheus Saueressig et al. "FEVER: Intelligent Behavioral Fingerprinting for Anomaly Detection in P4-Based Programmable Networks". In: *Advanced Information Networking and Applications.* Vol. 201. 2024, pp. 362–373.

[49] Dener Silva. *Overview | P4Docker.* Apr. 2024. URL: https://dnredsons-organization.gitbook.io/p4docker (visited on Feb. 18, 2025).

[50] Antoine Varet and Nicolas Larrieu. "How to Generate Realistic Network Traffic?" In: *2014 IEEE 38th Annual Computer Software and Applications Conference.* July 2014, pp. 299–304.

[51] Marcos A. M. Vieira et al. "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications". In: *ACM Computing Surveys* 53.1 (Dec. 2019), pp. 1–36.

[52] *VMware vSphere Virtualization Platform.* URL: https://www.vmware.com/products/cloud-infrastructure/vsphere (visited on Feb. 18, 2025).

[53] Qinshi Wang et al. “Foundational Verification of Stateful P4 Packet Processing”. In: *LIPIcs, Volume 268, ITP 2023* 268 (2023), 32:1–32:20.

[54] Fekadu Yihunie, Eman Abdelfattah, and Ammar Odeh. “Analysis of ping of death DoS and DDoS attacks”. In: *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*. May 2018, pp. 1–4.

[55] Xiangzhe Zhang, Zhaoyuan Liu, and Jiaqing Bai. “Linux Network Situation Prediction Model Based on eBPF and LSTM”. In: *2021 16th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*. Nov. 2021, pp. 551–556.

[56] Zhuping Zou et al. “A Docker Container Anomaly Monitoring System Based on Optimized Isolation Forest”. In: *IEEE Transactions on Cloud Computing* 10.1 (Jan. 2022), pp. 134–145.

Abbreviations

ML	Machine Learning
DL	Deep Learning
eBPF	extended Berkeley Packet Filter
AWS	Amazon Web Service
KNN	k-Nearest Neighbors
RF	Random Forest
CSG	Communication Systems Group
SDN	Software-defined Networking
DDoS	Distributed Denial of Service
DoS	Denial of Service
P4	Programming Protocol-independent Packet Processor
OS	Operating System
IETF	Internet Engineering Task Force
OPENSIG	Open Signaling
AN	Active Networks
GSMP	General Switch Management Protocol
API	Application Programming Interface
MUD	Manufacturer Usage Description
DT	Decision Tree
LR	Logistic Regression
NB	Naive Bayes
SVM	Support Vector Machine
t-SNE	t-distributed Stochastic Neighbor Embedding
PCA	Principal Component Analysis
DBSCAN	density-based spatial clustering of applications with noise
OCSVM	One-Class Support Vector Machine
IF	Isolation Forest
BMv2	Behavioral Model version 2
MRI	Multi-Hop Route Inspection
LOF	Local Outlier Factor
RSS	Resident Set Size
OW	Observation Window
COTS	Commercial-Off-The-Shelf
INT	In-band Network Telemetry
PID	Process Identifier
IoT	Internet of Things

BGP	Border Gateway Protocol
OSPFv2	Open Shortest Path First version 2
OSS	Operation Support System
LSTM	Long Short-Term Memory
JSON	JavaScript Object Notation
GUI	Graphical User Interface
FTTH	Fiber to the Home
MitM	Man in the Middle
OVS	Open vSwitch
vEth	virtual Ethernet
DNS	Domain Name System
CPU	Central Processing Unit
GPU	Graphics Processing Unit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
HTTP	Hypertext Transfer Protocol

List of Figures

3.1 IDAPN Overview 17

3.2 Topology of the Environment Built to Support the Thesis Development . . 18

3.3 Sequence Diagram of eBPF Code 23

4.1 Heatmap 1200 bytes Delay Run1 28

4.2 Heatmap Flooding Run2 29

4.3 Heatmap Normal Run1 29

A.1 Heatmap Normal Run2 51

A.2 Heatmap 300 bytes Delay Run1 51

A.3 Heatmap 300 bytes Delay Run2 51

A.4 Heatmap 600 bytes Delay Run1 52

A.5 Heatmap 600 bytes Delay Run2 52

A.6 Heatmap 900 bytes Delay Run1 52

A.7 Heatmap 900 bytes Delay Run2 52

A.8 Heatmap 1200 bytes Delay Run2 52

A.9 Heatmap 1500 bytes Delay Run1 53

A.10 Heatmap 1500 bytes Delay Run2 53

A.11 Heatmap 1800 bytes Delay Run1 53

A.12 Heatmap 1800 bytes Delay Run2 53

A.13 Heatmap Flooding Run1 53

List of Tables

2.1 Literature Review 13

3.1 Environment Review 19

3.2 Operating System Comparison 19

3.3 Topology Review 20

3.4 Data Extraction Review 21

3.5 Simulation Runs 24

3.6 Features Deleted form Both Simulation Runs 24

4.1 Cleaned Features Differences 30

4.2 Machine Learning Results 32

A.1 Cleaned Features in Both Datasets 50

Appendix A

Contents of the Repository

The code repository contains the following content: The entire code is available to the public at https://github.com/dattes/IDAPN.

A.1 Installation

Packages to install in order to work with libbpf.

- linux-tools-common
- linux-tools-generic
- linux-headers-‘uname -r‘
- linux-tools-‘uname -r‘
- libelf-dev
- zlib1g-dev
- libbpf-dev
- clang-12
- llvm-12
- gcc-multilib
- pkg-config
- git
- gcc

In this table one can see all features that were used in both simulations for the machine learning training.

Table A.1: Cleaned Features in Both Datasets

Features	Quantity	Names
Always present	264	__rcu_icq_cache_cpu_partial , __rcu_icq_cache_offset , __state , ac_mem , active_memcg_soft_limit bd_holders , bd_openers , bd_partno , bd_read_only , bd_stamp , bd_write_holder , bi_iocost_cost bio_bd_fsfreeze_count , bio_bd_holders , bio_bd_partno , bio_bd_read_only , bio_bd_stamp bio_bd_write_holder , bio_device_coherent_dma_mask , bio_device_id , bio_device_numa_node bio_driver_flags , bio_io_tlb_mem_nslabs , bio_io_tlb_mem_used , bio_msi_domain_mapcount biotail_autosuspend_delay , biotail_device_node_flags , biotail_dma_parms_max_segment_size biotail_dma_parms_min_align_mask , biotail_dma_parms_segment_boundary_mask biotail_dma_range_map_offset , biotail_dma_range_map_size , biotail_driver_flags biotail_io_tlb_mem_froce_bounce , biotail_io_tlb_mem_index , biotail_last_busy , biotail_milliwatts biotail_msi_domain_flags , block_start , bufs_offset , bug_table_line , cleancache_poolid , cpu cpu_pwqs_max_active , cpu_pwqs_nr_active , cpu_pwqs_refcnt , dev_autosuspend_delay dev_bd_holders , dev_bd_openers , dev_bd_read_only , dev_bd_stamp , dev_bd_write_holder dev_device_coherent_dma_mask , dev_device_node_flags , dev_device_numa_node dev_dma_parms_max_segment_size , dev_dma_parms_min_align_mask dev_dma_parms_segment_boundary_mask , dev_dma_range_map_offset , dev_dma_range_map_size dev_driver_flags , dev_io_tlb_mem_index , dev_io_tlb_mem_late_alloc , dev_io_tlb_mem_nslabs dev_io_tlb_mem_used , dev_last_busy , dev_milliwatts , dev_msi_domain_flags dev_msi_domain_mapcount , dev_msi_domain_revmap_size , device_coherent_dma_mask device_numa_node , dfd , dirtied_when , disk_diskseq , disk_major , dma_pad_mask dma_parms_min_align_mask , dma_range_map_offset , done , ei_funcs_etype ele_kmem_cache_min_partial , ele_kmem_cache_size , elevator_owner_num_bpf_raw_events elevator_owner_num_bugs , exec_max , expires , extable_data , flags , forceidle_seq gendisk_flags , gendisk_state , pl_syms_name_offset , gpl_syms_value_offset , hw_dir_sd_mkobj_rev hw_sd_holders_dir_id , hw_sd_kobj_flags , i_acl_a_count , i_blkbits , i_bytes i_crypt_info_ci_dirhash_key_initialized , i_crypt_info_ci_owns_key , i_data_flags , i_data_nrpages i_data_writeback_index , i_default_acl_a_count , i_flags , i_fsnotify_marks_flags i_fsnotify_marks_type , i_fsnotify_mask , i_generation , i_ino , i_mapping_flags , i_mapping_nrpages i_mapping_writeback_index , i_opflags , i_wb_avg_write_bandwidth , i_wb_balanced_dirty_ratelimit i_wb_congested , i_wb_dirty_exceeded , i_wb_dirty_ratelimit , i_wb_dirty_sleep , i_wb_frn_history i_wb_frn_winner , i_wb_last_old_flush , i_wb_state , i_wb_write_bandwidth , i_wb_written_stamp i_write_hint , idle_h_nr_running , init_layout_text_size , inode_ratelimit_flags , iowait_sum last_merge_ioprio , last_merge_write_hint , last_type , level , m_seq , max_open_zones mg_dst_cgrp_max_descendants , mg_dst_cgrp_nr_descendants mg_dst_cgrp_nr_populated_domain_children , min_flt , mmap_legacy_base , mq_ctx_flags mq_ctx_queued , msi_domain_mapcount , my_q_propagate , my_q_throttle_count , my_q_throttled nameidata_flags , nameidata_state , next_fd , notes_attrs_notes , nr_dirtied , nr_dirtied_pause nr_failed_migrations_affine , nr_failed_migrations_running , nr_forced_migrations , nr_wakeups_affine nr_wakeups_migrate , oom_mm_end_code , oom_mm_hiwater_vm , oom_mm_mmap_base oom_mm_start_brk , oom_mm_start_data , oom_mm_user_ns_flags , owner_autosuspend_delay owner_bd_fsfreeze_count , owner_bd_holders , owner_bd_openers , owner_bd_partno owner_bd_read_only , owner_bd_stamp , owner_bd_write_holder , owner_device_coherent_dma_mask owner_device_id , owner_device_numa_node , owner_driver_flags , owner_io_tlb_mem_index owner_last_busy , page__compound_pad_1 , pagefault_disabled , propagate , psi_expires psi_group_avg_last_update , psi_group_polling_until , pt_mm_exec_vm , pt_mm_highest_vm_end pt_mm_numa_scan_seq , pt_mm_stack_vm , pt_mm_startup_done , quotalen , real_percpu_count real_unix_inflight , recent_used_cpu , reclaimed_slab , root_seq , rt_base_cpu_nr_events rt_base_index_st , s_bdi_min_ratio , s_bdi_ra_pages , s_blocksize , s_blocksize_bits s_cancelled_write_bytes , s_count , s_dentry_lru_memcg_aware , s_dentry_lru_shrinker_id s_dquot_flags , s_encoding_flags , s_inode_lru_shrinker_id , s_magic , s_quota_types , s_read_bytes s_shrink_flags , s_shrink_id , s_shrink_seeks , s_stack_depth , s_syscr , s_wchar , s_write_bytes sas_ss_flags , seq , shrinker_id , slab_kmem_cache_red_left_pad , sleep_max , sleep_start special_vec_bv_len , src , state , stats_ac_minflt , stats_freepages_count , swapin_delay tag_set_queue_depth , thread_flags , thread_status , throttle_queue_id , throttled_clock_pelt total_link_count , tskgrp_ld_inv_weight , tskgrp_ld_weight , unbound_attrs_nice unbound_attrs_no_numa , unicode_map_version , uprobe_depth , user_namespace_flags user_namespace_level , user_namespace_parent_could_setfcap , util_avg , wait_count , wait_max wait_start , wait_sum , wakee_flips , wb_i_wb_last_old_flush , wb_i_wb_state wb_i_wb_written_stamp , workqueue_struct_flags , workqueue_struct_flush_color workqueue_struct_work_color , wq_flags , wq_nr_drainers , wq_saved_max_active , wq_work_color

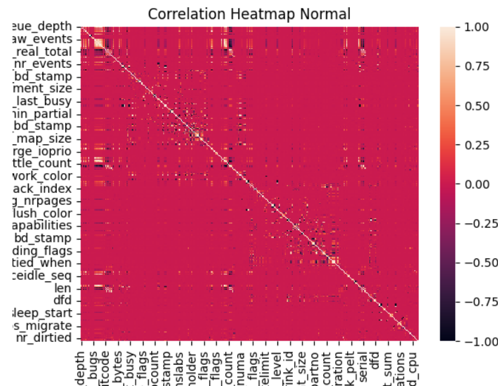

Figure A.1: Heatmap Normal Run2

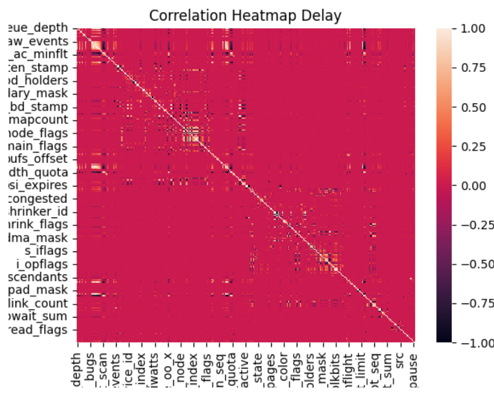

Figure A.2: Heatmap 300 bytes Delay Run1

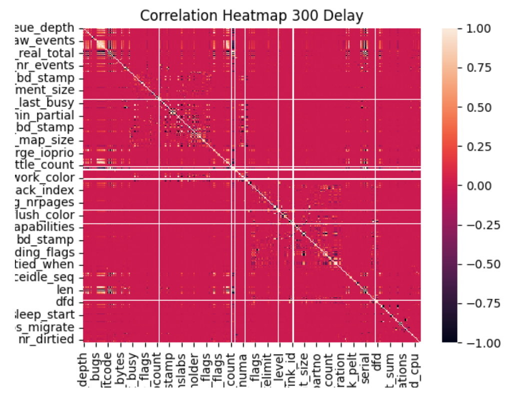

Figure A.3: Heatmap 300 bytes Delay Run2

A.2 Operation

The figures A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8, A.9, A.10, A.11, A.12 and, A.13 in this section are the heatmaps of the simulation runs. Run1 stands for the simulations with a standard deviation of 300 regarding the packet size and Run2 for the run with a 30 as the standard deviation regarding the packet size.

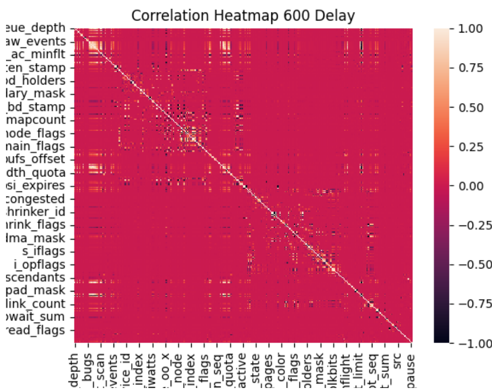

Figure A.4: Heatmap 600 bytes Delay Run1

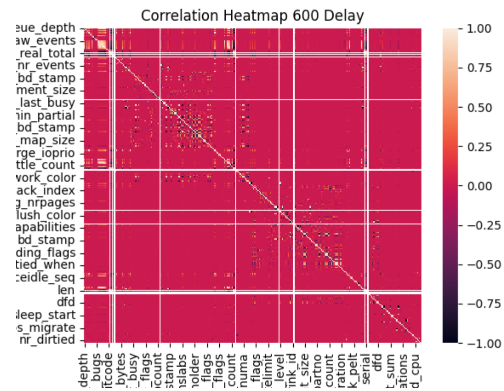

Figure A.5: Heatmap 600 bytes Delay Run2

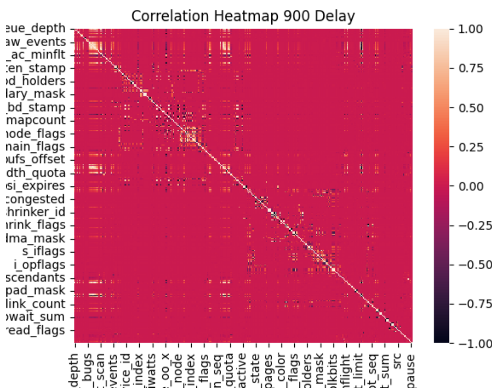

Figure A.6: Heatmap 900 bytes Delay Run1

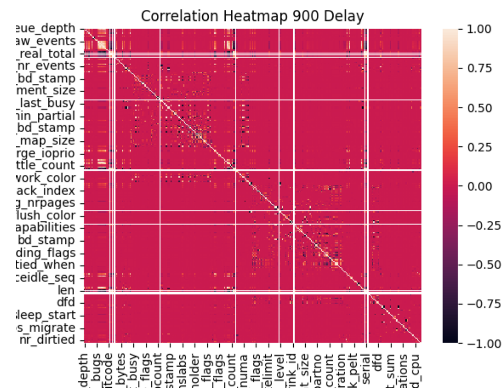

Figure A.7: Heatmap 900 bytes Delay Run2

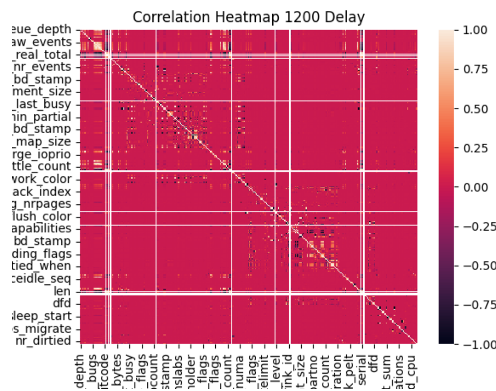

Figure A.8: Heatmap 1200 bytes Delay Run2

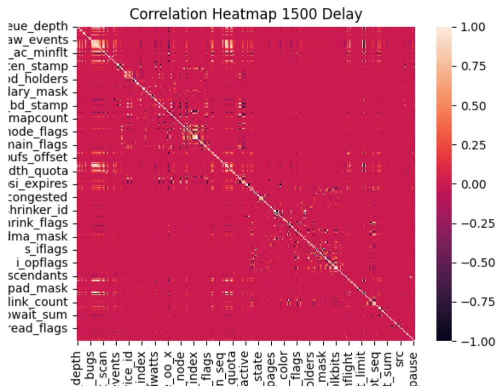

Figure A.9: Heatmap 1500 bytes Delay Run1

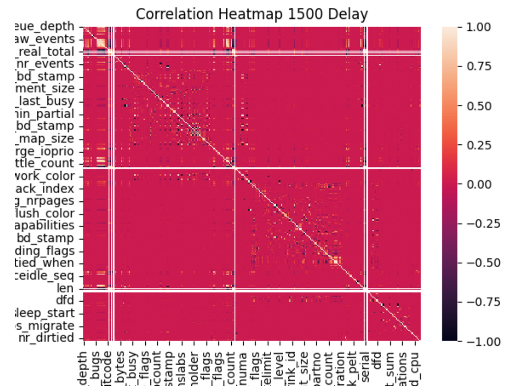

Figure A.10: Heatmap 1500 bytes Delay Run2

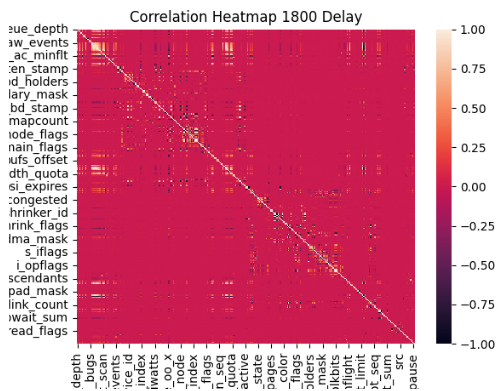

Figure A.11: Heatmap 1800 bytes Delay Run1

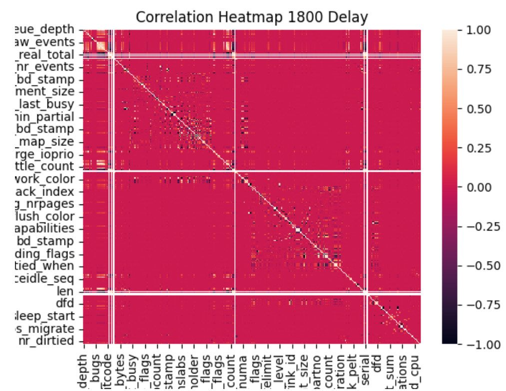

Figure A.12: Heatmap 1800 bytes Delay Run2

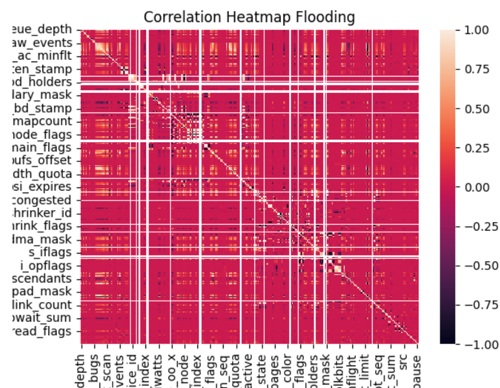

Figure A.13: Heatmap Flooding Run1