universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Concealed Honeypots: Developing a Tunneling System“

verfasst von / submitted by

Michael Mente, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2023 / Vienna, 2023

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 921
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Informatik UG2002
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Mag. Dr.techn. Edgar Weippl

Abstract

This work identifies and addresses approaches to developing a tunneling application for network traffic to and from multiple cloud VMs to a single honeypot device. This was accomplished by identifying the requirements for such a solution and examining the advantages and disadvantages of various technologies and approaches for effective implementation. The literature survey revealed that there are a variety of technologies and approaches that could be considered for implementing a tunneling solution. After a comprehensive evaluation of the technologies and approaches reviewed, we chose XDP as the base for our implementation. To evaluate the effectiveness of our tunneling application, we have identified the major requirements of scalability, throughput, and latency. Using a specially implemented test bed, we were able to simulate a realistic network scenario and evaluate a variety of network traffic types. The performance of our tunneling solution proved to be robust in all tested configurations and convincingly met the identified requirements. In our tests on a 1 Gbit link, our tunnel system consistently achieved a throughput of up to 317 Mbps, even under the challenging conditions of our virtual testbed, which does not offer hardware support for XDP. Despite these limitations, the introduced network latency remained remarkably low, consistently in the low microsecond range. This minimal processing latency in our test scenarios suggests that our application has the potential to scale with increasing complexity without adding significant latency. In the course of the evaluation, we also identified potential weaknesses in our implementation and in the approach using tunneling itself.

Our findings not only serve as a survey of the current state of tunneling technologies but also provide valuable insights for future research and development in this area.

Kurzfassung

In dieser Arbeit werden Ansätze zur Entwicklung einer Tunneling-Anwendung für den Netzwerkverkehr zu und von mehreren Cloud-VMs zu einem einzigen Honeypot-Gerät identifiziert und behandelt. Zu diesem Zweck wurden die Anforderungen an eine solche Lösung ermittelt und die Vor- und Nachteile verschiedener Technologien und Ansätze für eine effektive Implementierung untersucht. Die Literaturrecherche ergab, dass es eine Vielzahl von Technologien und Ansätzen gibt, die für die Implementierung einer Tunnellösung in Frage kommen. Nach einer Bewertung der untersuchten Technologien und Ansätze haben wir uns für XDP als die Basis für unsere Implementierung entschieden. Um anschließend die Effektivität unserer Tunneling-Anwendung zu bewerten, haben wir die wichtigsten Anforderungen an Skalierbarkeit, Durchsatz und Latenzzeit ermittelt. Mit einer speziell implementierten Testumgebung konnten wir ein realistisches Netzwerkszenario simulieren und eine Vielzahl von Netzwerk Szenarien testen. Die Leistung der Tunneling-Lösung erwies sich in allen getesteten Konfigurationen als robust und erfüllte die identifizierten Anforderungen überzeugend. In unseren Tests auf einer 1 Gbit Verbindung erreichte unser Tunnelsystem durchgängig einen Durchsatz von bis zu 317 Mbps, selbst unter den schwierigen Bedingungen unseres virtuellen Testbeds, das keine Hardwareunterstützung für XDP bietet. Trotz dieser Einschränkungen blieb die eingeführte Netzwerklatenz bemerkenswert niedrig und lag durchweg im niedrigen Mikrosekundenbereich. Diese minimale Verarbeitungslatenz in unseren Testszenarien deutet darauf hin, dass unsere Anwendung das Potenzial hat, bei steigender Komplexität zu skalieren, ohne dass nennenswerte Latenzen hinzugefügt werden. Im Zuge der Evaluierung haben wir auch mögliche Schwachstellen in unserem System und im Ansatz des Tunnelings selbst identifiziert. Unsere Ergebnisse dienen nicht nur als Überblick über den aktuellen Stand der Tunneling-Technologien, sondern liefern auch wertvolle Erkenntnisse für die zukünftige Forschung und Entwicklung in diesem Bereich.

Contents

List of Tables

List of Figures

List of Algorithms

1 Introduction

Advancing digitalization has led to a massive increase in Internet use and, as a result, a dramatic rise in cyberattacks. Cybercriminals use various technologies and methods to infiltrate, manipulate, and damage networks and systems. One way to detect and analyze such attacks is through using honeypots. Honeypots are special systems configured to look like real systems but are only used to identify attackers and analyze their behavior [34]. In particular, cloud-based networks and applications provided by public cloud providers are an attractive target for hackers due to their high availability and scalability [10].

While it is not always feasible to set up many physical honeypot devices for several reasons, such as cost or space, such a system provides the additional capability of analyzing which Internet Protocol (IP) ranges are relevant for attackers, as some attackers may only attack specific IP ranges. Consequently, honeypot operators are incentivized to use many different IP addresses simultaneously.

However, one challenge in using physical devices as honeypots is to securely and efficiently redirect traffic from multiple cloud Virtual Machines (VMs) to a single honeypot device, as depicted (cf. Figure 1.1).

This work addresses the main research question of how network traffic to and from multiple public cloud VMs can be securely redirected to a single honeypot device to detect and investigate attacks and threats. The goal is to develop a network system capable of collecting and processing traffic from multiple cloud VMs in a centralized location to facilitate the identification of potentially malicious traffic.

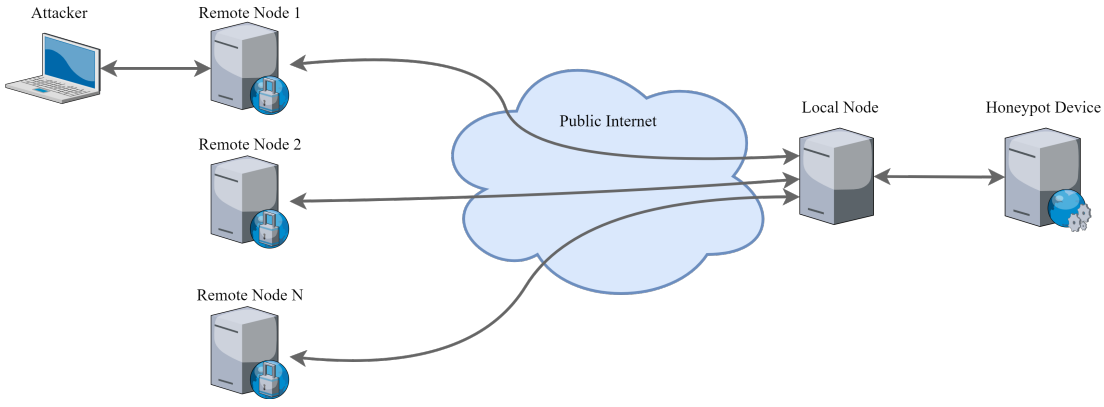

Figure 1.1: Application Architecture

The primary area of research will be on tunneling technologies and their use to redirect traffic from and to multiple cloud VMs to a honeypot device. In particular, the

implementation and configuration of network tunneling mechanisms and integration into existing networks will be addressed. Therefore, the technology evaluation also has to consider the aspect of efficient operation to minimize the required processing time. Cloud network security is a crucial research concern in this context. Furthermore, an objective is to ensure that the tunneling solution can function without raising any suspicion of potential attackers.

This work can improve network security in cloud environments by helping develop a tunneling system that facilitates threat detection. We provide an overview of the challenges, opportunities, and requirements in implementing such a system. We aim to provide a practical guide for implementing tunneling technologies in cloud environments by evaluating possible implementation approaches and presenting our own solution.

1.1 Objective

The objective of this master thesis is to develop a tunneling application that is able to effectively redirect network traffic from multiple remote entry points to and from honeypots. This is done by tunneling and forwarding traffic in such a way that the entry point behaves like a regular network component to potential attackers — i.e., the honeypot device itself. This is intended to hide the true honeypot from the attacker and make it much more difficult to detect or circumvent the tunneling system.

In addition, this thesis intends to perform a critical analysis of existing tunneling solutions, identify their limitations, and derive requirements for the application to be developed. This is done on the basis of a comprehensive literature review, which highlights both the current state of the art and future trends.

1.2 Relevancy

The relevance of this effort stems from several factors. First, given the increasing number and complexity of cyberattacks, stealthy honeypots can provide valuable information about attackers' tactics, techniques, and procedures (TTPs). Second, because traditional security measures are often insufficient to protect against advanced threats, honeypots can serve as a complementary defense strategy by proactively identifying and analyzing threats. Third, in an environment where attackers constantly adapt their strategies, detecting and defending against cyber threats requires innovative approaches such as the one pursued in this work.

1.3 Research questions

The presented significance of the topic underlines the importance of honeypot cloaking in the current cybersecurity landscape. In order to systematically address the complexity of the topic and define the scope of this thesis, it is essential to formulate precise Research Questions (RQs)s. These serve not only as a guide for the following scientific discussion

but also as a basis for evaluating the results obtained. Therefore, the central research questions will be discussed in the context of the tunneling application for honeypot concealment.

RQ1 What are the requirements for an effective tunneling solution for network traffic to and from multiple cloud VMs to a single honeypot device?

RQ2 Which technologies are best suited for an effective tunneling solution, and what are their advantages and disadvantages?

RQ3 How can the selected technology be implemented and evaluated to assess the effectiveness of the tunneling solution?

RQ4 What types of attacks and threats can be routed through tunneling solutions, and how effective are these solutions compared to other measures?

RQ5 How can the security of the tunneling software itself be ensured to prevent it from becoming the target of attacks, thereby compromising the integrity of the honeypot system?

RQ6 How can the results of this study help improve cybersecurity, and what other research topics and applications arise from the results?

1.4 Structure

In order to provide the reader with a clear orientation and a structured insight into the topic surrounding the creation of a tunneling application for camouflaging honeypots, this master's thesis is divided into several chapters that build on each other.

- The second chapter lays the foundation for understanding the following content. Here, basic terms and concepts that play a role in the context of this thesis are introduced and explained. It serves to ensure a uniform understanding of terms such as "honeypot" and "tunneling". In addition, a brief historical outline is given, highlighting the development of honeypots and the basic features of network tunneling.
- A comprehensive literature review is provided in the third chapter. This section presents existing research, approaches, and technologies in tunneling and honeypots. The structured analysis of the literature identifies both gaps and starting points for the author's own work. Additionally, this chapter deals with the requirements that can be derived from the problem definition and the findings of the literature review. Here, it is defined precisely which properties the tunneling application to be developed must fulfill and why these requirements are relevant.
- The fourth chapter translates the elaborated requirements into a concrete design. This section includes the selection of technology, the architecture of the system, and an examination of the technical challenges.

- The fifth chapter presents the actual implementation phase. Here, the steps, problems, and solution approaches in the development process of the tunneling system are described.
- The sixth chapter deals with the evaluation of the implemented solution. Different test methods are applied to check the functionality, performance, and security of the system and to compare the results with the requirements.
- In the seventh chapter, the main results of the work are critically reflected. Both the achieved goals and the limitations of the work are brought into focus. It provides space for in-depth discussion and places the results in the context of existing research.

  Additionally, we provide an outlook and outline potential further developments and optimizations of the system. Recommendations for future research in this area are also derived here.
- Finally, the eighth chapter rounds off the master thesis. It summarizes the key findings, reflects on the research process, and provides concluding thoughts on the significance of the work in the current research landscape.

2 Fundamentals

The field of network security and tunneling occupies a central place within the research field of information technology. It forms a critical gateway that is essential for both protecting digital infrastructures and ensuring operational integrity. With the rapid changes and continuous evolution of technological systems, network security research must also constantly adapt to effectively address current and future challenges. Particular attention is paid to the central concepts of this work, on the one hand, the investigation of the principle of tunneling network traffic and, on the other hand, the concept of honeypots as an integral element of security research. A thorough examination of the fundamental concepts of this field lays the foundation for further research and enables the development of innovative approaches to solutions in the context of modern Information Technology (IT) landscapes.

2.1 Definitions

In order to grasp the topic surrounding the tunneling application for honeypots in its full depth and to create a shared understanding for further explanations in this thesis, it is essential to deal with the central fundamentals and concepts. These fundamentals not only provide a solid starting point for the subsequent chapters but also ensure that all relevant terms and concepts are clearly and unambiguously defined. Before we delve deeper into the specific details of the tunneling application and its use in the context of honeypots, the central terms and concepts are first explained and contextualized below to create a common basis for discussion.

2.1.1 Tunneling

Tunneling, in network technology, refers to the practice of encapsulating or "tunneling" the data communications of one network protocol into the payload field of another protocol. This process allows data to be transported across incompatible network environments or between different network protocols.

In the core idea of tunneling, the main goal is to create a bridge or "tunnel" between two network points, treating the data to be tunneled in such a way that it is transparent to the transmitting network. This is often achieved by inserting the original data packets into the payload area of a new packet and then sending it over a transport network that may use an entirely different protocol or infrastructure.

An interesting feature of tunneling is its versatility with respect to OSI layers. The Open Systems Interconnection (OSI) model, which divides network protocols into seven

layers, provides different levels at which tunneling can be implemented. Depending on the chosen layer, different use cases and properties of tunneling can be realized:

- Data link layer (Layer 2): Tunneling at this layer is often used for bridging over networks that do not support native bridging protocols. Examples include the Layer 2 Tunneling Protocol (L2TP) and the Point-to-Point Tunneling Protocol (PPTP). These tunnels can transport Ethernet frames over IP networks.
- Network layer (Layer 3): This is where most Virtual Private Networks (VPNs) tunnels, such as Internet Protocol Security (IPsec) or Generic Routing Encapsulation (GRE), are implemented. They allow IP packets to be encapsulated within other IP packets.
- Transport layer (Layer 4): Tunneling at this layer can be used to create end-to-end connections that use specific transport protocols, such as TCP-based SSL/TLS protocol, for secure web connections.

Depending on the OSI layer, the benefits and use cases of tunneling can vary:

- Bypassing network boundaries: Tunneling can be used to bypass network restrictions or limitations, especially when certain services or protocols are blocked from a network.
- Security and privacy: In security-oriented applications, such as VPNs, tunnels are used to transmit data securely and privately over insecure networks.
- Protocol coexistence: Tunneling enables communication between different network protocols in heterogeneous network environments.
- Extension of network functions: At certain OSI layers, tunneling can be used to add or extend functions that are not natively available at that layer.

However, it is important to note that although tunneling offers significant benefits, it also comes with potential drawbacks. These include overhead from additional header information, potential delays from the additional processing, and error-handling challenges.

Overall, the versatility of tunneling in terms of OSI layers enables a wide range of applications and solutions that help address various challenges in network technology.

2.1.2 Honeypot

In today's digital landscape, characterized by an ever-expanding threat spectrum, honeypots have taken on special significance as specialized security threat detection and analysis systems. At their core, honeypots serve as bait for potential attackers by pretending to be legitimate and often vulnerable systems, while in actuality, they are designed to monitor and log every interaction.

The design of honeypots is based on the principle of deception. Rather than focusing on traditional security mechanisms such as firewalls or intrusion detection systems, honeypots

provide an alternative perspective by attracting attack attempts while collecting data on attackers' modus operandi and techniques. This passive approach to threat detection provides security professionals with valuable insight into current attack patterns and enables the development of proactive strategies to defend against future threats.

The diversity of honeypots can be divided into two main categories: Low-interaction and high-interaction honeypots. Low-interaction honeypots provide a superficial simulation of system services and are primarily designed to identify common attack patterns. In contrast, high-interaction honeypots provide a deeper and more comprehensive simulation by replicating complete operating system environments and applications, allowing for a more thorough investigation of attack techniques. [58, 38]

Although honeypots offer undeniable advantages in threat detection and analysis, they are not without potential pitfalls. For example, experienced cybercriminals could detect the presence of a honeypot and either bypass it or, worse, manipulate it by spreading misinformation or otherwise compromising the network.

Finally, amid the dynamism and complexity of modern IT infrastructures, honeypots continue to play an essential role, both in conventional networks and in more advanced environments such as cloud platforms. They provide unique opportunities to monitor, analyze, and respond to malicious activity, complementing other security measures and strategies.

Historical development of honeypots

The beginnings of honeypots can be traced back to the 1990s. The first attempts to apply deception in computer networks were quite rudimentary as they were often simple devices set up in networks to attract hackers. These early honeypots were usually nothing more than unprotected computers designed to serve as easy targets for attackers. Although using traps and deceptions as defense is not unusual outside the cybersecurity realm [12], the usage of honeypots was viewed with skepticism and was initially considered unethical [11].

In the early 2000s, the potential of honeypots was recognized in the commercial cybersecurity world. As interest grew, various solutions were developed and deployed, leading to the evolution of the concept. Security experts began to develop specialized honeypots for various purposes, which contributes to the various taxonomy schemes [38].

3 Systematic Literature Review

The research area of this thesis includes various aspects of computer science, especially network engineering and cybersecurity. Therefore, a comprehensive and systematic literature review is crucial to understand the current research state and identify gaps. Since some technologies used in this research area are still relatively new, searching a wide range of literature sources is essential to obtain a complete picture of the research. By conducting a systematic literature review, we can ensure that we recognize crucial contributions and put our understanding of the research area on a solid foundation. In addition, a systematic review allows us to critically evaluate and synthesize the existing literature to identify weaknesses in the research and raise new research questions. Consequently, the review was performed according to a defined process by Anders Kofod-Petersen. Kofod-Petersen identified the following key steps that make up this process that are relevant in the context of this thesis [27]:

1. Planning the review
    a) Specifying the research questions
    b) Developing a review protocol
    c) Evaluating the review protocol
2. Conducting the review
    a) Identification of research
    b) Selection of primary studies
    c) Study quality assessment
    d) Data extraction
    e) Data synthesis
3. Reporting the review
    a) Evaluating the report

3.0.1 Identification of Research

A robust and systematic methodology is essential for a successful literature review. With this in mind, several databases were selected for the present work to ensure the broadest possible coverage of all relevant publications. Both general and specialized libraries were considered to cover all relevant research areas and disciplines. The selected databases include, for example, Google Scholar, IEEE Xplore Digital Library, ACM Digital Library,

and ScienceDirect. Also, other databases were considered, such as CiteSeerX, Wiley Online Library, or SemanticScholar. However, those were discarded as they did not provide advanced search options or options to export the search results. Using these multiple sources and libraries aims to provide an accurate and comprehensive representation of the current state of knowledge on the diverse aspects of the research questions.

1. ACM Digital Library[1]
2. DBLP computer science library[2]
3. Google Scholar[3]
4. IEEE Explore[4]
5. ScienceDirect[5]
6. Scopus[6]
7. Springer Link[7]

After selecting the appropriate databases and identifying the relevant keywords and synonyms, as shown in (cf. Table 3.1), it is necessary to define the search strings. Here, the terms within each group are linked with boolean OR ($\vee$) operators, and the groups are linked with AND ($\wedge$) operators as such:

$$
\begin{aligned}
&([G1, T1] \vee [G1, T2] \vee [G1, T3] \vee [G1, T4]) \wedge \\
&([G2, T1] \vee [G2, T2] \vee [G2, T3] \vee [G2, T4]) \wedge \\
&([G3, T1] \vee [G3, T2] \vee [G3, T3]) \wedge \\
&([G4, T1] \vee [G4, T2])
\end{aligned}
$$

All possible variants of the relevant search terms can be covered through these combinations, as shown below. All relevant synonyms in each group must be considered to avoid missing any important results.

Before the filtering process can be applied, inclusion criteria must first be defined. These criteria determine which papers are relevant for the literature review and which should be excluded.

[1] `https://dl.acm.org/` (Accessed April 2023)
[2] `https://dblp.uni-trier.de/` (Accessed April 2023)
[3] `https://scholar.google.com/` (Accessed April 2023)
[4] `https://ieeexplore.ieee.org/Xplore/home.jsp` (Accessed April 2023)
[5] `https://www.sciencedirect.com/` (Accessed April 2023)
[6] `https://www.scopus.com/home.uri` (Accessed April 2023)
[7] `https://link.springer.com/` (Accessed April 2023)

	Group 1	Group 2	Group 3	Group 4
Term 1	eBPF	Tunneling	Honeypot	Network
Term 2	BPF	Forwarding	Security	Traffic
Term 3	XDP	Encapsulation	Cloud	
Term 4	Berkeley Packet Filter	routing		

Table 3.1: Search Terms

Inclusion Criteria

1. The Study is Open Access or accessible via the University of Vienna.
2. The Study is published after the year 2009.
3. The Study is published in English.
4. The Study is listed in one of the databases with the search-query.
5. The Study matches the keywords of the search string.

Quality Criteria

1. The Study is put into context with other studies.
2. The Study has a precisely formulated goal.
3. The Study's primary concern is related to one or more of the research questions.

3.0.2 Search Result

As mentioned, search strings using boolean operators are entered if the databases support advanced search mechanisms. Otherwise, the various combinations of the individual search strings were searched for directly.

This search method contributes to the high number of duplicates found in the libraries Springer Link, IEEE Explore, or DBLP, as shown in (cf. Table 3.2) . First, only search results published after 2009 and only papers written in English were considered. The right column "usable results" is the number of papers filtered that are Open Access or accessible via the University of Vienna. Afterward, the inclusion and quality assessment process can be applied to the remaining papers.

3.0.3 Selection of studies

In order to identify the relevant papers and eliminate unwanted results, the quality criteria are applied in a multi-stage filtering process. In this process, papers are first filtered based on title and abstract. In the next step, the full texts of the remaining papers are scanned for relevance, and finally, the remaining pieces are evaluated based on defined

Library	# Results	# unique Results	# usable Results
ACM Digital Library	282	154	154
DBLP computer science library	810	532	230
Google Scholar	428	236	202
IEEE Explore	201	32	32
ScienceDirect	362	238	95
Scopus	248	211	211
Springer Link	734	158	117
Total	3065	1561	1041

Table 3.2: Results of digital libraries

quality criteria. Through this process, the number of relevant articles can be reduced to a manageable number, which can then be used for the literature review.

After all relevant articles were retrieved from the databases, initial filtering was performed based on the title and abstract to exclude studies that did not meet the inclusion criteria. This process reduced the number of papers from 1041 to 98. Here, the reference management tool Zotero[8] was applied to check the inclusion criteria. Afterward, the inclusion criteria were extended to the full texts of the studies to achieve a further reduction to 54 papers. Subsequently, the relevance and validity of the topics covered were assessed in the context of the quality criteria and excluded. After this multi-stage filtering process, 23 articles remained suitable for further analysis, as shown in (cf. Figure 3.1). Finally, the relevant studies were supplemented by a forward and backward reference search to ensure all relevant articles were considered. The final dataset contains 33 studies.

3.1 Results

In the subsequent analysis, we differentiate between technologies for implementing such solutions and existing protocols and implementations that cover a similar use case. While the former comprises a range of technologies and frameworks that can be used to develop customized tunneling applications, the latter represents a collection of pre-built solutions and protocols that could be adapted for our use case.

This distinction allows considering each category's advantages and limitations concerning the tunneling application's specific requirements. It also helps to understand the extent to which existing protocols and implementations can contribute to achieving the goal and the extent to which the development of new, customized solutions is required. The results of this investigation are presented and discussed in the following section.

[8] `https://www.zotero.org/` (Accessed October 2023)

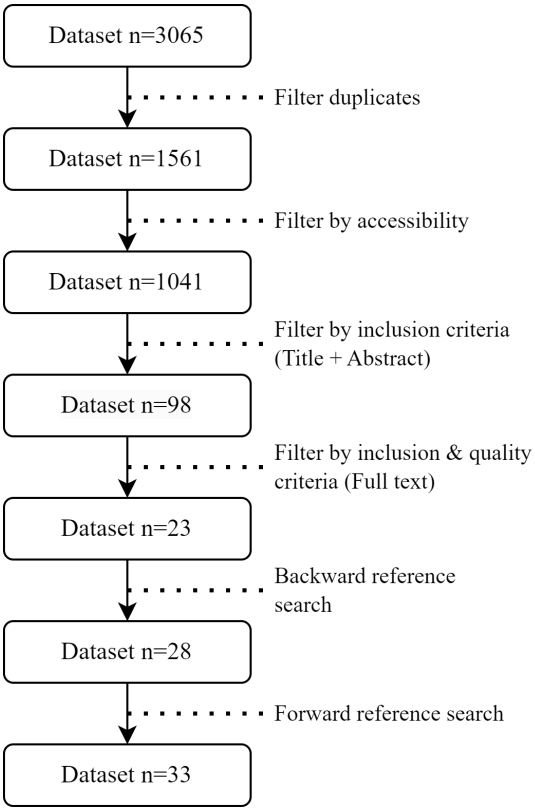

Figure 3.1: Study selection process

3.1.1 Technology Overview

During the preparation of the literature review, several technologies were identified that revealed themselves as a potential basis for the implementation of our project. Thorough analysis and evaluation of these technologies were critical to ensure that the chosen solution would not only meet the technical requirements but also current technology standards and best practices.

Of the technologies found, a few stood out in particular. In particular, these stand out because they offer a promising approach to implementation or are commonly used in a related field while being contemporary. It was critical to choose technologies that are modern and supported by an active developer and user community. This not only guarantees better performance and security but also enables easier integration, customization, and future extensibility. By highlighting their strengths, weaknesses, and areas of application, we can better understand why they are suitable for our specific application and how they can help achieve our project goals. It is critical to consider not only the technology itself but also the context of its development and application in order to make an informed decision about which technology is best suited for our purposes.

Programming Protocol-Independent Packet Processors (P4)

One technology that has been used more recently in software-defined networking is P4. P4 is a declarative, domain-specific programming language that allows for defining the processing process of network packets. P4 is an important tool in Software Defined Networking (SDN) and Network Function Virtualization (NFV) programming because it accelerates network processing by enabling developers to create highly customizable network behavior. P4 is a promising approach to meet the needs of tunneling applications as it allows protocols to be programmed and adapted flexibly since P4 itself is protocol agnostic. Some P4-based platforms, such as the Behavioral Model v2 (BMv2), allow custom protocols to be defined and network functions to be implemented [24, 43]. P4 also allows easy programming of packet header manipulation, packet filtering, packet forwarding, and traffic control, all essential aspects of tunneling applications. However, one potential challenge in using P4 for tunneling applications is the complexity of the proprietary P4 language, which may require a higher level of expertise. Overall, however, P4 is a promising technology well suited for tunneling applications because it allows protocols to be programmed and adapted flexibly and efficiently. [29, 31, 54, 60]

Data Plane Development Kit (DPDK)

Another prominent technology in the field of high-performance Software Defined Networking is the Data Plane Development Kit. Originally developed by Intel and now open-sourced, DPDK provides a collection of libraries and drivers for fast packet processing. The main appeal of DPDK is its ability to bypass the overhead traditionally associated with the kernel networking stack, enabling significantly higher packet processing rates. The framework supports a wide range of network applications, from software routers and switches to load balancing and security applications, as it allows developers to modify packets in a protocol-agnostic manner.

Additionally, it is compatible with many different platforms, so DPDK can be an attractive option for developing tunneling applications. It enables the processing of packets with very low latency and high data rates, exploiting parallelism using multiple CPU cores simultaneously to process data in parallel [32].

Another advantage of DPDK is its support for various network adapters and drivers. It provides custom drivers for a wide range of network adapters and offers a flexible API, allowing developers to write their own drivers or customize existing ones. [20]

However, DPDK's poll-mode driver is more computationally intensive than other drivers, such as interrupt-based or NAPI drivers. This means DPDK constantly queries the network card to receive and send packets from the network, which can lead to a higher constant CPU load [1, 6]. Therefore, when selecting DPDK as a technology for an application, it is essential to consider that it will impact system requirements and that sufficient CPU capacity is provided to ensure the correct execution of the tunneling application.

Although widely supported in general, there are limitations considering certain platforms, operating systems, or runtime environments. Containerization frameworks such as Docker

and Kubernetes, for example, might require additional configuration and optimization steps to achieve acceptable network handling capabilities [7]. Also, since DPDK binds the network interface in the userspace, it must be considered that the operating system cannot interact with the network interfaces used by DPDK. This introduces additional implementation overhead to correctly configure a system that works as intended, as all network traffic must be processed correctly. In addition, most network analysis tools do not work with DPDK-bound network interfaces, complicating debugging possibilities and complicating the implementation process [1, 6].

Nevertheless, DPDK provides a very powerful option for developing tunneling applications. However, it requires careful configuration and optimization, and its use can present some challenges depending on the platform and operating system.

Open vSwitch (OVS) + OpenFlow

Open vSwitch combined with OpenFlow represents another attractive technology option for implementing Software Defined Networking solutions.

Open vSwitch is a virtual switch that runs in the kernel and enables network virtualization. OpenFlow is a protocol used by OVS to route traffic through the switch [31]. OVS and OpenFlow aim to abstract network functions from physical hardware devices and implement them in software [17].

Therefore, it is possible to combine OVS and OpenFlow to create a tunneling technology. It enables the configuration of tunnels through centralized control and simplifies network management. Compared to other technologies, the configuration of OVS and OpenFlow is more straightforward and flexible as it works on a higher level, as OVS supports existing tunneling protocols such as Virtual Extensible LAN, GRE, and Generic Network Virtualization Encapsulation (Geneve), making it a versatile solution [59].

One drawback of OVS, however, is its performance. Compared to other technologies such as DPDK or eBPF, the processing speed of OVS is lower. This is due to the flow-based approach OpenFlow uses to identify and process each traffic based on predefined rules. These reasons lead to a comparatively computationally intensive process, which can harm performance [43].

Overall, combining OpenFlow and Open vSwitch provides a robust choice for the implementation of tunneling technologies when flexibility and ease of configuration are more critical than performance. It offers centralized control, broad support for existing tunneling protocols, and is a proven technology.

eBPF

The Extended Berkeley Packet Filter (eBPF) represents one of the most recent and promising developments in the field of SDN and systems monitoring. eBPF has attracted considerable attention in the networking community because it offers a revolutionary approach to network processing and monitoring.

Unlike DPDK and P4, the eBPF runtime is integrated into the Linux operating system kernel, allowing deeper integration with the operating system [5, 53]. eBPF builds

upon Berkeley Packet Filter, a technology already widely used for network analysis and monitoring [37]. eBPF extends the functionality of BPF and allows developers the novel approach to write kernel code without recompiling or modifying the kernel. The eBPF technology allows developers to perform specific actions on certain operating system events, such as system calls or network-specific events [7, 37], modifying the operating system behavior. This aspect grants potential performance gains over other technologies as eBPF code is executed in kernel space, allowing it to bypass parts of the Linux kernel entirely [1, 54].

The integration into the Linux kernel provides another advantage, namely broader compatibility. This, in turn, increases the developers' flexibility regarding the deployment scenario, as the tunneling application can be used in many different operating environments [46].

At the same time, the integration of eBPF into the Linux kernel also poses challenges, such as stringent security requirements and difficulties regarding typical debugging and validation processes [57]. Those security requirements are achieved by checking the bytecode before it is executed by a verifier, which ensures that the program does not perform unsafe operations and is guaranteed to terminate [5]. Also, the limited instruction count of 4096 of eBPF programs for specific Linux systems can challenge developers who want to create complex tunneling applications [46].

In summary, eBPF offers a highly promising technology choice for developing tunneling applications due to its deep integration into the kernel and high portability. However, due to the debugging challenges, it requires a deep understanding of security requirements as well as the architecture of the Linux kernel. Its flexibility, performance, and security make it a technology that is attracting much attention in both academic research and industrial applications, which also accounts for the rapid growth in popularity of this framework [18].

eXpress Data Path (XDP) Notably, eXpress Data Path represents a breakthrough technology in modern network engineering that stands out, especially in terms of packet processing on the network. The XDP technology is part of the eBPF framework and enables efficient packet processing directly at the network level.

XDP provides a new way to execute eBPF programs directly in the driver of a supported network interface before the kernel stack itself has the possibility to process the packet. This enables extremely fast and efficient packet processing, which is especially beneficial in high-throughput environments, leading to significantly reduced latency [20, 54]. The benefit of high-speed packet processing is accompanied by the disadvantage that XDP support is required at the driver level of the physical network interfaces. However, the XDP support of network cards is relatively limited. Nevertheless, even if the network card driver might not support the execution of XDP programs, the operating system provides the possibility to execute XDP programs prior to other operating system tasks, however without the performance advantage over regular eBPF networking [37]. This leads to the minor drawback that XDP requires a specific network card configuration to achieve optimal results [46]. However, XDP is considered a very effective way to perform packet

processing at the network level, which is affirmed by the adoption of this technology in very high-load scenarios such as load balancer implementations [14, 19, 40].

LibPcap

LibPcap represents an established and widely used technology that is particularly prominent in security-related applications. LibPcap is a library for capturing network traffic at the packet level. It is commonly used in network monitoring and security applications to analyze and monitor traffic. In particular, it plays a central role in security-related applications, providing the ability to dive deep into network traffic and gain critical insights into potential security threats. LibPcap does so by providing various functions for capturing, filtering, and analyzing network traffic [23, 48].

LibPcap allows developers to create applications that can read and filter traffic directly from network interfaces. This feature can be critical when developing a tunneling application for network traffic from cloud VMs to a honeypot device. Using LibPcap to intercept traffic, the application can identify targeted traffic forwarded to the honeypot device. In addition, LibPcap supports using BPF to apply efficient filters to the captured traffic, improving the application's performance [23, 56]. One limitation of LibPcap is that it can only passively monitor network traffic without any ability to interact [48].

While LibPcap is not a direct tunneling technology, it still provides valuable network traffic capture and analysis capabilities that can be used in a custom tunneling application, not only during deployment but also during development. When combined with appropriate tunneling protocols such as GRE, VXLAN, or other solutions, LibPcap can contribute to the development of an effective and efficient tunneling application that selectively routes network traffic from public cloud VMs to a honeypot device.

3.1.2 Existing approaches for tunneling applications

In addition to the technologies and tools for implementation already discussed, there are also a large number of existing technologies, applications, and protocols that have been developed specifically for tunneling and related applications. Integrating or adapting such existing solutions can often be an efficient and proven alternative to the development of completely new solution. In this overview, we have only covered a small selection of available technologies that seem particularly promising to us due to their relevance and potential. Numerous other solutions and approaches exist that could be considered depending on the specific use case and requirements.

Virtual Private Networks

Virtual Private Networks are an established approach to tunneling applications. This is done by creating an encrypted tunnel between the participating devices to ensure secure and private communication, although using public networks. VPNs are commonly used to refer to the complete implementation of such a tunneling application. Furthermore, a key feature of VPNs is to create virtual network spaces within a larger physical

network, e.g., to interconnect different parts of a single network without the need for a dedicated physical connection. To accommodate different usage requirements, VPNs can be implemented to transport different traffic types associated with the levels of the Open Systems Interconnection (OSI) model, typically network, transport, or application-level traffic [47].

Depending on the types of traffic required to be transmitted, existing protocols, such as the IPsec Protocol, Layer 2 Tunneling Protocol, or Point-to-Point Tunneling Protocol, can be used as a base protocol to build a VPN application [3]. However, it is also possible that VPN applications such as OpenVPN[9] or Wireguard[10] use custom application layer protocols based on standard transport layer protocols such as the User Datagram Protocol (UDP) or the Transmission Control Protocol (TCP). Therefore, it is possible to implement the specified tunneling application on top of VPNs [13].

Nevertheless, VPNs often represent ready-made solutions that offer limited possibilities for individual configurations and customizations in terms of how they work to adapt the behavior of the protocols used to specific use cases and requirements [3]. Although VPNs represent effective tunneling solutions, they are only partially suitable for the specific requirement of forwarding and tunneling network traffic from cloud VMs to a honeypot device. Considering these reasons, alternative or complementary tunneling methods and protocols should be considered to develop the best possible solution for the predefined scenario.

Layer 2 Tunneling Protocol (L2TP)

One way to implement tunneling applications is the usage of the Layer 2 Tunneling Protocol. This protocol was developed specifically to enable the transfer of data between remote users and private networks over the Internet. The Layer 2 Tunneling Protocol is a tunneling protocol initially developed by Cisco Systems and Microsoft to enable the creation of VPNs. L2TP operates at the application layer of the OSI model and is used to encapsulate layer two traffic. This enables combination with other standard protocols such as the IPsec to create a private and secure tunnel either between networks or individual devices [3, 47, 52].

L2TP is transmitted using the transport protocol UDP, resulting in the benefit of supporting usage in many different environments, such as public cloud VMs. However, compared to complete tunneling solutions such as Virtual private networks, L2TP does not have its own encryption functionality and, therefore, must be combined with other protocols to ensure data security and integrity, such as IPsec [3].

For the specific use case of tunneling network traffic from multiple cloud VMs to a single honeypot device, L2TP might not be well suited as it is mainly designed to aid the implementation of VPNs and may not provide the flexibility to enable many-to-one forwarding. In addition, combining L2TP with IPsec can increase implementation complexity and impact performance. Another aspect is that L2TP also encapsulates L2

[9] `https://openvpn.net/` (Accessed October 2023)
[10] `https://www.wireguard.com/` (Accessed October 2023)

traffic, which is not required in our specific use case.

Virtual Extensible LAN (VXLAN)

Another promising technology for implementing tunneling applications is the Virtual Extensible LAN protocol. The primary purpose of the VXLAN protocol is to extend existing Layer 2 networks over Layer 3 infrastructures to increase the scalability of existing networks such as data centers or cloud networks [33, 42]. As with L2TP, UDP is used as the underlying transport protocol, enabling the interconnecting of different network segments [48]. Therefore, it potentially offers advantages for implementing a tunneling application that routes network traffic from cloud VMs to a honeypot device. However, VXLAN is primarily designed to increase network scalability and not necessarily to direct traffic to a specific device such as a honeypot. VXLAN has the advantage of allowing flexible configuration. Regarding traffic encapsulation, VXLAN behaves similarly to L2TP in that, in both cases, L2 traffic is transmitted in UDP [59]. However, VXLAN was initially designed for a different purpose than L2TP.

Similar to L2TP, the transmission of L2 traffic is not a requirement for the application. Therefore, implementing a custom tunneling application with VXLAN requires a thorough analysis of the requirements of the specific use-case and a mindful technology configuration.

Generic Routing Encapsulation (GRE)

In addition to the tunneling technologies discussed so far, the Generic Routing Encapsulation protocol is also a potential option to implement our application. GRE was developed to transport a variety of network protocols over another, possibly heterogeneous network. It is particularly recognized for providing a simple and flexible way to implement a Layer 3 tunneling protocol designed to transport network traffic of multiple protocols over IP networks [42, 30]. To do so, GRE creates an IP tunnel that serves as a transport encapsulation protocol for other network protocols at the same layer of the OSI model. The original packets are encapsulated in this process, and GRE headers are added before they are transported over the IP network. GRE itself is protocol agnostic and can be used to encapsulate and transport both IP and non-IP traffic, which is attributed to its flexibility [3, 41, 59]. Due to its versatility, GRE is suitable for targeted tunneling of network traffic to a honeypot device [3, 30].

Therefore, it can serve as a foundation or as part of a custom tunneling application tailored to the specific requirements of the research questions. However, concerning the research questions, it must be considered that more data than necessary may be transmitted.

3.1.3 Comparison of technologies and approaches

In the next section, we will provide a detailed discussion and comparison of the previously selected technologies. For this comparison, we focused on the criteria of performance,

flexibility, and security. These factors were selected due to their critical importance for our specific application and the general requirements of modern network applications:

- Performance: This aspect is critical for network applications, especially if they are to be used in high-traffic environments. Here, not only the pure data transmission speed is considered, but also the ability of the technology to operate with minimal latency and how efficiently it works under load.
- Flexibility: This factor refers to the adaptability and expandability of the technology. On the one hand, we look at how easily the technology can be integrated into existing infrastructures and on the other hand, what possibilities it offers for future extensions or modifications to respond to changing requirements.
- Security: Given the ever-increasing threats in the area of cyber security, this aspect is of central importance, especially since our application directly interacts with potentially malicious traffic. Here, we analyze what security mechanisms are provided by the technology by default, what potential vulnerabilities exist, and how well they can be used in combination with other security solutions.

Performance

When evaluating the capabilities of these technologies and approaches in terms of performance, it is essential to consider several factors, including speed, resource consumption, and efficiency in processing network traffic. P4 and eBPF both offer high performance by providing fine-grained control over packet processing and forwarding. Additionally, eBPF also provides leveraging of hardware offload using XDP and enables faster traffic processing, especially for large volumes of data. In turn, the resource consumption of XDP is relatively low [1]. DPDK also offers very good performance in terms of network traffic processing speed accounted by its userspace driver system. However, the high resource demands of DPDK due to the poll mode drivers is a very important aspect to consider, especially in cloud environments where billing can be based on CPU time [6]. Open vSwitch and OpenFlow can also perform well by providing a programmable network control model, considering the requirements [43]. However, as with DPDK, low resource usage is preferred. The performance of custom-implemented software-based solutions based on the technologies just mentioned depends on the efficiency of the implementation and can be compromised when dealing with large amounts of data or high network loads. Ready-made VPN solutions can also provide adequate performance but are highly implementation dependent [13], especially considering the usage scenario. Using existing tunneling protocols such as L2TP, VXLAN, or GRE could generally provide sufficient performance since these have been developed explicitly for tunneling scenarios. However, this might depend on the specific implementation of the protocol. Therefore, testing these technologies prior to implementing such a forwarding scenario is necessary to make an accurate statement. The strong impression that emerges from the examination of different technologies is that optimal performance can be achieved with eBPF-based implementations. In particular, XDP stands out in this context because it was developed

specifically for high throughput and minimal latency in packet processing while at the same time being very resource-efficient in contrast to DPDK.

Flexibility

Flexibility is another crucial consideration in choosing technology for developing a tunneling application. In this context, flexibility comprises several aspects: compatibility with different network protocols, the possibility of adapting the underlying technology to the requirements, and flexibility regarding deployment options. Software-framework-based solutions such as P4 and eBPF offer very high flexibility [57]. The same applies in principle to solutions such as DPDK or XDP, although the flexibility is already somewhat reduced here because these have specific hardware or driver requirements [1, 6]. Generally, using low-level programming languages and APIs allows extensive customization and configuration to meet the specific requirements of our application. Open vSwitch and OpenFlow also offer a flexible solution for managing and controlling network traffic. Among existing tunneling approaches, GRE and L2TP offer high flexibility due to their ability to be combined with other protocols. In contrast, VPNs are less flexible because they are usually pre-built solutions that leave little room for customization regarding their operating mode.

In terms of flexibility, eBPF-based solutions once again appear exceptionally compelling. On the one hand, they offer remarkable freedom during the development phase since developers are hardly limited in terms of design and functionality. For another, the eBPF runtime is already integrated into many recent Linux distributions. This significantly reduces the requirements for the deployment scenario and simplifies the integration into existing infrastructures.

Security

In addition to the previous aspects, security poses a critical aspect when selecting a technology or approach for our tunneling application. Security depends on several factors, such as traffic encryption, authentication, and the ability to handle attacks and threats. Furthermore, an important aspect is that the tunneling application itself does not become a victim of security attacks by exploiting vulnerabilities in the implementation itself. P4, DPDK, Open vSwitch, and OpenFlow do not provide specific security mechanisms, but they can contribute to developing secure network structures and tunneling solutions, handing over the responsibility for a secure system to the application developers. Combining these technologies with additional security protocols or procedures is therefore critical. eBPF and XDP however offer a significantly higher level of security regarding implementation errors since extensive compiler checks are performed due to the in-kernel VM, which guarantees that the program cannot be exploited to harm the host operating system [51, 57]. L2TP and VXLAN however, do not provide any direct security features but allow security policies to be applied at a higher level of the OSI model. To mitigate these shortcomings, it is possible to use, for example, IPsec, as already mentioned, to ensure secure communication [3, 42]. Also, GRE has no integrated security features, but

due to its high flexibility, it is possible to use other protocols in addition to ensuring secure transmission of network traffic [42]. Ready-made virtual private network solutions provide inherent security through encryption and authentication, as the combination of different protocols is already available in one software package [13].

When relying on pre-existing implementations and protocols, there is an implicit reliance on them being implemented correctly and securely. Placing trust in such ready-made solutions carries the risk of missing potential vulnerabilities or flaws that may be present in the underlying implementation. It, therefore, requires a high degree of confidence in the quality and integrity of the underlying technology and its developers. While leveraging established and widely used protocols can speed up the development process and save resources, it requires careful evaluation in terms of security aspects and reliability.

3.1.4 Requirements for the tunneling application

Developing a sustainable application requires not only a sound understanding of the available technologies but also a deep awareness of the specific parameters and requirements that are critical to the success of the project. In the context of this work, three key performance parameters have been identified that are critical to the reliability and effectiveness of the solution being developed. These performance parameters - scalability, throughput, and latency - represent the core requirements such an application must meet to provide efficient and unobtrusive tunneling functionality.

Scalability

The requirement for scalability is vital for many-to-one tunneling, as it reflects the ability of a tunneling application to handle a growing number of machines efficiently. When forwarding network traffic from many public cloud VMs to a single honeypot device, it is paramount that the tunneling solution can easily scale with an increasing number of connected devices. Without scalability, a tunneling application could reach its limits as the number of devices increases, leading to bottlenecks, increased latency, and potentially reduced throughput. These reasons could impact the efficiency and effectiveness of the honeypot device and ultimately limit its ability to detect and analyze security threats effectively. A scalable tunneling application allows many machines to connect to a single honeypot instance efficiently and optimizes network and hardware resource usage. [10, 55]

Therefore, it is critical to consider scalability when developing a tunneling application for a given use case to ensure reliable and effective communication between many machines and a single honeypot device.

Throughput

Since many-to-one tunneling implies connecting an increasing number of devices to a single honeypot device, it is essential that the tunneling solution can handle increasing network traffic. One side of the tunnel must handle the additional traffic without degrading performance or overloading resources. It is important to note that the bandwidth

requirements for a single tunnel are typically not exceptionally high in the many-to-one tunneling application. The increased load occurs when considering the sum of many such connections. As the tunneling application connects numerous public cloud VMs to the honeypot device simultaneously, the cumulative bandwidth resulting from these connections can increase the load. In this scenario, the tunneling application must efficiently handle this load without impacting the systems' performance. [34]

By offering high throughput and efficient handling of accumulated network traffic, the tunneling application can maintain the necessary bandwidth requirements, allowing the honeypot device to monitor and analyze network traffic even as the number of connected devices increases.

Latency

Another critical requirement for a tunneling application for the many-to-one tunneling scenario is network latency, which refers to the transmission time of a single network packet from a network source to its destination. In the context of the tunneling application, latency refers to the time required to forward traffic from the VMs to the honeypot device. Since the system needs to make it appear that the honeypot device is in the cloud, the latency requirements in this use case are generally particularly stringent. After all, the distance that network traffic must travel is significantly greater than it should appear. Therefore, latency must remain small enough not to raise suspicion. Excessive latency could be perceived as an anomaly by potential attackers, alerting them to the presence of the tunneling infrastructure. [34, 55]

Therefore, it is necessary to develop a tunneling application that minimizes latency and thus ensures the effectiveness of the honeypot as an unobtrusive monitoring and analysis tool.

These requirements, identified by examining existing technologies, not only help us to define our application goals more clearly but also provide a basis for evaluating the suitability and efficiency of potential technologies in relation to our specific needs. It is therefore essential to make a technology selection that meets both current challenges and future requirements.

3.2 Discussion

Using the requirements identified so far for both the technology to be used and the application to be implemented, it is now possible, on the one hand, to make a recommendation regarding the choice of technology and, on the other hand, to answer the research questions already posed using the findings to date.

The comprehensive analysis of existing literature has also revealed that no comparable applications adequately address and fulfill the specific requirements and challenges of the proposed tunneling approach. Instead, it has been shown that existing research predominantly pursues the approach of bringing honeypot systems into the cloud and

virtualizing them instead of relaying network traffic, such as the approach demonstrated by Clemente et al. [10].

3.2.1 Suited technologies and approaches

Based on the application's specific requirements, such as scalability, throughput, and latency, P4 and eBPF – especially XDP-based approaches – pose as the most appropriate technologies and approaches for developing the tunneling application. Using P4 enables high flexibility and adaptability to process and control traffic efficiently. P4 allows custom network protocols and features to be developed that are well-suited to the specific needs of the tunneling application. Implementing a custom solution using P4 provides the ability to best tailor the tunneling application to the needs of the cloud infrastructure and honeypot traffic. Although DPDK can provide high performance and low latency in some cases, it is less suitable for this use case. The high resource consumption of DPDK and the associated poll mode drivers of the framework lead to lower efficiency of the tunneling application. These factors affect the successful implementation and integration of the solution into a cloud environment and make DPDK a less suitable choice compared to P4 and eBPF. Complementing this, eBPF offers high performance and flexibility in processing network traffic. Because eBPF runtime operates directly in the Linux kernel, network functions, and operations can be performed at a very low level, reducing overhead and latency whilst maintaining efficiency. This is particularly important for the tunneling application, as low latency is required to ensure effective communication between the cloud VMs and the honeypot device. In addition to the technologies and approaches discussed so far, it is essential to note that relying on existing protocols and applications for the tunneling application is possible but not optimal, especially concerning the latency requirement. Existing protocols and applications are usually not explicitly designed for the requirements of the tunneling application that is being evaluated, making them less flexible and adaptable. These reasons can result in higher latency than a custom solution that uses other technologies, such as P4 and eBPF. Both P4 and eBPF provide a robust and adaptable framework to create a solution that meets scalability, throughput, and latency requirements. Additionally, P4 and eBPF allow for the development of custom network protocols and features, while eBPF ensures potentially faster processing of network traffic with low latency. These technologies enable optimal tuning to the requirements of the tunneling application and integration into cloud infrastructures. Therefore, we can recommend P4 and eBPF as suitable technologies and approaches for developing the tunneling application as described.

3.2.2 Answering of Research Questions

Based on the results of our literature synthesis, we can now answer the Research Question one, two, and three.

RQ1 What are the requirements for an effective tunneling solution for network traffic to and from multiple cloud VMs to a single honeypot device?

An effective tunneling solution to forward network traffic to and from multiple cloud VMs to a single honeypot device includes scalability to allow communication between multiple entry nodes and a single exit node, throughput to handle increasing traffic due to the rising number of VMs, and latency to ensure effective communication without raising suspicion. These requirements must be met whilst maintaining minimal attack surface to ensure the reliable operation of the application.

RQ2 Which technologies are best suited for an effective tunneling solution, and what are their advantages and disadvantages?

The most suitable technologies for such a scenario are P4-, eBPF-, and especially XDP-based approaches. They offer significant advantages regarding flexibility, customization possibilities, and performance. Furthermore, they allow the development of custom network protocols, while in addition, eBPF offers deep integration into the Linux kernel and, thus, good performance. Finally, P4 and eBPF are well-suited technologies that match the specific requirements of the proposed tunneling application. The DPDK framework is another option that matches the requirements. However, it is less practical due to the higher resource consumption of the framework architecture, and it also increases the complexity of the implementation, as the network cards used by DPDK are not accessible to the operating system.

RQ3 How can the selected technology be implemented and evaluated to assess the effectiveness of the tunneling solution?

The effectiveness of the tunneling application can be evaluated based on several criteria. First, quantitative aspects can be measured directly based on the identified requirements. Throughput, latency, and scalability pose the most important performance benchmarks, all of which can be measured and evaluated objectively using standard methods. Additionally, qualitative aspects are also crucial in assessing the effectiveness of the application. These include the robustness of the application, i.e., how resilient and reliable the solution is in different operating conditions and possible error states. In addition, the quality of the implementation can be assessed by evaluating the effectiveness of the capabilities of the remote nodes acting as the honeypot system.

4 Conception and design

In this chapter, we synthesize theory and practice by selecting a suitable technology based on the findings of the literature review and designing the architecture of the application. This includes not only the specification of the algorithms for network packet processing on each part of the components of our application but also the central challenges that arise from the need for state information in our tunneling application. In addition, we explain how the application deals with unexpected network traffic scenarios and special cases in order to ensure robust and reliable operation.

4.1 Technology selection

In the debate over the optimal technology for implementing network tunnels, different approaches have their own strengths and weaknesses. Technologies such as P4, eBPF, and others all have their own characteristics that make them suitable for specific applications. However, this paper argues that XDP is superior to the other technologies, specifically for the intended tunneling application. This section highlights the reasons for this choice.

One of the main advantages of XDP is its ability to handle packets directly in the driver portion of the network stack. This means that the overhead typically associated with driving packets up through the entire stack is bypassed. This feature ensures not only exceptional packet processing speed but also greater efficiency, especially in high-throughput environments.

XDP is based on eBPF, which means it benefits from the flexibility and capabilities of this technology. With eBPF, customized programs can be created that are precisely tailored to the specific requirements of tunneling. This is particularly valuable when it comes to implementing specific behaviors or functions such as IP address rewriting.

The scalability of XDP is remarkable in that it is able to handle both small and very large network loads. This ensures that the tunneling application is able to handle a wide range of application and network scenarios without requiring any changes to the core implementation.

Because XDP is built on top of eBPF, it benefits from the security features built into eBPF. The eBPF system ensures that all programs are checked for potentially unsafe operations before they are executed. This reduces the risk of errors or security vulnerabilities that could arise in the tunneling application.

Considering the above arguments, XDP is clearly positioned as the preferred technology for implementing the tunneling application in this work. It offers a combination of performance, flexibility, integration, scalability, and security that cannot be found to the same extent in the alternative technologies.

4.2 Architecture

The conceptualized software architecture for honeypot disguise is based on a multi-part system, with each component performing specific and critical functions as shown in the already proposed application architecture in (cf. Figure 1.1):

4.2.1 Remote Node

The first of these components is the remote node. This acts as the primary point of contact for external entities, especially potential attackers. Any traffic initiated here is captured, encapsulated, and forwarded to the next entity, the Local Node, for further processing. Additionally, any traffic received from the Local Node will be decapsulated and sent to its destination. This is important as network traffic flows usually require two-way communication to work, as well as to provide any information about the honeypot entity. Therefore, our application also must be able to handle egressing traffic flows.

4.2.2 Local Node

The Local Node is strategically positioned in the honeypot operator's network and receives traffic transmitted from the Remote Node(s). After initial processing and decapsulation, this traffic is then forwarded to the actual honeypot device. This step is critical, as it is here that the data streams are analyzed and interpreted in a controlled environment before reaching their final destination.

When the honeypot generates a response - for example, a response to a request or simulated behavior - the traffic follows the same path in reverse order: from the honeypot device to the local node, from there to the remote node, and finally back to the original initiator, the potential attacker. It is critical to maintain certain information regarding inbound network traffic to correctly align any outbound network traffic to the correct remote node to ensure the proper operation of the application.

The clear separation and sequencing of these processes ensure not only the security and integrity of the data but also the obfuscation of the honeypot's actual location and purpose, which is fundamental to its effectiveness and credibility.

A sequential sequence of steps can now be derived from the basic application architecture described. These steps describe the foundation of the process and interactions between the various components, from the moment external traffic is initiated to the point at which a response is sent back to the initiator. This sequencing provides a clear structure and enables a deeper understanding of how the architectural components work and interact.

4.2.3 Inbound packets

This sequence of steps for inbound network packets, therefore, corresponds to the following defined sequence: First, they are received by the remote node, then forwarded through the tunnel from the entry point to the exit point (local node), and finally transmitted to

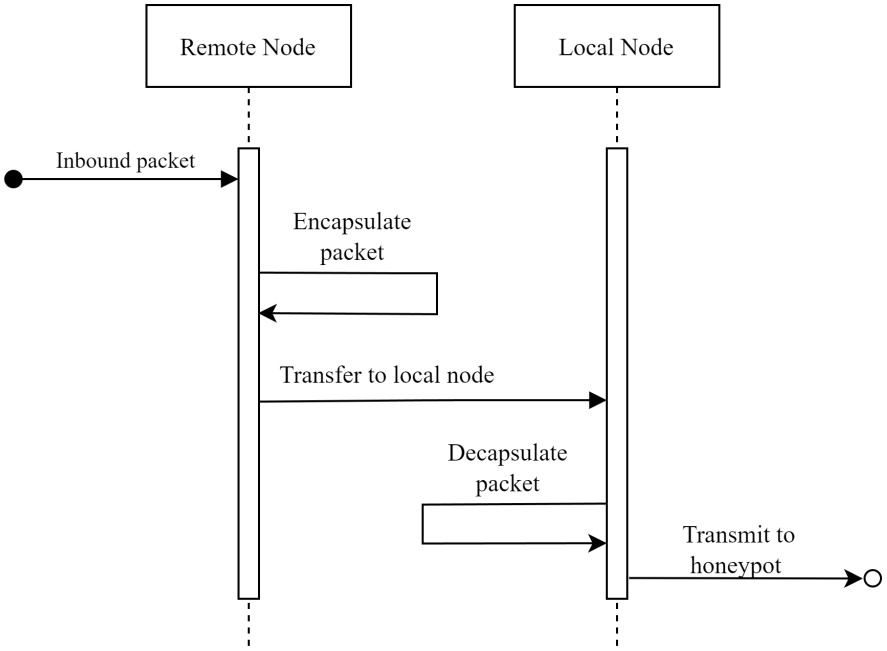

Figure 4.1: Inbound Packet Handling Procedure

the honeypot device for further processing. Each of these steps is critical to ensure the integrity and authenticity of the packet, as shown in (cf. Figure 4.1).

4.2.4 Outbound packets

Outbound packets go through the same flow as inbound packets, with the direction of network traffic exactly reversed: They are emitted at the honeypot device, are forwarded to the local node, pass through the tunnel to the remote node, and are finally returned to the original sender, or any other destination which might be addressed by the honeypot device (cf. Figure 4.2).

4.3 State information and its necessity

In network communication, especially in the implementation of tunneling applications, state information plays a crucial role. This section is dedicated to a deeper examination of the relevance of state information, its handling, and the resulting necessity for our tunneling application.

4.3.1 Definition of state information

State information, often referred to as "state," pertains to data that describes the current status or state of an application, connection, or process at a particular point in time.

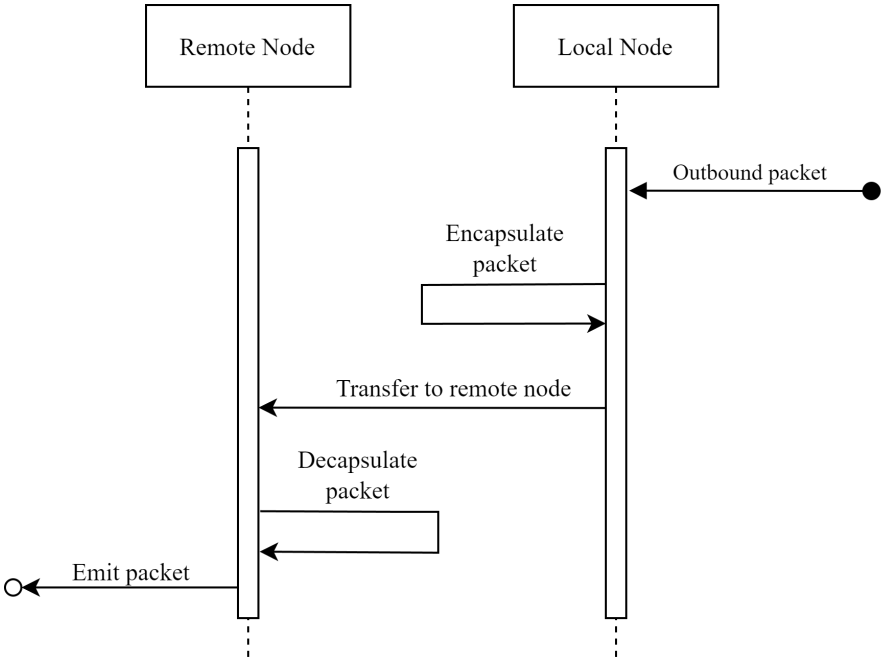

Figure 4.2: Outbound Packet Handling Procedure

In a network context, state information can include details such as the IP addresses of communication partners, ports in use, sequence numbers, timestamps, and other metric data.

4.3.2 Significance for tunneling applications

In tunneling applications, state information is particularly relevant because it ensures the integrity and continuity of data transmission. In particular, it can be leveraged for the following:

- Connection establishment and termination: State information can ensure that data streams are properly initiated, continued, and terminated.
- Packet correlation: They help associate individual packets with a particular connection or session, which is especially important when dealing with fragmented data or protocols maintaining sequencing.
- Security aspects: Storing state information can make it easier to detect anomalies or unusual activity and filter out potentially harmful traffic patterns.

4.3.3 State information in simple tunneling applications

In simple tunneling scenarios, especially when only a single remote node is involved, state information is not fundamentally required. Packets can be transmitted directly from

source to destination without the need for an intermediary or state management. Such systems can dispense with state information because they essentially serve as a simple relay function and do not have the complexity of multi-layered or multi-part systems. In such systems, not even tunneling itself would be required, as just rewriting of IP address information would be sufficient to redirect traffic to the actual honeypot device.

4.3.4 Challenges and necessity

Managing state information brings both benefits and challenges:

- Resource consumption: State information must be able to be stored and retrieved efficiently to avoid creating a data transmission bottleneck.
- Consistency: It is critical that state data is kept consistent, especially in the event of failures or network interruptions.
- Security concerns: The storage of state information, if not handled properly, can pose security risks as potentially sensitive information could be exposed.

Despite these challenges and the possibility of avoiding state information in specific scenarios, managing state information is essential for our tunneling application, especially as multiple remote nodes are involved, and hence, state management is indispensable. Understanding and correctly handling state information is, therefore, central to the successful design and implementation of a robust tunneling solution.

Minimizing State Information

Managing state information becomes essential once traffic is received from multiple remote nodes. One of the main reasons for this is the correct assignment of outbound traffic. When network packets enter the system from different entry points (remote nodes) and are forwarded via the same exit point (local node), there is a risk of these packets being mixed or mismatched. This can lead to a collision of data, faulty data transmission, and, ultimately, security problems.

State information here provides the ability to correctly track and mark inbound packets and ensure that they are routed correctly. For example, a packet received by Remote Node A could be given a specific identifier or state so that, when it is transmitted through the tunnel, a response packet can be correctly routed back to its original origin based on the incoming traffic. The same is true for packets received from other remote nodes. Without state information, it would be an immense challenge to manage and route these packets correctly, especially in a highly dynamic environment with variable network loads and constantly changing traffic sources.

A central goal in the design of the tunneling application was to keep the amount of state information as small as possible. This approach resulted from several considerations, both technical and security-related.

1. Resource efficiency: Managing a large amount of state information requires significant system resources, both in terms of memory requirements and processing power. Minimizing state data can reduce resource consumption, which is especially beneficial in resource-constrained environments or high-load scenarios.

2. Response time: A system that has to manage less state information can respond faster to requests. This improves performance and ensures faster data transfer through the tunnel, which is especially important for latency-sensitive applications.

3. Reducing complexity: Leaner state management simplifies application setup and maintenance. Error-proneness decreases because fewer components and data structures are involved, which could lead to inconsistencies or implementation issues.

4. Security aspects: Any stored information represents a potential target for attackers. By limiting the state information, the attack surfaces are reduced. In addition, the risk of data leakage or misuse is reduced because less sensitive information is stored.

5. Flexibility and scalability: Systems with minimized state management are generally more flexible and easier to scale. They can be more easily adapted to changing requirements or growing network topologies because there are fewer dependencies and data structures to consider.

In conclusion, reducing state information was a key consideration in the development of the tunneling application. The resulting benefits - from resource efficiency to performance and security - can contribute significantly to the robustness and efficiency of the final solution.

4.4 Tunneling Algorithm Design

The designed overall system shows similarities to conventional load-balancing systems, with a significant difference in the main direction of traffic. While in classical load balancing systems, the focus is on distributing inbound requests from clients to several servers to ensure an even load distribution, our system behaves inversely in this respect. Here, inbound traffic from various remote nodes is efficiently routed to a central point, the local node, and then forwarded to the destination honeypot. This inversion of the traditional load-balancing concept is necessary to meet the specific requirements of tunneling and disguising honeypots. This realization was essential in order to benefit from already proven concepts of load balancing.

When developing the tunneling algorithm, the initial focus was not primarily on the tunneling aspect itself. Rather, the focus was on solving a central problem: *How can network traffic from an arbitrary attacker be correctly assigned to the correct remote node to ensure that the data packets are returned to the original sender from the same remote node?* This question formed the core of the initial development phase, as its answer is essential for the functionality and efficiency of the entire system. Interestingly, the solution to this core question itself required some form of tunneling. Therefore, after

in-depth analysis and conceptualization, it was recognized that tunneling was not just a complementary feature but an integral part of the solution to this problem. This illustrates how closely the mechanisms of mapping and tunneling are linked in the context of this work.

4.4.1 Bottom Up Design

In developing the algorithm, we adopted a bottom-up approach. Instead of starting with an all-encompassing, highly complex system and then refining it step by step, we started with the most basic question, specifying the previous question: *How can outbound traffic from a honeypot device be accurately attributed to a specific attacker?* This approach took place under the assumption that transmitting inbound traffic to the honeypot device was already working properly. Thus, we wanted to ensure that the basic functionality was implemented reliably and efficiently before adding more layers and complexities.

The solution to this central issue is to retain the original source IP address, i.e., the attacker's IP address, of the network packet unchanged. The reason for this approach lies in the nature of the honeypot device itself: It acts like a black box and cannot be directly influenced or modified. Keeping the original source IP address ensures that traffic sent back from the honeypot can be correctly attributed to the original attacker without requiring any modifications on the honeypot itself.

This realization brought to light the need to encapsulate inbound traffic. By tunneling the traffic, we can ensure that the original information of the network packet, especially the original source IP address, is preserved.

However, this insight entails another important implication: For this approach to be successful, the exit node must be located in the same local network as the honeypot. Since the packet carries an " incorrect" source IP address - namely that of the original source - conventional IP-based routing mechanisms cannot be used within this network. Instead, we rely on Layer 2 addressing, or more specifically, MAC addresses. This means that traffic routing decisions must be made based on MAC addresses, which emphasizes the need for the exit node and the honeypot to be on the same local network.

4.4.2 Algorithm

The collected findings led to the development of specific algorithms (cf. Algorithms 1, 2, 3, 4), which are visualized in detail in the following section using figure (cf. Figure 4.3) for inbound traffic and for outbound traffic (cf. Figure 4.4) respectively.

It, in fact, requires four independent algorithms to make the system efficient. These each include specific instructions for inbound and outbound packets on both the local and remote nodes. Together, these algorithms form a coherent and integrated system that is critical to the efficient operation of the tunneling application.

Technical as well as economic efficiency was a central concern in the development of the algorithm for the remote node. By consistently reducing the algorithmic complexity, it was possible to minimize the computational effort. This reduction has not only technical

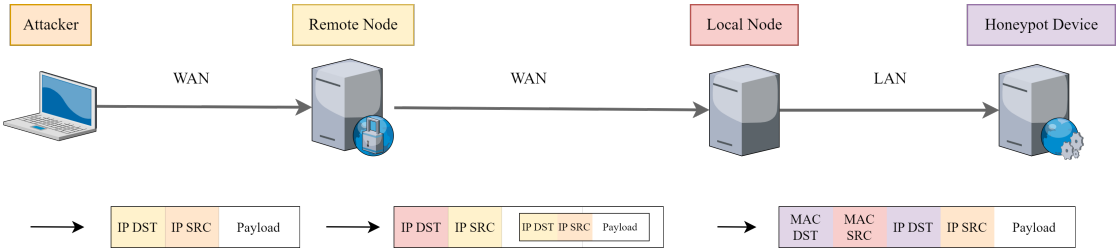

Figure 4.3: Inbound Packet Tunneling Mechanism

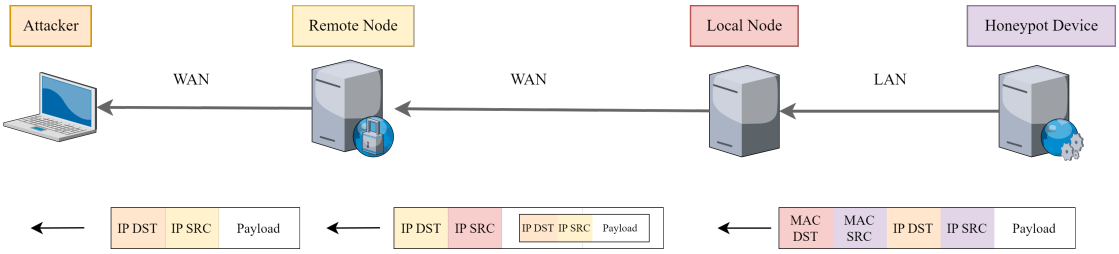

Figure 4.4: Outbound Packet Tunneling Mechanism

but especially economic implications. The limited computational complexity helps to reduce the costs possibly incurred when using the services of public cloud providers.

The decision to minimize algorithmic complexity on the remote node results in an increased computational load on the local node. By reducing the processing load on the remote node, the local node must be able to handle the additional computational load. This represents a deliberate trade-off: It has been prioritized to minimize complexity and, thus, computational overhead in an external environment while increasing responsibility and performance in its own controllable network environment. This approach ensures that the local node provides optimized performance while the remote node is less burdened.

Remote Node Algorithm

Algorithm 1: Remote Node Inbound Packet Address Handling Algorithm (Encapsulation)

```
initialization;
if packet is valid and packet.srcIP != localNodeIP then
    add space to network packet;
    write outer IP header;
    emit packet;
else
    drop packet;
end
```

Algorithm 2: Remote Node Outbound Packet Address Handling Algorithm (Decapsulation)

```
initialization;
if packet is valid and packet.srcIP == localNodeIP then
    strip outer IP header;
    emit packet;
else
    drop packet;
end
```

The algorithm on the remote node is designed to be highly efficient and simple. For inbound packets, the main operation of this algorithm is to add an additional outer IP header, wrapping the inner IP header to fulfill the tunneling functionality. This results in a minor extension of the original packet, leaving the original data packet unchanged (cf. Algorithm 1). On the contrary, for outbound packets, the previously added outer IP header is methodically removed. This returns the network packet to its original state before it is forwarded (cf. Algorithm 2).

Local Node Algorithm

Algorithm 3: Local Node Inbound Packet Address Handling Algorithm (Decapsulation)

```
initialization;
Map IPmap;
IpAddress lastRemoteNode;
if packet is valid and packet.srcIP != honeypotIP then
    strip outer IP header;
    if packet.srcIP not in IPmap then
        IPmap.store(key=packet.srcIP, value=packet.dstIP);
    end
    lastRemoteNode = packet.srcIP;
    emit packet;
else
    drop packet;
end
```

On the local node, the processing and control of network packets are significantly more complex than on the remote nodes. On the remote nodes, the primary function is to add or remove the outer IP header of a packet, a procedure that is essential for tunneling. This procedure is relatively straightforward and focuses mainly on adjusting the packet header.

Algorithm 4: Local Node Outbound Packet Address Handling Algorithm (Encapsulation)

```
1  initialization;
2  Map IPmap;
3  IpAddress lastRemoteNode;
4  if packet is valid and packet.srcIP == honeypotIP then
5  |  IpAddress srcIP;
6  |  if packet.dstIP in IPmap then
7  |  |  srcIP = IPmap.get(packet.dstIP);
8  |  else
9  |  |  if lastRemoteNode is set then
10 |  |  |  srcIP = lastRemoteNode;
11 |  |  else
12 |  |  |  drop packet;
13 |  |  |  return;
14 |  |  end
15 |  end
16 |  packet.srcIP = srcIP;
17 |  add space to network packet;
18 |  write outer IP header;
19 |  emit packet;
20 else
21 |  drop packet;
22 end
```

The local node, on the other hand, requires more advanced processing. In addition to the tasks also performed by the remote node, such as adding and removing the IP header, additional information must be maintained and managed here. In order to correctly assign outbound traffic to the associated remote node, it is essential to store state information. This state information, which serves as a kind of "memory" for the relationships between individual packets and their corresponding remote nodes, enables packets to be efficiently and correctly returned to their original senders.

Specific state information is generated for inbound traffic on the local node (cf. Algorithm 3). This is used to later correctly assign the outbound traffic to the original source, i.e., the corresponding remote node (cf. Algorithm 4). The said state information generated for inbound traffic essentially consists of the IP addresses of the attackers. These addresses are stored in a mapping scheme that contrasts them with the corresponding IP addresses of the remote nodes. This ensures a systematic overview of which attacker interacted with which remote node. This mapping scheme is essential to ensure that outbound traffic is correctly routed back to the initial source and is a central component in the flow of the developed algorithm. The systematic collection and processing of this state information is thus a key element in ensuring the integrity and efficiency of data communications.

This task, combined with the need to ensure the integrity and consistency of this state information, underscores the increased complexity and responsibility placed on the local node.

Handling Unknown Traffic A special case that was taken into account in the implementation concerns outbound traffic that cannot be clearly assigned to an attacker and, thus to a specific remote node. In such scenarios, instead of blocking or dropping the traffic, a predefined procedure is applied: Traffic is forwarded to the last remote node used (cf. Algorithm 4 line 10). This mechanism ensures that communication breakdowns are avoided and smooth interaction between the honeypot and potential attackers remains guaranteed, even if no clear assignment is temporarily possible.

If the outbound traffic is identified as unknown and no assignment to a previous remote node has been made so far, the traffic in question will be systematically discarded by the honeypot device (cf. Algorithm 4 line 12). It should be emphasized that such a situation typically occurs when the local node is restarted or its state is reset. This procedure serves not only to avoid potential inconsistencies in network traffic but also to minimize the risk of mismatches. Targeted discarding of traffic in these specific scenarios is a strategic measure to maintain the integrity and operational reliability of the system.

The special case described is not optimal, but it is currently the only viable solution to deal with the special challenge of outbound traffic assignment. When multiple attackers interact with the system at the same time, a situation can arise in which the last remote node used is not the one actually intended. Such a mismatch has the risk of allowing attackers to guess the existence and operation of the system. Although this could compromise the goal of covert monitoring and deception, the current approach is the only method available to minimize the challenges that arise and keep the system

functional. Similarly, in practice, it is necessary to observe and evaluate whether this theoretical shortcoming actually has a significant impact on the effectiveness and reliability of the system. While theoretical considerations can provide important clues to potential weaknesses, only practical application and continuous observation will reveal the extent to which this implied shortcoming actually leads to operational challenges. It is, therefore, essential to conduct constant monitoring and analysis of the system in real-world use to gain such insights and adjust the system architecture as necessary.

5 Implementation

The implementation phase represents a critical transition point from the conceptual to the functional stage of a project. In this chapter, the implementation of the architectures and algorithms previously defined in the conceptual and design phases is discussed in detail. We explain the types of network traffic that the tunneling application processes and how these decisions were justified based on measurement results taken by representative internet traffic captures. An essential part of this section describes our approach to creating a testbed for the application, including the choice of technologies and architectures used. We delve deeper into the details of the implementation using eBPF and XDP, highlighting both the challenges, such as the recomputation of checksums, and potential limitations that emerged during the implementation process. We also describe the iterative process of our implementation in order to justify our approach and the resulting decisions fully. This is to ensure not only the functionality but also the traceability and reproducibility of the developed solution.

5.1 Scope of Traffic to be considered

During the implementation phase, a specific focus was deliberately placed on IPv4 traffic. This is a decision based on the current dominance and relevance of IPv4 on the Internet. Within this IPv4 framework, a targeted differentiation was further carried out: Only traffic based on the TCP, UDP, and ICMP protocols will be considered and forwarded. This decision was made because these protocols represent the predominant means of communication in IPv4 traffic and thus have the greatest relevance in the application environment under consideration. Other, less common protocols were deliberately excluded in this implementation phase to ensure optimized and efficient processing. However, it should be noted that this restriction also introduces potential limitations in certain application scenarios and may need to be re-evaluated in future revisions or extensions of the system.

5.1.1 Reasoning for chosen traffic types

According to the traffic capture[1] of the passive sampler performed by the renowned CAIDA organization (Center for Applied Internet Data Analysis), the said traffic, consisting of the TCP, UDP, and ICMP protocols within the IPv4 framework, actually represents a significant part of the total Internet traffic (cf. Table 5.1). In the dataset under consideration, the share of IPv6 traffic, however, represents a marginal quantity (cf. Table

[1] equinix-nyc.dirA.20180517-125910.UTC.anon.pcap

Transport Layer Protocol	Packets Total	Packets Fraction
TCP	22285317	83.782%
UDP	3893262	14.637%
ICMP	143352	0.539%
Other	277172	1.042%
Total	26599103	100%

Table 5.1: IPv4 Traffic Transport Layer Protocol Composition

Internet Protocol Version	Packets Total	Packets Fraction
IPv4	26599103	98.465%
IPv6	414665	1.535%
Total	27013768	100%

Table 5.2: Internet Protocol composition of WAN Trace captured in 2018

5.2). This dataset, which is used to calculate the mentioned values, represents a traffic sample of 49.52 seconds. This sample was captured on May 17, 2018, at an Equinix switching center in New York City. The acquisition time was at 12:59 UTC, corresponding to 08:59 local time.

This further substantiates the decision to focus on these protocols during the implementation phase. Focusing on these specific protocols not only enables optimized processing but also ensures that the majority of Internet traffic that actually occurs is captured and handled.

The data set on which the analysis of the individual protocols is based is from 2018. Despite the time difference, one might have concerns about the validity and relevance of such data. Nevertheless, according to the CAIDA organization, the ratio of IPv4 to IPv6 remains stable over time, as shown in the table (cf. Table 5.3 [2]). This suggests that despite the age of the collected data, the trends and ratios reflected in it remain valid. In this respect, this data, even if it does not represent the latest status, can be seen as a

[2]The data of table (cf. Table 5.3) can be publicly retrieved at the website of the CAIDA organization [8]. The data is based on a one-hour traffic sample captured on 17th January 2019 at 12:59 UTC at the same data center, therefore resulting in significantly higher total packet count values compared to the values presented in the previous table (cf. Table 5.2). The comparison of the data sets is particularly valid due to the fact that they were recorded at the same time at the same data center; only the recording date differs.

Internet Protocol Version	Packets Total	Packets Fraction
IPv4	2339350262	98.856%
IPv6	27069656	1.144%
Total	2366419918	100%

Table 5.3: Internet Protocol composition in 2019

solid basis on which to formulate justified assumptions for the present project.

Flexible protocol adaption To future-proof, our application implements a whitelist-type approach that only allows predefined transport layer protocols to be encapsulated and transported. This is not only for clarity and easier maintenance of the system but also for security, as an extension of protocol support requires a corresponding adaptation and review of the security measures of the network the tunneling application is deployed in. Although our system would technically be able to encapsulate any transport layer protocol and thus enable fully protocol-agnostic behavior, we prioritize limited and controlled protocol support by handling only the most common transport layer protocols, TCP, UDP, and ICMP. However, if a specific use case or exploit requires other protocols to be tunneled, this is comparatively straightforward to implement.

In this context, limiting the focus to these protocols provides a reasonable balance between coverage of real traffic and efficiency in processing. It confirms the relevance of the chosen scope of the implementation and underlines its practicality in a majority of real-world scenarios.

5.2 Testbed

In the research and development of software applications, the preparation phase before actual implementation is an essential part of the process. However, before the actual implementation can begin, an adequate test infrastructure must be established. This so-called "test environment" serves as a critical framework within which the software application is evaluated under controlled conditions.

This test environment - often characterized as a "test bed" - has the task of replicating a realistic scenario of the future operating conditions of the application. In doing so, it should not only simulate the overall complexity of the final system but also provide developers with specific tools to monitor and analyze the performance, stability, and interoperability of the software.

The need for such an environment stems from the multi-tier nature of the system being developed. It is, therefore, imperative that the test bed be able to emulate the dynamics and interactions between the individual components accurately. The possibility of continuous evaluation during the development phase allows anomalies and inconsistencies to be identified and addressed at an early stage.

In summary, it can be stated that the test bed is not just a supplementary component but rather a central element in the development process. It ensures that the resulting software application, once it is transferred to productive operation, meets the defined scientific and technical standards.

5.2.1 Testbed Requirements

The elaboration of an effective test bed is a central prerequisite for the successful development process of the software application. The specific requirements placed on this test

bed are diverse and must be carefully considered. The main criteria are listed below:

1. Realistic network topology: The test bed should represent a realistic network topology, which corresponds in its structure and dynamics to the intended usage scenario in productive operation. This ensures that the software is tested under conditions that approximate its subsequent operating environment. The requirement for a realistic network topology requires a more detailed specification in order to meet the requirements of a real deployment scenario. In detail, this topology should be designed as follows:

   a) Remote nodes: At least two remote nodes should be part of the test bed. As these nodes will be placed on the public Internet, the actual testbed must mimic interaction conditions between the nodes and the potential attacker.

   b) Local Node: In addition to the remote nodes, there should also be a local node in the test bed. While it can be addressable on the public Internet, it is critical that it also has a connection to the local network and is in direct contact with the target system.

   c) Honeypot system: A honeypot system must also be integrated within the test bed to replicate interactions with attackers realistically. The honeypot system is the actual decoy system that attracts attack attempts and processes them in a controlled manner. It should be designed to mimic the typical characteristics and behaviors of the actual systems being protected. It is essential that the honeypot system is positioned in the local network and operates in the immediate vicinity of the local node in order to simulate the entire communication chain authentically.

   d) Attacker simulation: For an authentic test environment, it is also essential to include a simulated attacker that is also positioned on the simulated "public" Internet. This ensures that the threat vectors and the system's responses to such threats are realistically mapped.

   This specified layout of the network topology ensures that all relevant actors and network components are represented. It is intended to enable realistic simulation and testing of the software in an environment that is as close as possible to actual operating conditions.

2. Simplification of hardware requirements: While it is beneficial to have a realistic test environment, it should not be too complex or resource-intensive to the host system running the testbed. It is essential to find a balance between realism and practicality so that the hardware requirements are not prohibitively high.

3. Flexibility and adaptability: A key feature of the test bed should be its ability to be flexible. Development is often a dynamic process, and the test bed must be able to adapt to changing requirements. This includes the ability to compile code directly in the test environment to allow rapid iterations and adaptations.

4. Dynamic adaptability: Since software development is an iterative process, the test bed must be able to adapt quickly to changing requirements or new test conditions. This is especially important to drive the development process efficiently and continuously.

5. Reproducibility: A decisive criterion for the test bed is also its reproducibility in order to meet scientific standards. It is essential that experiments and tests can be repeated under the same conditions to verify results, ensure the credibility of research, and avoid the discrepancies that can occur with repeated installations of the test environment. This requirement is supplemented by the requirement of portability of the test environment to increase reproducibility further.

6. Monitoring and analysis tools: An efficient test bed should have the possibility for interoperability with tools for monitoring and analysis. This enables developers to conduct systematic tests, collect performance data, and detect any weaknesses or malfunctions early on.

In summary, the test bed should not only provide a reproducible and realistic environment, but it must also have the necessary flexibility and dynamics to meet the constantly changing requirements during the development process. It serves as a bridge between theoretical planning and practical implementation, thus ensuring the quality and reliability of the final software solution.

5.2.2 Testbed Implementation

The realization of such a test bed can be done via different technological approaches. Each of these technologies brings specific advantages and challenges. The technology approaches being considered are listed subsequently:

1. Physical hardware infrastructure: This approach involves using real devices and network connections to create a physical environment that closely replicates the real production network.

2. Virtual Machines: VMs allow multiple isolated systems to run on a single physical machine. Tools such as VMware[3] or VirtualBox[4] enable rapid setup and management of virtual network environments.

3. Containerization: Technologies such as Docker[5] or Kubernetes[6] enable the creation of isolated application instances in containers. These are more lightweight than VMs and offer speed and flexibility in setup and scaling.

[3] `https://www.vmware.com/` (Accessed October 2023)
[4] `https://www.virtualbox.org/` (Accessed October 2023)
[5] `https://www.docker.com/` (Accessed October 2023)
[6] `https://kubernetes.io/` (Accessed October 2023)

4. Network simulation tools: Software such as Network Simulator 3 (NS3)[7] or Mininet[8] enables simulation of complex network topologies and testing of network behavior in a controlled environment.

5. Cloud-based solutions: Common public cloud platforms such as AWS[9], Microsoft Azure[10], or Google Cloud[11] provide the ability to create and scale virtual infrastructures as needed. They also offer a variety of services that make it easy to set up and manage network test environments.

6. Hybrid approaches: A combination of physical hardware, Virtual Machines, containers, and/or cloud resources can be used to create a comprehensive and flexible test bed that meets the specific needs of the project.

The implementation of a suitable test bed can be realized by many different technology approaches. The choice of the right approach depends on a number of different factors. For practical reasons, however, only virtual solutions were initially considered. This decision was based on factors such as implementation speed, cost, maintainability, and the ability to quickly and efficiently adapt the test bed to changing requirements. Despite the focus on virtual solutions, it should not be ignored that other alternative approaches could also be considered in later project phases or for specific requirements.

Testbed Implemented in Docker

The initial approach to implementing the testbed was based on Docker, a widely used platform for containerization. However, despite its popularity and versatility, it turned out that Docker was not ideally suited for the specific requirements of this scenario. Here are the main reasons identified considering the previously mentioned requirements for the testbed:

1. Network topology: Creating a realistic and complex network topology within Docker can be challenging. Docker uses a simple network bridge by default, which is sufficient for many applications but is not ideal for simulating a realistic network scenario with multiple nodes and specific routing requirements, as the network drivers provided by Docker by default do not allow the creation of complex network topologies especially as multiple containers would be involved, which in turn requires the usage of orchestration tools adding another layer of complexity.

2. Flexibility in development: While Docker offers the ability to run code in containers, there can be limitations in dynamically adapting the test bed during development. This is because Docker containers are typically configured for a specific set of tasks and do not necessarily provide the flexibility required. This highlights the primary

[7] `https://www.nsnam.org/` (Accessed October 2023)
[8] `https://mininet.org/` (Accessed October 2023)
[9] `https://aws.amazon.com/` (Accessed October 2023)
[10] `https://azure.microsoft.com/` (Accessed October 2023)
[11] `https://cloud.google.com/` (Accessed October 2023)

shortcoming of using Docker as the testbed during the implementation stage, where frequent changes of the implementation would require a re-compilation of the entire container environment, which slows down the development process significantly.

3. Hardware emulation limitations: Docker is primarily intended for software containerization and not for full emulation of hardware environments. Thus, there may be limitations in accurately emulating certain hardware requirements. During the development of the testbed, it was not entirely clear if special driver requirements were met to interoperate with the requirements of the chosen implementation technology, XDP.

In summary, although Docker offers many advantages in software development, the specific requirements, especially the requirement of using the testbed as the development environment, of this project were not fully met. This led to the need to investigate alternative approaches for implementing the test bed.

Using VirtualBox to configure Testbed

Given the identified drawbacks of Docker for the specific requirements of the project, it became clear that the use of Virtual Machines would be better suited to address them. VMs offer greater flexibility and control in many aspects, particularly in terms of network topology and realistic emulation of hardware and network environments. Therefore, the next approach considered was to configure the test bed using VirtualBox. VirtualBox, a widely used tool for virtualization, allows for detailed configuration of network interfaces and accurate emulation of hardware components and also provides the ability to run multiple networked VMs on a single physical host. This makes it possible to create a realistic yet controllable network environment that meets the specific requirements of the test bed. However, using VirtualBox to configure the test bed came with a significant drawback: the requirement of reproducibility was not guaranteed. While VirtualBox is capable of creating a detailed and realistic network environment, the process of setup and configuration is highly manual. Each VM instance, network, and associated setting must be configured manually. This means that transferring the setup to another environment or setting it up again after a system failure is not trivial, as the process depends on many variable human actions that could lead to discrepancies in repeated deployments.

Therefore, despite the advantages of VirtualBox, its use was not optimal for the desired test bed.

Testbed Implemented in Vagrant

The use of Vagrant[12] addresses exactly this shortcoming of the previous approach. Vagrant provides a solution focused on the creation and management of virtualized development environments, with a primary focus on automation and reproducibility. Instead of manually creating and configuring a virtual machine, as in VirtualBox, Vagrant allows

[12] `https://www.vagrantup.com/` (Accessed October 2023)

users to create a systematic and standardized definition of the desired environment in so-called Vagrantfiles. These declarative files specify various aspects of the virtual machine, including the underlying operating system, hardware configuration, and preinstalled software. Several benefits are realized through the use of these files considering prior requirements:

1. Reproducibility: Every time a Vagrant up command is executed on such a Vagrantfile, exactly the same environment is created. This means that regardless of the underlying physical machine - be it a developer laptop, a lab server, or a cloud instance - the result is consistent.

2. Portability: Vagrantfiles are text-based in nature and can, therefore, be easily stored in version control systems and shared between developers or researchers. This facilitates collaboration and ensures that all stakeholders are working in identical environments. Furthermore, portability is increased as the Vagrant system in itself does not specify the underlying virtualization technology, reducing the specific software requirements of the whole testbed.

3. Transparency: Explicitly defining all aspects of the environment in a Vagrantfile creates transparency. Anyone viewing the file can understand exactly how the environment is configured and what dependencies exist.

4. Automation: As Vagrant environments are automatically created from their definitions, human error, which often occurs in manual configuration processes, is reduced.

Given the scientific requirements, particularly the need to conduct experiments in a controlled, consistent, and traceable environment, Vagrant provides a compelling solution for implementing the test bed. Despite the additional layer of complexity introduced by integrating Vagrant into the development process, the benefits of this approach clearly outweigh the drawbacks. The systematicity and consistency provided by Vagrant in dealing with virtual environments enable developers and researchers to ensure that the test environments generated are free of configuration anomalies or unexpected dependencies. This guarantee of a standardized and reproducible environment not only reduces potential sources of errors but also facilitates debugging processes. Thus, despite the added complexity, Vagrant represents a strategic choice that significantly increases the quality and reliability of the development and research process.

Testbed Architecture In the context of implementing the test bed, we decided to use *Ubuntu*[13] *22.04.2 LTS* as the base operating system for all components. There were several reasons for this decision. For one, Ubuntu provides a wide selection of available software packages[14] and, therefore, is often preferred for research and development, ensuring

[13] `https://ubuntu.com/` (Accessed October 2023)
[14] `https://packages.ubuntu.com/` (Accessed October 2023)

consistency and comparability of work. For another, Ubuntu enjoys widespread adoption in cloud environments. The dominance of Ubuntu in such scenarios provides some assurance in terms of stability, security, and support for required packages. Furthermore, the choice of such a widely used system promotes the transferability of the developed approach to real and industrial scenarios, increasing the relevance and applicability of the research.

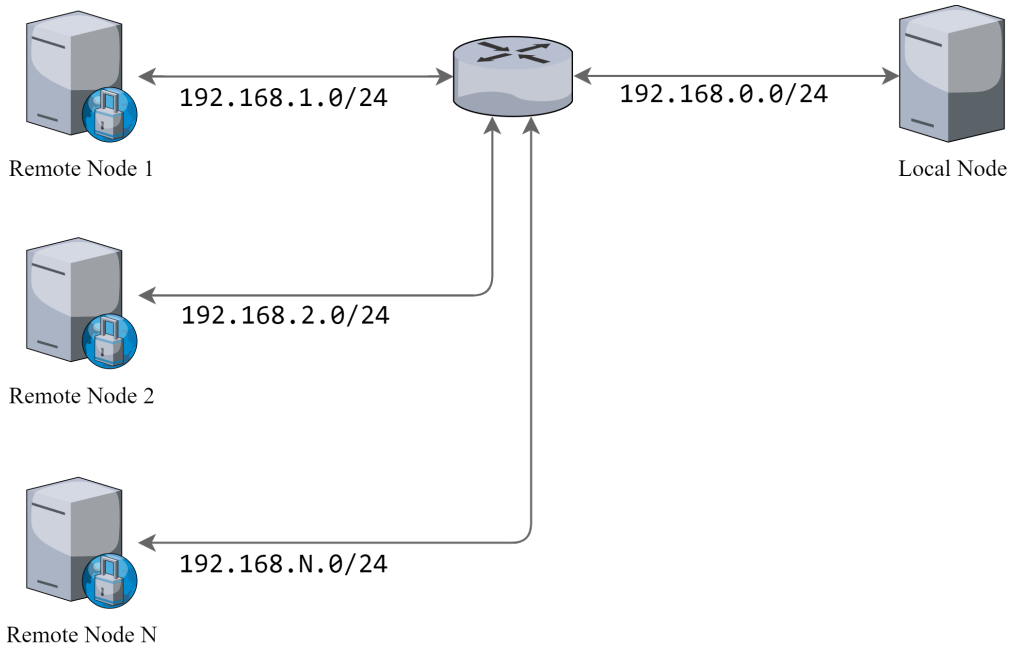

Figure 5.1: Initial Testbed Network Topology Setup

Initially, the test bed architecture comprised only four different systems. These included two remote nodes, a router system, and an exit node. The architectural arrangement was such that the two remote nodes and the exit node were directly connected via the router system, resembling a star topology. This router acted as a central interface that facilitated communication via a Layer 3 Link between the three aforementioned components and ensured that both data flows and network management processes were efficiently routed. This way the testbed solely reflected the application architecture (cf. Figure 1.1) as shown in Figure (cf. Figure 5.1).

However, at a later stage of the project development, it became clear that two additional nodes were still required to fully represent the planned network scenario: an Attacker Node and a Honeypot Node. It was essential to integrate the honeypot node into the same internal network as the local node. This was necessary to represent the network topology and interactions realistically and to ensure that the honeypot could operate effectively. This expanded setup resulted in a more comprehensive representation of the deployment scenario and created a controlled environment conducive to in-depth analysis and development. As part of the modeling of the test bed, however, a simplifying

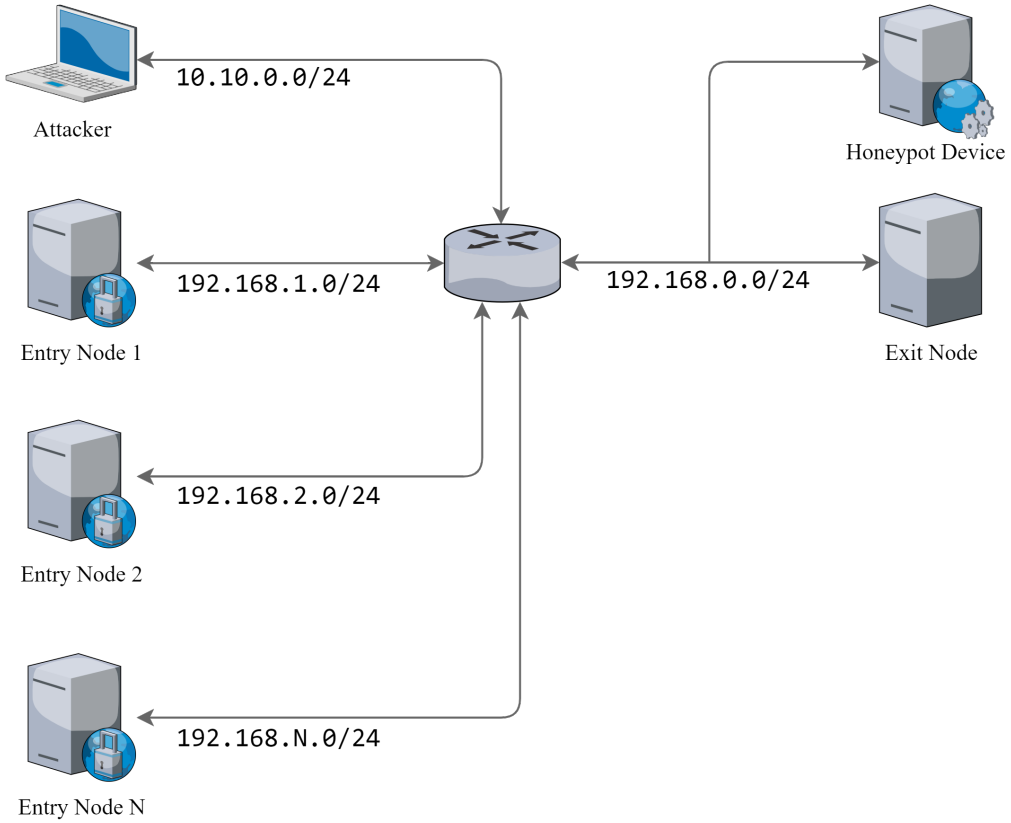

Figure 5.2: Testbed Network Topology Setup

assumption was made with regard to the network topology. The Layer 3 router, which interconnects the various instances, was used as a representation of the entire Internet. This is, of course, a considerable simplification compared to the actual complexity of the Internet. Furthermore, to simplify the routing process, it was decided to define only static routes on the individual hosts. This means that a fixed path on the network was manually defined for each route rather than using dynamic routing protocols that would make the path decision in real time based on various factors. These simplifications were necessary to keep the focus on the core elements of the project and to keep the implementation effort within manageable limits. To test the system's ability to handle concurrent traffic, the test bed was designed to provide dynamic scalability regarding the number of remote nodes. This means that depending on the test requirements and scenarios, the amount of remote nodes can be flexibly adjusted. This flexibility ensures that different network situations and load scenarios can be simulated during development and testing, as well as reducing the load on the host system when not all remote nodes are required to operate. It also enables comprehensive evaluation of the system under varying conditions, which is crucial for creating a robust and reliable system resulting in the system shown in Figure (cf. Figure 5.2).

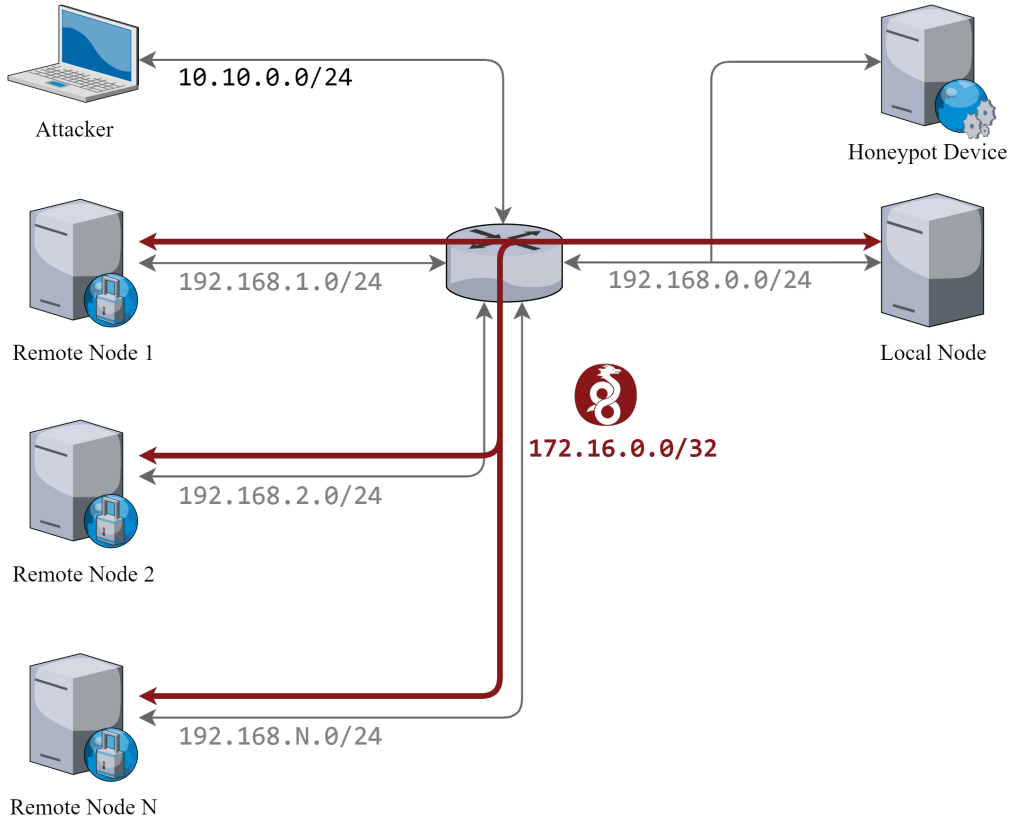

Figure 5.3: Testbed Network Topology Setup with Wireguard Overlay Network

In the initial planning phase, it was planned to integrate WireGuard into the tunneling system. It was therefore deemed necessary to also implement a WireGuard solution in the test bed. WireGuard is based on a server-client principle. As part of the test bed, it was ensured that the setup and configuration of WireGuard were automated to facilitate development and testing. It is worth noting that this implementation was designed to be flexible so that it is compatible with a variable number of remote nodes. This offers the advantage that the system remains able to accommodate and dynamically adapt to different network topologies and configurations while still not being limited in functionality. The implemented tunnels using WireGuard provide a transparent way to establish a direct connection between the remote node and the local node. This ensures immediate communication between these two components while hiding the complexity of the underlying network from traffic. As per connection from a remote node, it is added to the created overlay network as shown in Figure (cf. Figure 5.3). Since the underlying network infrastructure is not visible, it would not be necessary to simulate the rest of the network topology for tunneling using an existing VPN solution exclusively.

5.3 Using eBPF for packet handling

With the test bed successfully set up, we could now tackle the next crucial step: the actual implementation of the application. This robust test environment provided us with an ideal basis to effectively and systematically pursue the main goal of the project. As the project progressed, the focus was now on selecting a suitable eBPF implementation and toolchain. Initially, we considered Rust as the implementation language, particularly due to its many advantages in terms of memory safety and performance. However, closer examination revealed that the existing eBPF bindings for Rust were not actively maintained[15] and also lacked stability. This posed a significant risk to the reliability and future-proofing of our project. Additionally, we considered compiling eBPF code manually using the LLVM compiler[16]. LLVM, known for its flexibility and ability to optimize intermediate code, initially seemed an attractive option, especially for custom applications. However, after deeper analysis and evaluation, it became clear that this approach could present several pitfalls and challenges. Manually compiling and integrating eBPF using LLVM could be not only fragile but also rather complex. This added complexity could lead to longer development times, make debugging more difficult, and ultimately complicate project maintenance. Therefore, this approach was deemed less suitable for our project as well.

Given the challenges and complexities that came with manual compilation approaches, we chose BPF Compiler Collection (BCC)[17] as our primary development tool. BCC has emerged as a leading framework of eBPF implementations, offering a wide range of tools and libraries optimized for both eBPF and XDP.

One of the outstanding features of BCC is its ease of use. Not only is it easy to install, as it provides precompiled binaries for the chosen platform, but it also provides a structured and intuitive interface for developers working with eBPF. This makes it easier to get started, reduces the amount of code writing required, and enables rapid prototyping.

In addition, BCC integrates seamlessly with existing Linux infrastructures and provides extensive documentation and examples to facilitate the development process further. Thanks to these extensive resources and the active community that continuously contributes to the improvement and enhancement of the framework, we were confident that we had chosen a stable and reliable platform for our eBPF-based development.

5.3.1 Initial approach: Wireguard

As we progressed with our implementation, the original approach was to use WireGuard as the key component for tunneling. This decision was based on various considerations, in particular, WireGuard's reputation as a modern, secure, and efficient tunneling protocol.

For the successful use of WireGuard in combination with XDP, it would have been necessary to implement XDP on both the physical and virtual WireGuard interfaces.

[15] `https://github.com/foniod/redbpf` Archived since Jul 11, 2023 (Accessed on 26.08.2023)
[16] `https://llvm.org/` (Accessed October 2023)
[17] `https://github.com/iovisor/bcc/` (Accessed on 26.08.2023)

WireGuard's design is based on the principle of lean and efficient tunneling, which led to the assumption that this integration would work seamlessly and without major difficulties.

However, we encountered significant challenges during our implementation attempts. It became apparent that XDP and WireGuard did not interact optimally. The main problem was that XDP was not working correctly with the WireGuard interface, an issue not specific to our setup [50]. This presented us with a critical dilemma, as the core functionality of our solution relied on these two technologies interacting efficiently and correctly.

Several approaches were considered to address this issue, including customization of the XDP implementation. Nevertheless, our investigations revealed that none of the adjustments on the XDP side could solve the problem with WireGuard. This suggested that the incompatibility could not be fixed by modifications to XDP but that there were deeper problems in the interaction between XDP and WireGuard.

One potential solution strategy could have been to create our own implementation of WireGuard since WireGuard's source code is openly available. The ability to make direct changes to the WireGuard code base would have theoretically given us the freedom to address the incompatibilities with XDP and develop a customized solution for our specific requirements. However, this approach would have presented significant challenges. Interfering with an existing and proven code base, especially in an area as critical as network security, would not only have presented significant technical difficulties but would also have risked unexpected side effects and security issues. In addition, the effort required for such a custom adaptation would have been enormous. Developing, testing, and maintaining a custom WireGuard version just to resolve this particular incompatibility would require significant resources and would greatly expand the scope of our project. Given these considerations, we concluded that, although theoretically feasible, it would not make sense from a practical standpoint to pursue this path. It seemed far more purposeful to consider other approaches to achieve the desired functionality without the associated high risks.

In addition to the challenges already mentioned, another significant problem emerged that could not be solved even with a custom WireGuard implementation. This problem concerns the simultaneous execution of two XDP programs: one on the physical network interface and the other on the virtual WireGuard interface.

The need to run two XDP programs in parallel that must interact with each other is not optimal. The interaction between these two programs results in significantly increased system complexity. Each additional component or interaction introduced into a system not only increases the likelihood of errors but can also affect maintainability, transparency, and, ultimately, the stability of the entire system. It is always desirable to keep systems as simple as possible, as this reduces the likelihood of errors and facilitates long-term maintenance. In this context, adding two XDP programs to run and interact with each other at the same time could have created significant technical challenges and potential operational problems.

Any network packet addressed to the virtual WireGuard interface would also be captured by the physical interface first. As a result, inbound packets destined for the WireGuard

interface would have to be processed twice - once by the XDP program on the physical interface and once by the XDP program on the virtual interface. This duplicate processing would not only be redundant but would also affect the overall efficiency of the system. The resulting overhead costs would be significant and could lead to performance problems, especially at high data rates or with many concurrent connections.

Given this added complexity and the potential impact on system performance, it became clear that the integration of WireGuard in conjunction with XDP, at least as proposed, was not optimal. It would have been necessary not only to adapt the WireGuard implementation but also to find a solution to the problem of duplicate processing of packets, which would have significantly increased the effort and complexity of the project. It was therefore deemed inadvisable to pursue this approach.

This meant re-evaluating our implementation approach and looking for alternative solutions or workarounds to achieve the intended functionality without compromising the benefits of either technology.

5.3.2 Reconsidering the Event Hook

Given the present challenges regarding XDP and WireGuard in our intended network topology, we continued to evaluate the use of WireGuard as the underlying technology. While XDP represents the primary point of intervention in the Linux kernel packet processing pipeline, the eBPF framework allows manipulation and monitoring of system behavior via a variety of other event hooks in the operating system.

As a result, we have focused on event hooks that start at higher levels of abstraction in the kernel's network stack. The goal of this investigation was to identify a suitable hook that intervenes after the initial processing of the XDP event but before the traffic is forwarded to WireGuard. This methodological investigation intended to preserve WireGuard's potential as a robust and efficient tunneling protocol while leveraging the powerful and flexible networking customization capabilities of the eBPF framework.

The approach of using event hooks at a higher abstraction level of the network stack offers the significant advantage of being able to monitor and manipulate both inbound and outbound network packets. This is in contrast to the use of XDP, which can only handle inbound packets on a single interface. This higher level of abstraction can enable a more coherent and integrated approach by reducing complexity and consolidating implementation into a unified program framework. This not only facilitates development but also potentially reduces error-proneness and maintenance costs.

In the search for an adequate event hook to meet the specific requirements of our project, we considered various techniques and approaches. In particular, we looked closely at kernel probes, queuing disciplines, and the raw socket hook.

- Kernel probes, commonly referred to as kprobes, while providing an efficient method of instrumenting kernel code at runtime and gaining insight into kernel activity, do allow monitoring but not the modification of the network traffic passing through.
- Queuing disciplines, used in the kernel to manage inbound and outbound network packets, do provide ways to influence network traffic. Still, there are limitations

with this method with respect to non-standard traffic, which may well occur in real-world scenarios, especially in our application, and is therefore relevant in our consideration.

- The raw socket hook gives direct access to the network stack, also allowing inspection and processing of packets. However, the challenge here is that not all traffic, especially non-standard traffic, can be reliably captured, as with queuing disciplines.

Among the analyzed eBPF hooks, although the queuing disciplines appeared to be potentially suitable, we decided to use XDP to achieve optimal results. It is important to note that the other eBPF events considered are not directly comparable to XDP regarding functionality and efficiency.

5.3.3 Custom Tunneling

Due to the multiple shortcomings and technical hurdles that arose in conjunction with WireGuard, we made the decision to choose XDP as our primary technology, leaving WireGuard out of the equation. Our analysis showed that optimizing and stabilizing our system by using XDP, without the additional complications of WireGuard, was more efficient and targeted. Given the challenging implications of WireGuard, we felt it necessary to develop our own tunneling mechanism. Although this increases the complexity of the XDP program, we consider this approach an acceptable trade-off. Developing a custom tunneling mechanism gives us more control and flexibility, and the increased complexity that comes with it outweighs the disadvantages of integrating WireGuard.

The decision to implement tunneling itself entailed a number of consequences and possibilities. The first and most important was determining the specific type of tunneling to use. It was inevitable that we would have to encapsulate the traffic on Layer 3 to include routing information to ensure the mapping already highlighted in previous discussions and visualizations. This necessity stemmed from the technical and functional requirements of our project.

However, now that we had control over the tunneling process itself, we had the opportunity to directly determine at which network layer this encapsulation should occur. This opened up a discussion about which layer would be best suited to achieve our specific goals while also considering performance and implementation complexity. The choice of layer not only affects how data packets are processed but can also have an impact on the overall performance and security of the system. Thus, careful consideration of the advantages and disadvantages of each option was required.

Encapsulating IP traffic directly in IP, i.e., so-called IP-in-IP tunneling, offers some decisive advantages that are particularly relevant to our project. A primary advantage of this method is the efficiency achieved. By enclosing IP packets directly in IP, the structure of the data remains relatively unchanged, resulting in a minimal increase in packet size.

Another significant advantage of this technique is the reduced bandwidth overhead. With many other tunneling techniques, noticeable overhead is created by additional

header information and protocol data. In contrast, IP-in-IP tunneling introduces only one additional IP header, which significantly minimizes the overhead compared to other methods. This feature is especially beneficial in network environments where bandwidth is a critical factor.

In addition, this method simplifies packet handling and processing because network equipment and applications are already optimized to work with IP data packets. It, therefore, eliminates potential complexities and challenges that could arise from working with foreign or less common protocols. Overall, IP-in-IP tunneling thus offers a lean and efficient approach that fits well with our requirements.

Despite the obvious advantages of IP-in-IP tunneling, we ultimately decided to encapsulate the traffic in a stateless transport protocol, specifically UDP, as Wireguard does. While IP-in-IP tunneling is an attractive paradigm in its simplicity and directness, it still poses risks in terms of general acceptance in different network environments. Some network infrastructures, including cloud providers, firewalls, or ISPs, might recognize IP-in-IP packets as anomalous and interfere with them accordingly [4] [39]. In contrast, UDP proves to be a widely and universally accepted protocol that usually flows through diverse network topologies without any problems.

A key advantage of UDP is its statelessness, which means that no connection needs to be established or maintained. This makes it an efficient vehicle for our tunneling scheme. Additionally, the UDP header adds only 8 bytes of overhead, which is minimal in terms of bandwidth efficiency. This minor overhead, coupled with the wide acceptance of UDP, led to our decision to encapsulate IP traffic in UDP packets. This allowed us to ensure that our system would work smoothly in most network environments without being constrained by potential protocol-specific limitations.

5.4 Interacting with Network Traffic

Based on the findings and considerations discussed earlier, we decided to largely retain the implementation algorithm originally presented (cf. Figure 4.3, Figure 4.4), but with the important modification of using UDP as our primary transport protocol. This approach entails more complex processing of network traffic than initially anticipated.

One of the primary challenges here is the need to adjust both the IP and MAC addresses of the packets being transmitted. These changes are not trivial and require precise configuration. As packets traverse the network, the IP addresses in the packet headers must be changed by our application to ensure that each packet is correctly forwarded and delivered to its true destination. Proper manipulation of the IP addresses is essential to effectively route the encapsulated packet through different network topologies and environments.

Furthermore, MAC addresses, which are at a lower level of abstraction in the OSI model, must also be adapted. This is particularly important since MAC addresses are responsible for correct packet delivery within a local area network, both of the local and the remote nodes. Rewriting MAC addresses ensures consistent and interference-free communication between the participating devices within the tunneling system.

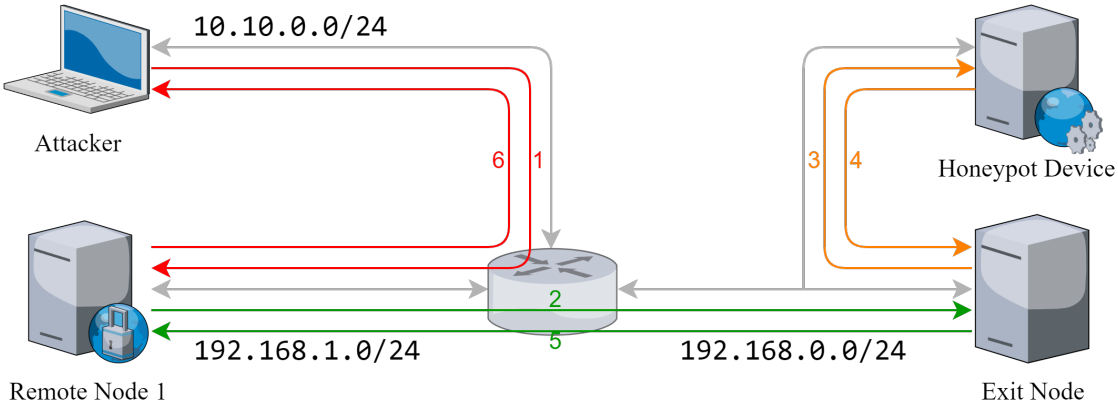

Figure 5.4: Inbound and Outbound Traffic flow

In Figure (cf. Figure 5.4), the planned flow of packets through the system is shown in detail. The sequence of this packet flow is numbered numerically for clearer distinction. Furthermore, different color schemes have been used for better visualization:

- Red marks the communication between the attacker and the remote node. In this context, the remote node represents the supposed honeypot, which is the primary target of the attacker.
- Green visualizes the encapsulated traffic. This traffic originates from the attacker's interaction and is encapsulated using our mechanism to deliver it safely and transparently from the remote node to the local node and vice versa.
- Orange represents the modified traffic that takes place between the local node and the actual honeypot. This traffic has been previously decapsulated and modified to ensure that it matches the expected effect at the honeypot that the attacker would hope to achieve.

5.4.1 Remote Node

As part of our implementation strategy, we decided to start with the remote node first. This step made logical and strategic sense, as the overall functionality and behavior of the local node are highly dependent on the mechanisms and processes implemented in the remote node. The foundation laid by the correct implementation of the remote node serves as the basis for the subsequent development of the local node and ensures a coherent and efficient implementation of our entire system.

Handling L2 Traffic

In traditional network communication scenarios, MAC addresses are of central importance because they represent the physical addresses of the devices in the local network. For

effective communication within the tunnel system, packets must be modified so that they are correctly forwarded to the respective target device. It is important to note that in Ethernet environments, the address information is modified at every hop of the packet.

One of the mechanisms we have implemented to meet this requirement is to swap the receiver and source MAC addresses for inbound packets on the remote node. This method ensures that response frames are correctly returned to the original source device. This means that when a packet is sent from device A to device B, the MAC address of device B becomes the source address, and the MAC address of device A becomes the destination address when device B replies to device A.

In the network architecture in which the remote nodes are deployed, i.e., in cloud scenarios, the network structure often follows a tree-like topology [9]. In such structures, there is usually a central node (for example, a main router or gateway) through which all traffic is routed. Therefore, from the remote node's point of view, the same peer is always communicated with, regardless of whether it is inbound or outbound traffic. This structural aspect is crucial for our address rewriting approach. Swapping the receiver and source MAC addresses in such a scenario ensures that packets are correctly sent back to the original destination since they always take the same path back through the network topology (cf. Figure 5.4). In the tree-like structure, packets sent from the cloud back to the remote node are always routed through the same path, making MAC address rewriting and reuse a particularly efficient mechanism in this environment.

Another significant advantage of swapping MAC addresses in our approach is that the process is stateless. Specifically, this means that no additional information needs to be stored to rewrite the addresses. This has two major implications: First, statelessness leads to a significant reduction in complexity. Since there is no need to store context-specific information or tables about active sessions, we can ensure that our system operates efficiently and without significant delays, even in the presence of high traffic.

Second, statelessness contributes to the greater robustness of the system. Without the need for a state, there are fewer attack vectors, and there is no risk of errors due to outdated or incorrect state information. In addition, the independence from state storage improves the scalability and maintainability of the system.

However, to optimize performance, we store the MAC address of the remote node as a static state, which avoids the repeated allocation of memory, especially since the MAC address does not change during runtime.

In summary, by simply swapping MAC addresses, the stateless approach provides not only an elegant but also a powerful and secure solution to the L2 address rewriting problem for the remote nodes in our tunneling system. It not only enables correct packet delivery but also ensures consistent and transparent communication, which is crucial for the smooth functioning of our tunneling system, as all further layers are built on top of it.

Handling L3 Traffic

Of course, it is equally essential to modify and interact with IP traffic accordingly to successfully implement the encapsulation (cf. Figure 4.3) and decapsulation (cf. Figure

4.4) algorithm. In contrast to L2 address information, L3 addresses are typically not changed as they state the original sender and the final destination of the packets.

When dealing with L3 traffic, in our case IP traffic, in the remote node, further specific requirements arise. In this environment, inbound IP traffic is encapsulated. The key here is to distinguish between outbound and inbound traffic, with the IP address serving as the decisive criterion for this.

Identification is based on a simple but effective principle: If a packet has the IP address of the local node as its source address, it is outbound traffic from the point of view of the remote node. Otherwise, the inbound traffic is considered to originate from an attacker addressing the honeypot device.

However, for this distinction and correct traffic handling, a minimal static condition is required: Knowledge about the IP address of the local node. This information must be stored in the remote node and represents a constant reference point that can be used to determine the direction of traffic. As with MAC addresses, this static state has the advantage that no dynamic adjustments are necessary during runtime, thus avoiding potential sources of error and complexity. Besides that, we have also implemented a whitelisting mechanism to allow traffic to bypass our system and instead forward any packets matching any whitelisted IP address to the operating system network stack. This feature is especially critical considering that access to the devices running our software still must be maintained for management purposes.

Encapsulation When the remote node processes the traffic, it goes through a multi-stage inspection process. First, the IP header of the inbound packet is inspected to identify the source IP address. This step is critical to determine if the traffic originated from the local node or if it came from another device within the network.

If the source IP address of the inbound packet does not match the IP address of the local node, the packet is further examined to determine to which protocol the payload belongs. This is done by analyzing the corresponding fields in the IP-header to determine whether it is TCP, UDP, or ICMP traffic.

If one of these protocol types is detected, the data packet is classified as relevant and enters the encapsulation process. Here, the original packet is converted into a new format suitable for further processing and transport over the network. This encapsulation mechanism ensures that only relevant traffic is processed and not necessarily all data packets reaching the remote node. If the inbound data packet on the remote node does not match one of these specified protocols, it is not processed further. Instead, the packet is immediately dropped and not forwarded further into the network or processed by the operating system network stack. This mechanism ensures that only relevant and expected traffic flows through the system, minimizing the risk of unwanted or potentially malicious traffic that could expose the tunneling system. Any traffic that does not meet the specified protocol criteria is thus effectively ignored and eliminated.

The encapsulation process on the remote node follows a structured flow to ensure that inbound data packets are processed properly and efficiently, implementing the Algorithm as described in (cf. Algorithm 1):

1. Preprocessing: First, the source MAC address of the inbound packet is extracted and temporarily stored. This MAC address is later used to send back the processed packet. Also, the destination IP address, which represents the address of the device itself, is read and stored for later use.

2. Memory allocation: Before encapsulation begins, space is allocated and prepended for the new outer headers. For this purpose, an additional memory space of 28 bytes is reserved before the original data packet. This composition is divided into 20 bytes for the outer IP header and 8 bytes for the encapsulation UDP header.

3. Header redefinition: First, the new MAC header is created, in which the address information contains the swapped MAC addresses compared to the inbound Ethernet header. Next, the IP header is created that specifically addresses the IP address of the local node. Also, the previously stored address of the device is used as the source address Finally, the UDP header is added, which is positioned directly in front of the original, unmodified IP packet. During this process, the original Ethernet header will be overwritten and, therefore, does not need to be considered anymore. In summary, this process transforms the original data packet of any protocol into a UDP packet, replacing the original MAC header with a new one and prepending additional headers to route the packet to the local node correctly.

Decapsulation To ensure that only relevant packets are handled, the decision process not only checks whether a packet is already encapsulated or not. An additional criterion is the check of the transport protocol. Here, it is specifically checked whether the packet is a UDP packet and whether the source and destination port used is 17227[18]. This port was specifically defined for the use of this application. This additional check prevents unwanted or irrelevant traffic from being incorrectly processed by the decapsulation logic. Thus, this is an additional security and efficiency measure in the process.

The decapsulation process on the remote node, which is essentially a reverse of the encapsulation process, proceeds as follows (cf. Algorithm 2):

1. Preprocessing: First, the destination MAC address is extracted from the inbound encapsulated packet and cached for subsequent processing.

2. Encapsulation removal: The next step is to remove the first 28 bytes of the packet - i.e., the space allocated for the outer IP header and the UDP header. As a result, the data packet is reduced in size, and only the original content remains.

3. Rewriting the Ethernet header: A new Ethernet header is then created at the beginning of the packet using the previously stored MAC address, overwriting the

[18] For the selection of UDP port 17227, we were guided by the list of port numbers specified by the Internet Assigned Numbers Authority (IANA). The specific port was identified as unoccupied after systematic verification, minimizing the likelihood of port conflicts with existing applications. It was essential to select such an unoccupied port to avoid potential interference or unforeseen operational disruptions. [25]

remaining bytes of the encapsulation headers. This is necessary to ensure that the packet is sent correctly to its original destination.

In summary, during the decapsulation process, the encapsulated packet is returned to its original format, with only the MAC addresses adjusted accordingly to ensure correct delivery.

Within our decapsulation concept, the algorithm (cf. Algorithm 2) in the remote node is based on a fundamental premise: The initial preprocessing and modification of the packet address information has been completely and accurately performed in the local node. This implies that the packet encapsulated within the outer frame already contains adequate address information. Consequently, additional modification or re-evaluation of this address information during the decapsulation process is obsolete, which ensures an increase in process efficiency and reduction of potential error sources.

Preliminary Validation of the Implementation

During the implementation phase of the Remote Node, we were faced with the challenge that no direct communication partner was yet available for testing and validation. In order to still be able to perform a review and analysis of the development process, we made use of a special capability of XDP: XDP allows redirecting processed packets to userspace instead of reflecting them directly into the network.

This method allowed us to get detailed feedback on the packet modifications and encapsulations performed in real time. By intercepting these packets forwarded to userspace, we were able to perform an in-depth analysis of the encapsulated data streams. For this inspection, we used the widely used packet inspection tool Wireshark[19]. Wireshark allowed us to inspect each segment and header of the modified packet at a granular level and ensure that the packet manipulations met our exact specifications and expectations. However, the decapsulation functionality presented a further challenge, as it required interaction with a second communication partner. A solution to this dilemma was provided by configuring two instances of the remote node and creating a simple one-to-one tunnel. This setup allowed for a more comprehensive analysis of both encapsulation and decapsulation operations by having the two instances communicate with each other, ensuring simulated end-to-end traffic. This approach was critical to ensure that the remote node implementation worked correctly before integrating it into a broader system and extending the functionality to implement the local node.

5.4.2 Local Node

After the successful completion of the lightweight remote node implementation, we focused on the implementation of the local node. Here, we used the knowledge gained in the remote node as a basis and starting point to ensure that both components interact seamlessly and efficiently with each other.

[19] `https://www.wireshark.org/` (Accessed October 2023)

Handling L2 Traffic

Handling traffic at the L2 level, especially in the context of MAC addresses, turns out to be significantly more complex on the local node than on the remote node. This is primarily due to the structure and requirements of the local network.

In this particular context, it is essential to communicate with two different partners, as shown in Steps 2–5 in Figure (cf. Figure 5.4): First, with the remote node or gateway of the local network, and second, with the honeypot device.

It is essential to have prior knowledge of the specific MAC addresses of both the gateway and the honeypot device. In contrast to the situation on the remote node, where a stateless procedure is possible by simply swapping the MAC addresses, the local node requires the management of some state.

This state, more precisely, the information about the MAC addresses of the gateway and the honeypot, is essential to address and route traffic correctly. However, it is a static state, as this address information should not change at runtime, and thus, no dynamic adaptation or constant updating is required. This minimizes complexity and the risk of inconsistencies during operation. It thus provides a stable basis for the address rewriting procedure and further traffic handling on the local node.

This process ensures that traffic is correctly routed to the intended destination, be it the remote node, the local network gateway, or the honeypot device, and that the appropriate responses are correctly returned to the original sender.

Handling L3 Traffic

The processing of IP traffic in the local node is conceptually very similar to the implementation in the remote node. Rather than redesigning or significantly modifying the existing functionality of the remote node, we focused mainly on adding the necessary features to meet the specific requirements of the local node. This not only ensured a consistent approach across both nodes but also allowed for an efficient implementation by building on already-established concepts.

In the local node, inbound traffic is carefully analyzed and processed using the IP address criterion as in the remote node. If a packet has the honeypot's IP address as the source address, it is encapsulated for further processing and sent to the corresponding remote node. For packets with any other IP address, further differentiation is made: If the packet uses the UDP transport protocol and both the source and destination ports match the previously specified value, decapsulation is performed. This packet is then forwarded directly to the honeypot device.

As with the remote node, the local node also incorporates a whitelisting mechanism based on IP address recognition to ensure management of the system.

Decapsulation The decapsulation of traffic in the local node is similar in many aspects to the process in the remote node. However, there are significant differences that must be taken into account in the local node. After the actual decapsulation, additional processes are run in which certain information is extracted and stored. This information is essential

to update the current state of the application. The main purpose of this state update is to ensure that outbound traffic can be associated and routed correctly afterward. This ensures that traffic redirection remains consistent and reliable in a dynamic environment where communication patterns can change constantly.

When processing inbound traffic in the local node, the following steps are performed:

1. Decapsulation: Once an inbound packet arrives in the local node, its contents are first revealed through the decapsulation process. This is done analogously to the procedure in the remote node, where the outer header of the packet is removed to expose the original content.

2. Attacker Address Association: To ensure proper traffic flow, the next step is to capture the destination address (which is the address of the remote node) of the decapsulated packet and associate it with the source address (which is the address of the attacker). This association allows future packets to be routed correctly and ensures that responses are returned to the correct attacker. This step is performed every time, even if the attacker's address is already known. We do this because, on the one hand, it is possible that the attacker enters the system through another remote node, and on the other hand, it increases the efficiency of the system because fewer operations have to be performed since there is no need to query in advance.

3. Logging of the last remote node used: In order to keep the system flexible and responsive, the last remote node used for data traffic is also recorded after the association step. This information can be essential, especially in situations where packets are sent to a previously unknown third-party entity.

4. Adjusting the destination IP address for the honeypot: The final but crucial step is to rewrite the destination IP address in the packet so that it now points to the actual honeypot rather than the remote node. This ensures that the packet is successfully forwarded to the honeypot and that the honeypot considers the packet legitimate.

Encapsulation Encapsulating traffic in the local node represents the most complex step in the entire system and corresponds to the conversion of traffic from its state in step 4 to its state in step 5, as shown in Figure (cf. Figure 5.4). The encapsulation process in the local node is mostly the same as in the remote node. But before the actual encapsulation, there are specific modification steps that are applied to the packet (cf. Algorithm 4).

1. Remote Node Lookup: First, the packet is inspected to identify the original destination IP address. This information is used to determine to which specific remote node the packet should be forwarded. If the target address of the remote node is looked up, there is a possibility that the honeypot device will not respond directly to the attacking device but instead to a third, previously unknown entity. In such a scenario, it is problematic to select a specific remote node because the destination address cannot be identified. To circumvent this problem, a method has been

implemented to trace which remote node was last used for outbound traffic. This ensures that traffic, even if directed to an unknown third party, is still forwarded via the last active remote node.

2. Address Redefinition: Once the correct remote node is identified, the next step is to change the source IP address of the inbound packet. The original address belonging to the honeypot is replaced with the IP address of the previously identified remote node. This process is a critical part of the system as it ensures that the true identity of the honeypot remains obscured from any attackers.

3. Encapsulation: After adjusting the IP address, the packet is encapsulated. This step follows the exact same process as for the remote node. An additional header is added to prepare the packet for transmission, ensuring that the integrity and confidentiality of the data are maintained during forwarding. The previously determined remote node address is used again, this time as the destination address of the encapsulation header.

5.4.3 Handling of ARP Requests

In a complex network environment, not all traffic types are the same, and it is often necessary to handle certain traffic types differently to ensure that the network functions smoothly. One such example in the context of the implementation of this system concerns the handling of Address Resolution Protocol (ARP).

ARP plays a central role in IP networks and is responsible for resolving IP addresses to their associated MAC addresses. When a device on a network attempts to send data to an IP address, it needs the MAC address of the destination device. ARP makes it possible to discover this MAC address. Without ARP, a device would not know which MAC address to send its data to, effectively bringing the network to a halt.

Therefore, to integrate our system into existing networks, it was essential to give special treatment to ARP traffic on the local network. Not handling ARP requests or handling them inaccurately by treating them as traffic to be encapsulated would result in devices on the network not recognizing the network route to each other and, therefore, not being able to communicate.

Consequently, an exception for ARP traffic was integrated into the system during development. This ensures that ARP requests and responses are processed correctly and that the network topology and IP address management can function without interference. It was critical to exclude ARP traffic from the regular encapsulation and decapsulation process and instead ensure that ARP functions correctly and IP address resolution is guaranteed on the network.

When dealing with ARP requests in our system, we faced a decision-making process regarding the best approach to handle this particular type of network traffic:

- Direct handling in XDP: One option was to design the eBPF program within XDP to directly detect and respond to ARP requests. While this would have offered the advantage of faster processing at the XDP level, it would have been necessary

to manually implement many details of the ARP protocol in XDP, introducing additional complexity and potential sources of error.

- Delegation to the operating system: As an alternative solution, we could leave ARP requests untouched and forward them to the operating system's network stack. This approach relies on the operating system's proven mechanisms for handling ARP, significantly reducing the risk of implementation errors, although potentially at the expense of immediate processing speed.

After careful consideration of these options, we finally decided to leave the handling of ARP traffic to the operating system. This ensured higher reliability and maintainability by relying on the established network mechanisms of the operating system and explicitly not providing any special handling for ARP within XDP.

5.5 Recomputing Checksums

During the development phase of our system, we found that in addition to the conceptual hurdles, the need to recalculate checksums, in particular, was one of the most complex and challenging tasks. Checksums are an integral part of network protocols and are used to ensure the integrity of data packets during transmission. Any modification to a packet's data, including its header information, requires an adjustment to the associated checksum in order for the packet to continue to be recognized as valid.

In our system, the modification of header information for L3 and L4 network traffic is central. This is due to the basic functionality of the system, which aims to disguise the origin and destination of network traffic. For each of these modifications - whether rewriting IP addresses, changing port numbers, or adding and removing headers - the associated checksum must be correctly recalculated.

Correct checksum calculation is not only technically challenging but also critical to the functionality of the entire system. An error in this step can cause packets to be detected as corrupted by any of the devices involved in transmission and discarded. This would, in turn, reduce the effectiveness of our system and could lead to unexpected behavior or even system failures.

In addition, we had to ensure that checksum calculation was performed at high speed to avoid introducing significant latency into the network path. This led to further challenges in terms of performance optimization.

The eBPF framework already provides helper functions at some higher abstraction levels to facilitate checksum calculations of network packets. These functions make working with eBPF much easier by abstracting complex calculations and allowing developers to focus on the core logic of their applications. In the ideal world, these features would be available in all eBPF environments.

However, we encountered a limitation in our technology stack: XDP does not offer these convenient utility functions. This presented us with a particular challenge, as the implementation of checksum calculations in XDP must be done manually and from scratch. This meant that we had to implement checksum calculations ourselves without

relying on predefined helper functions. In particular, we had to calculate checksums for different protocol types. At the L3 level, this applied to the IPv4 protocol, while at the transport level (L4), calculations had to be performed for TCP, UDP, and ICMP. Each of these protocols has its own structure and idiosyncrasies that must be incorporated into the calculation, which further complicates the implementation process. It also requires a deep understanding of the protocols and their respective checksum algorithms.

In conclusion, although the lack of abstraction in XDP presented challenges, it also gave us the opportunity to develop a custom solution that was tailored to our exact requirements. It was a task that required both technical skill and a deeper understanding of the underlying protocols, as incorrect checksums lead to packets being recognized as invalid.

Overall, these requirements highlighted how critical the correct handling of checksums is in a system that performs active packet modifications and how essential it is to be precise and efficient in doing so.

In our effort to implement proper checksum calculation in XDP, we started by following the standards of RFC 1071. This RFC defines in detail the method for calculating the checksum to ensure the integrity of data packets in IP networks.

When implementing the checksum according to RFC 1071, it should also be noted that the checksum is calculated as the complement of the total sum of the 16-bit sections of the packet. This means that each 16-bit word of the header is added together. Then, the resulting sum is inverted using one's complement. The result of this calculation is the checksum, which is inserted into the header [45].

However, the requirements of RFC 1071 presented us with a complex task, primarily because the algorithm requires that the entire data of the packet or header be read, which may not always have a fixed size. In the context of XDP, where packet processing performance and efficiency are of paramount importance, this approach was particularly challenging. Dynamically checking pointer boundaries in XDP is not trivial. Reading large amounts of data from a network packet requires constant monitoring and updating of the current read pointer to ensure that it is not accidentally read beyond the limits of allocated memory.

In addition to the inherent challenges the algorithm presented, checking pointer boundaries in XDP was not only complicated but also inefficient. Each of these checks adds additional CPU cycles to a process already burdened by reading and processing large amounts of data. In short, although RFC 1071 provides an established and proven method for calculating checksums, its requirements, combined with the specifics of XDP, presented a significant challenge, both in terms of implementation complexity and efficiency.

5.5.1 Incremental Updating of Checksums

The initially implemented method for calculating the checksum, as described in RFC 1071, proved to be less than optimal for our requirements. In particular, the calculations involved, which had to take into account the entire packet contents, were both complex to implement and inefficient in terms of execution. This inefficiency resulted from the need to use XDP to dynamically check the bounds of the pointers, which led to increased

processing times and decreased performance. Given these limitations, we searched for an alternative method of checksum computation that would be more efficient and easier to implement. Our search led us to RFC 1141 and its revision in RFC 1624, which describes a method for incremental checksum calculation. Here, instead of completely recalculating the headers checksum, only the difference resulting from the change is considered and added or subtracted to the original checksum. This incremental method simplifies the process considerably since only the modified parts of the packet need to be included in the calculation [35, 44].

The implementation, according to RFC 1624, offered us two main advantages. First, the increase in efficiency, since only a small section of the package needed to be looked at for the checksum update. And second, the reduction in complexity, since no complete review and recalculation of the entire packet contents was required. Therefore, this approach was a significant improvement over our initial implementation.

5.5.2 Omitting Checksum in Encapsulation Header

The decision not to compute a checksum for the encapsulation UDP header was made after careful consideration of the advantages and disadvantages. Two main reasons significantly influenced this decision:

- Computational overhead: Calculating a checksum for the encapsulation UDP header would involve considerable additional computational overhead. This is because not only must the checksum itself be calculated, but it must also be verified by the receiver, i.e., the other node, to be of any utility. Adding this step significantly increases the computational overhead, both when the sender sends the packet and when the receiver receives the packet. Each time a packet is sent or received, these calculations must be performed, which can slow down the entire communication process.

- Integrity of encapsulated headers: Another strong argument against the need for an additional checksum for the encapsulation UDP header is the fact that the inner encapsulated headers - both the IP header and the transport header - have their own checksums. These checksums are already updated when changes are made to the packets. Thus, they already ensure the integrity of the data. Since these headers independently own and update their own checksums, it can be safely assumed that data integrity is maintained by these existing checksum mechanisms.

The decision to calculate the checksum of the encapsulation header would not only have an impact on the computational effort but also on the complexity of the entire implementation. If we had chosen to perform checksum calculations for the encapsulation header, it would have had a profound impact on the overall system.

- Response to damaged packages: Including a checksum for the encapsulation header would have meant that our application would also need to be able to respond appropriately to situations where the payload of a packet changes in transit between

local node and remote node. Such a scenario would mean that the checksum would not match the received data, resulting in an error.

- Need for retransmissions: If our application detects that a packet is corrupted (due to a failed checksum check), it would need to be able to request a retransmission of that packet. This would bring the concept of retransmissions into play. While retransmissions are a fundamental part of many communication systems, they introduce an additional layer of complexity and state information into the system.
- State management: Adding retransmissions would mean that the system would have to maintain a state of packets sent and received. This state would contain information about which packets were successfully received and which need to be retransmitted. Managing this state would add significant complexity to the architecture and implementation of the system.

Taken together, the decision to calculate and verify checksums for the encapsulation header would introduce a number of additional challenges and complexities to the implementation. These additional steps and the associated state would have made the system less efficient and more cumbersome while not introducing significant value. Overall, it was concluded that adding another checksum for the encapsulation UDP header would not add significant value in terms of data integrity but would rather lead to an unnecessary increase in computational effort and implementation complexity. It was, therefore, decided to dispense with this additional checksum.

5.6 Maintaining State

In a networked system like ours, the state represents one of the essential facets. Knowing what state the system is in at any given time enables efficient data processing and secure interactions. In particular, in our system, two distinct types of state exist: static state and dynamic state.

5.6.1 Static State

The static state of a system refers to all information and parameters that remain constant and unchanged over the entire runtime of the system. In the context of networked and distributed systems, the static state is particularly important because it forms the basis for the system's operation and expected actions.

The implementation of a static state in our system benefited significantly from the advantages of the BCC framework. One of the most concise features of BCC is just-in-time compilation. Specifically, this means that the XDP code is recompiled every time our application is launched. This approach allowed us to define specific information directly in the source code at compile time, utilizing the eBPF compiler's macro system. This has the advantage that we don't have to reload configurations or dynamically change states at runtime, which significantly reduces the potential for errors.

By integrating this important information - such as IP and MAC addresses and the specific UDP port for encapsulation - directly into the code, we not only avoid potential sources of error but also increase the efficiency of the entire system. This is because it eliminates the overhead normally associated with looking up or retrieving configuration data from external sources. Combined with constant communication between nodes, this ensures rapid packet processing.

Of particular note, this type of implementation eliminates the need for dynamic adjustments or queries at runtime. In both nodes, the remote node and the local node, we have predefined all the necessary address information, resulting in the remote node being able to function exclusively with a static state. This guarantees trouble-free and efficient communication. Overall, it shows that the BCC framework provides not only a robust but also an efficient method for handling static states in our system.

5.6.2 Dynamic State

In our system, the dynamic state is a critical component, with its implementation strictly limited to the local node. This variable nature of the dynamic state allows for modifications and adaptations during system runtime. Specific to our scenario, this primarily involves IP address information, which is essential to ensure an accurate correlation between an attacker's IP and the corresponding remote node.

To adequately manage this constantly changing state, we have chosen to integrate specific data structures provided by the eBPF architecture. In particular, we use the BPF_HASH data structure, which provides an efficient and direct association between the attacker's IP address and the associated remote node by ensuring constant lookup time independent of its size, which is very important to ensure the requirement of scalability. In addition, we use the BPF_ARRAY structure, which provides the functionality to store data related to the most recently used node. The relevance of these specialized data structures stems from the design of XDP: each block of code is executed in isolation for each individual inbound packet, making a traditional stateful approach infeasible.

This inherent nature of XDP, where each packet processing is autonomous and independent of previous processing, calls for the specific capabilities of eBPF data structures such as BPF_HASH and BPF_ARRAY. They provide the ability to dynamically encapsulate information and systematically retrieve it at a later time. Thus, they ensure consistent and accurate package processing across numerous iterations. Their integration is, therefore, essential to meet the complex requirements related to dynamic state within the local node.

The eBPF data structures not only offer advantages in terms of storing and retrieving state information at runtime, but they also have a remarkable ability to interact with userspace. This bidirectional communication capability allows data to be exchanged between eBPF programs running in the kernel and applications running in userspace. This opens up opportunities for advanced monitoring and analysis tools that can provide deep insights into the operation and performance of eBPF applications.

5.6.3 Enabling Monitoring and Analysis

In addition, eBPF's userspace communication capability opens up the prospect of implementing an implementation that is not compiled immediately at runtime. Instead of compiling each time it is executed (just-in-time compilation), an approach could be taken in which the necessary data and configurations are loaded directly from userspace. Such an approach could potentially contribute to leaner, more efficient application execution by reducing the number of dependencies and eliminating the need for repeated compilation. However, it is important to evaluate whether, in such an application, the associated access to these data structures has significant performance implications compared to our approach of integrating static state into the system.

In the context of developing our system, we were faced with the decision of either integrating standalone monitoring and analysis tools or taking advantage of existing instrumentation tools. After careful consideration and evaluation, we chose to ensure interoperability with existing XDP toolsets, particularly Xdpdump[20], which is maintained by the XDP-developers.

The main reason for this decision was that Xdpdump was already established as a powerful tool for XDP-level packet tracing. By maintaining compatibility with this tool, we can ensure deep package-level analysis without having to reinvent the wheel. Not only does this mean that developers and operators can work with an already familiar toolset, but it also allows us to focus on optimizing and enhancing other aspects of our system without spending resources on developing features that are already effectively covered by existing tools. It also allows us to benefit from future improvements and enhancements to Xdpdump, as our system can be seamlessly integrated with it.

5.7 MTU Issues

The Maximum Transmission Unit (MTU) is a crucial parameter in networks that specifies the maximum size of a data packet that can be transmitted over a network without being fragmented. Each network medium and protocol layer can have its own MTU values, depending on technical and physical constraints. Determining an optimal MTU value is critical to optimize network performance and avoid transmission delays or failures. In the course of implementing our system, we encountered a critical issue related to MTU. Upon closer examination, we found that the user data size, or payload, was reduced by exactly 28 bytes in our use case. This is not coincidental but corresponds exactly to the size of the header we add for encapsulation.

For better illustration, taking standard IP traffic and adding another header to introduce an additional protocol layer (in our case, encapsulation) adds additional data that must fit in the original data frame. As a result, the remaining space for the original payload data shrinks by exactly the size of the added header. This may mean that if a data packet has already reached its maximum size (corresponding to the MTU), adding an additional

[20] `https://github.com/xdp-project/xdp-tools/tree/9aba9e9aee38ee69ac2be4e728b82873cc2b3c03/xdp-dump` (Accessed on 28.08.2023)

header will cause this packet to become larger than the allowed MTU.

This reduction of the effective payload by 28 bytes meant that data packets that were already close to the MTU limit exceeded the MTU due to our encapsulation. This, in turn, would mean that such packets would either have to be truncated or fragmented, which could lead to numerous problems, such as reduced network performance, increased latency, or even packet loss. In addition, such MTU overflows could lead to interoperability problems with network devices or protocols that may not be able to process or forward larger packets correctly. Problems could also arise when fragmented packets need to be reassembled, especially if the network paths or devices involved vary. This challenge made it necessary to carefully rethink our implementation and design to ensure that the introduction of encapsulation did not lead to unexpected and undesirable network problems. It became clear that we needed to develop mechanisms or strategies to deal with this MTU reduction, whether by adjusting MTU values in the network, performing fragmentation and reassembly processes, or even rethinking the encapsulation mechanism itself.

5.7.1 Existing Solutions

The problem of reducing the effective payload size due to traffic encapsulation is a common phenomenon in network environments and not a unique case of our implementation. In particular, when data packets are modified by adding encapsulation or tunnel headers, the MTU may be exceeded, which leads to various challenges [22], as mentioned before.

Existing tunneling technologies like WireGuard face exactly this dilemma. Again, the introduction of the tunneling protocol adds an additional header to the original data packets, reducing the available space for payload data. To solve this problem, WireGuard implements an elegant solution: it creates its own virtual network interface. This interface has a specially adapted, reduced MTU that is tuned to provide enough space for the encapsulation header. This adaptation ensures that the total size of the encapsulated data packets does not exceed the standard MTU, thus avoiding potential data transmission problems, fragmentation and packet loss. The principle behind this solution is that the use of a separate network interface with a customized MTU makes it possible to control the overall size of the data packets so that they can accommodate both the additional encapsulation header and the actual payload data without exceeding the allowable MTU limits of the physical network interface.

Another approach to cope with the impact of the encapsulation header regarding the MTU is to increase the MTU of all network devices involved in processing encapsulated network packets, as it is done in the infrastructure of Cloudflare [49]. We deemed this approach as unusable for our tunneling application, however, as traffic that is encapsulated will be transmitted via the public internet – infrastructure, which might not be in our control to adjust the MTU.

While there still is the traditional method of using a virtual interface with a reduced MTU to overcome such challenges, we decided to take an alternative approach. Using a virtual interface with a customized MTU would certainly have been a viable solution. It would allow us to add the header for encapsulation without exceeding the default MTU

while ensuring a clear separation between encapsulated and non-encapsulated traffic.

Despite these advantages, however, we decided against this method as default since we wanted our system to be as lean and efficient as possible without adding complexity in the form of additional network interfaces or other additional configuration requirements. Therefore, we were looking for an approach that would allow us to address the MTU problem directly within our system without relying on external solutions or additional system configurations.

However, it is still possible to run the XDP program on a previously manually configured virtual interface with reduced MTU. This configuration option provides an additional layer of flexibility and customizability, allowing the system to be deployed in a variety of network environments and scenarios. It also allows easier integration into existing infrastructures where a reduced MTU has already been specified or where special MTU requirements apply.

This option gives users the freedom to opt for the traditional approach and benefit from its advantages. By using a reduced MTU interface, the system can perform traffic encapsulation without exceeding the specified MTU limit, ensuring traffic integrity and consistency.

5.7.2 Path MTU Discovery

Our focus on dealing with MTU issues led us to seek a method that was both efficient and adaptive. As an alternative to using a reduced MTU on an interface, we decided to support Path MTU Discovery (PMTUD) within our system, as specified in RFC 1191 [15]. Path MTU Discovery is a mechanism that allows nodes to determine the maximum transmission unit along the path to a destination. Basically, PMTUD allows the smallest possible MTU to be discovered across all network segments involved so that data packets do not have to be fragmented or dropped. In this method, the sender sends a packet with a Don't Fragment (DF) flag. When a router or other network device along the path receives a packet that is too large to be forwarded, it sends a "Fragmentation Needed" ICMP message back to the sender. This message informs the sender of the maximum packet size allowed. The sender can then adjust its packet size accordingly. By implementing PMTUD in our system, we were able to ensure that the packet size is always optimally adapted to the network in which they are transmitted. This eliminates the need to set a reduced MTU manually and allows dynamic adaptation to different network environments and conditions. It also provides a robust solution to the MTU problem by preventing packet loss due to oversizing while ensuring that encapsulation headers are handled correctly.

Adding support for PMTUD was a critical step in ensuring the robustness and efficiency of our system in a wide variety of network environments. It allows us to use the full potential of our encapsulation mechanism while ensuring that data packets are transmitted efficiently and without interruptions.

In an ideal scenario, PMTUD enables senders to efficiently use the maximum transmission unit by adjusting the packet size to make the best use of network resources and minimize the probability of fragmentation and packet loss.

However, the reality of network communications is often more complex. One of the main drawbacks of PMTUD, especially when used in conjunction with TCP, is its dependence on ICMP messages, more specifically ICMP Type 3 Code 4 ("Fragmentation Needed and Don't Fragment was Set"). While PMTUD mandates sending packets with the "Don't Fragment" flag, it relies on receiving "Fragmentation Needed" messages to detect and adjust the MTU along a path accordingly [15].

However, problems arise here in networks where firewalls or other security devices block or restrict ICMP messages. ICMP can be blocked for a variety of reasons, such as security concerns or misunderstandings about ICMP's role in the network.

If a firewall blocks ICMP "fragmentation needed" messages, PMTUD is effectively crippled. The result is that the sender continues to attempt to send packets that are too large for the path, leading to the black hole problem. While the TCP handshake often completes successfully in such scenarios, the actual data communication can fail because the packets that are too large are dropped by the network, and the sender is never informed of the need to adjust the packet size [28].

This makes the problem particularly insidious because, at first glance, the network seems to be working correctly (for example, a successful TCP handshake), but data communication can still fail in certain cases. For our system, this meant that we had to be aware that PMTUD, although an effective solution to the MTU problem, might not always work reliably in certain network environments.

6 Evaluation

Evaluating a scientific project is a central element in verifying the effectiveness and relevance of the developed solutions. It not only offers the opportunity to assess the performance and functionality of the created system but also provides sound insights that can be used for optimization or further adjustments. In the context of this master thesis, the evaluation, therefore, forms a crucial milestone to measure the success of the developed approach and to position it against existing solutions.

In this chapter, we focus on evaluating the previously implemented tunneling application from various perspectives. In doing so, we have assessed the initially identified factors of performance, scalability, and usability, with a particular focus on performance, which manifests itself in aspects such as latency, bandwidth, and jitter. In addition, extensive quality tests were performed to check the effectiveness and functionality of our tunneling solution thoroughly. Another essential part of this evaluation is the security assessment of our system, which was carried out on the basis of theoretical considerations. Finally, we also tested the deployability of our solution in real network environments, addressing possible problems, limitations, and the resulting optimization potential. To ensure an objective and meaningful evaluation, both quantitative and qualitative measurement methods were used, as well as a critical evaluation of our measurement methods and their influence on the results presented below. These serve to highlight both the strengths and potential weaknesses or areas for improvement of our application.

The aim of the evaluation is to create a solid basis for discussions, conclusions, and future developments. It is crucial to establish a clear link between the defined goals, the design decisions made, and the results achieved. This will help position the contributions of this work in the context of the broader research field and highlight their relevance to the community.

6.1 Test environment

The quality and reliability of a scientific evaluation crucially depend on the control and consistency of the test environment. Especially in complex applications consisting of multiple interacting components, the test environment can have a significant impact on the results generated. An uncontrolled or inconsistent test setting could lead to results that are either misleading or not repeatable - a serious shortcoming in a scientific context.

A carefully designed and controlled test environment not only ensures the replicability of results but also ensures that the measured performance indicators and other parameters truly reflect the characteristics of the application under test and are not influenced by external factors. The test environment must, therefore, be designed to simulate realistic

conditions while eliminating or minimizing confounding factors that could skew test results.

With respect to an application such as the tunneling application presented in this work, the test environment is particularly critical. The application consists of multiple components that must work together in a synchronized manner. Unanticipated behaviors or interactions between these components could easily be overlooked in an insufficiently controlled environment. In addition, differences in network latency, hardware performance, or other system-related factors could lead to significant variations in test results if not adequately controlled.

In conclusion, the choice, design, and control of the test environment are central to the quality and integrity of the evaluation. It is an aspect that must be handled with great care and attention to detail to ensure sound, reliable, and replicable scientific findings.

6.1.1 Choice of the test bed as test environment

To conduct our evaluation, we chose the test bed mentioned earlier. This decision was logical not only for reasons of continuity and consistency but mainly because this test bed was explicitly designed for such investigation purposes. It provides a structured and controlled environment that allows accurate simulation and monitoring of network scenarios relevant to our application.

The test bed ensures that all tests are performed under comparable and repeatable conditions. It eliminates variables and confounding factors that might occur in less controlled environments. It also provides tools and mechanisms to perform fine-grained measurements and analysis, which is essential for sound evaluation.

Using the test bed specifically designed for such evaluations not only gives us confidence that our results are accurate and reliable but also facilitates the process of data collection and analysis. The infrastructure and tools brought by the test bed allow for systematic investigation, further strengthening the quality and depth of our evaluation.

Therefore, we decided to instantiate the virtual test bed on a Virtual Machine that exclusively serves the test bed. The selected VM is configured to use the operating system Ubuntu 22.04.3 LTS (Jammy Jellyfish) with all packages up to date as of September 2023 and is also configured to support Nested Virtualization. Nested Virtualization is a feature that allows a Virtual Machine to run within another Virtual Machine. This is particularly important for our test environment as we want to test the virtual testbed using Vagrant with multiple nodes. The hardware specifications of the VM have been carefully chosen to ensure that sufficient resources are available for testing but also to minimize overhead due to virtualization. The VM is hosted on an Intel Xeon Gold 6230 processor, where 16 cores are allocated to the VM. This processor and other Intel Xeon series products from the same generation are also used in commercial cloud VMs, among others [36, 21, 2], which is important to create realistic testing conditions. In addition, the VM has 48 GB of RAM, which is sufficient to run the test bed and all the required tools and services.

The test bed itself requires only 9 GB of RAM and 9 CPU cores with two remote nodes. This effectively leaves sufficient resources for managing the virtual environment through Vagrant and other necessary tasks. By carefully selecting the hardware specifications

and using Nested Virtualization, we hope to keep the VM overhead low and thus achieve meaningful and replicable results.

Before we started the performance measurements in the testbed, we took precautionary measures regarding the MTU to avoid potential problems that could occur in connection with our tunneling application. Specifically, the MTU was reduced to a value of 1450 bytes on both the attacker and honeypot sides. This adjustment helps prevent the previously discussed issues regarding exceeding the default MTU that could arise due to the additional headers of our tunneling application. Thus, a smoother flow of the performance measurements was ensured while at the same time ensuring that the measurement results were not affected by unforeseen packet fragmentation or loss.

6.2 Performance tests

When evaluating a complex application like ours, performing an accurate and comprehensive performance analysis is critical. Diverse approaches and methods are available for a detailed picture of application performance. In the following, we present the essential measurement methods to assess the performance of our tunneling application, which we were initially able to identify as critical requirements.

6.2.1 Latency

Latency is a key performance criterion in network applications and refers to the time delay that occurs when a packet is transmitted from one point in the network to another. To get a comprehensive picture of the latency of our application, we performed special measurements. In doing so, we measured at several critical points in the network - both on the remote node and on the local node. These simultaneous measurements allow us to get a more accurate picture of how latency behaves across the entire transmission link and where any bottlenecks or delays occur.

Measurement Setup

During the evaluation process, we performed several test measurements to develop an optimal benchmark setup. Our tool of choice for these measurements was "ping," which is known not only for its reliability and simplicity but also for its ability to measure latency directly. Through the iterative process of adjustment and re-measurement, we tried different parameters and finally determined, in total 2^{17} packets to be transmitted. This decision was based on the consideration that such a large number of packets would allow us to obtain the most meaningful and reliable results. A larger amount of data reduces the influence of anomalies or random fluctuations, so the final results provide a solid basis for our further analysis and conclusions.

In order to keep both the measurement time short and, in particular, to test the resilience of the system with regard to high packet rates, we placed particular emphasis on a minimum interval between the packets sent. The goal was to keep the time delay between packets as small as possible in order to maximize the system's capacity and

responsiveness. By setting the interval in the ping settings to zero, we continuously flooded the system with requests. This not only offered the advantage of completing the test in the shortest possible time but also gave us valuable insights into how our system performs under extreme conditions and whether it remains stable while doing so.

During our search for the optimal packet interval, we encountered an interesting phenomenon: The ping interval we set seemed to influence the measured latency directly. Specifically, this meant that the ping tool recorded lower latency when the interval between packets was reduced. This may initially appear to be some sort of artifact or anomaly, possibly caused by the use of the virtual test bed. However, this is not a mere peculiarity of our test environment. In fact, this phenomenon is an already known property of the ping tool. An analysis by Google confirmed this behavior and showed that the interval at which ping requests are sent can affect the measured response time [16]. This underlines the importance of thoroughly understanding the tools used and their possible pitfalls if you want to achieve reliable and meaningful measurement results. This phenomenon has encouraged us to use the lowest possible interval times, as these provide more realistic measurement results that more accurately reflect actual network latency.

End to End Latency

At the beginning of our measurement series, we conducted a measurement between the attacker and the remote node without activating the tunneling system. This was necessary to establish a baseline against which subsequent measurements with an active tunneling system can be compared. This reference value provides an important basis for assessing the actual performance of the system.

Based on the results (cf. Figure 6.1), we can observe that the measured ping time has an average latency of 1.08 ms. This measurement represents the time required for only the first hop and the sixth hop, as shown in (cf. Figure 5.4).

Subsequently, the measurement was performed again under the same conditions, but with the difference that the tunneling system was activated. In this scenario, the attacker communicated with the remote node, but all requests were forwarded to the honeypot device. As expected, this configuration resulted in significantly higher latency values, as packets had to go through additional network hops (cf. Figure 5.4) to get from their source to their final destination and back (cf. Figure 6.2).

When we analyze the distributions of the two measurements (cf. Figure 6.3), we find that activating the tunneling system not only increased the average latency but the spread of the latencies also increased noticeably. There are several reasons for this increased variance in the measured values. On the one hand, additional network steps naturally lead to greater variability in the latencies, as each additional link in the chain is a potential source of variation. This can be caused, for example, by varying workloads, delays in the transmission media, or temporary bottlenecks, which partly can be attributed to the use of a virtual testbed in the first place.

On the other hand, the influence of the tunneling system itself may also contribute to the increased variation. These observations underscore the complexity and the many

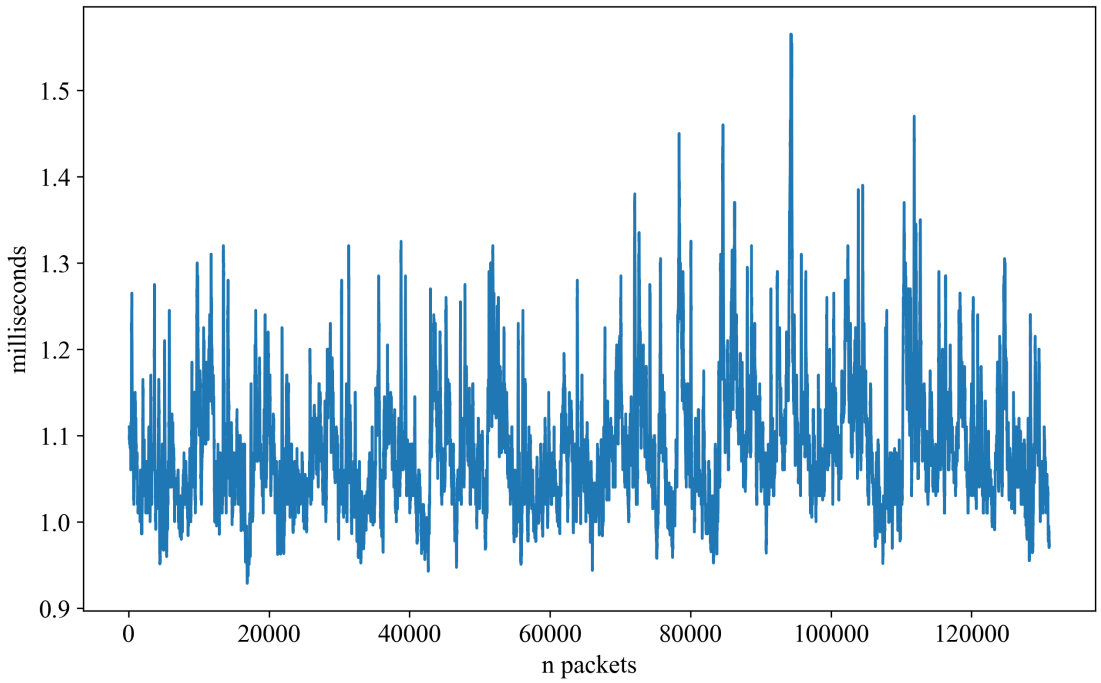

Figure 6.1: Attacker to Remote Node Latency Rolling Median (n=100) (No Tunneling)

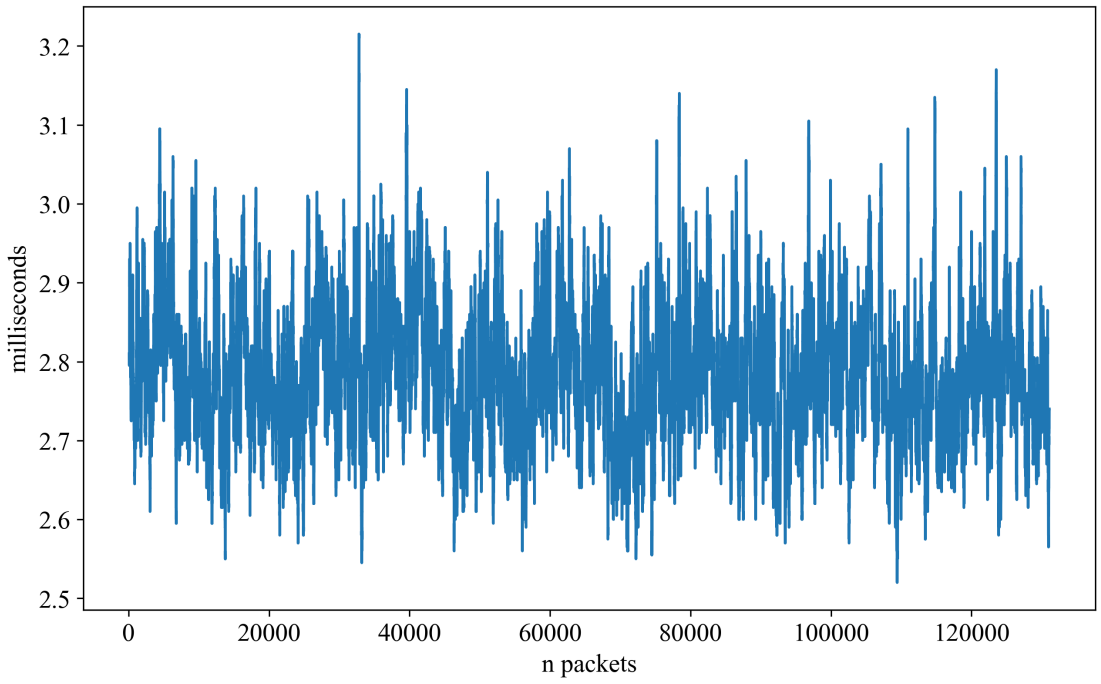

Figure 6.2: End to End Latency Rolling Median (n=100) (Tunneling via Remote Node)

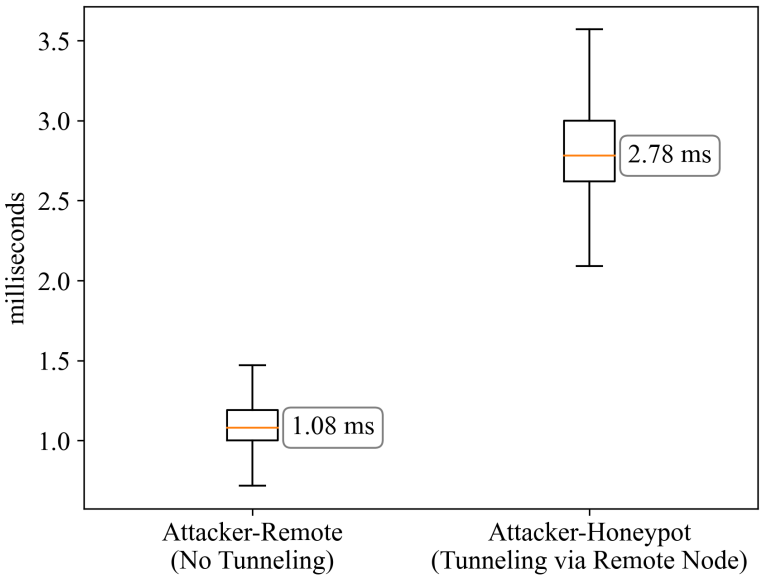

Figure 6.3: End to End Latencies

factors that must be considered when implementing a tunneling system, especially when it comes to performance and reliability.

Estimating the additional time added by the tunneling system, we find that the increase in time is rather close to the measured results. This is a very revealing finding. It implies that the tunneling system itself adds very little additional latency to the system. Thus, most of the additional time can be almost directly attributed to the additional network latency.

Based on the measurements without the tunneling system, where we observed a median latency of 1.08 ms, we can derive a 0.27 ms latency per hop (There are four hops as the packet traverses the router on its path from the attacker to the entry node (cf. Figure 5.4)). The complete packet flow in our application consists of 10 network hops, which leads us to a rough estimate of 2.7 ms. We think that our assumption that every network hop has a similar latency is valid, as they are configured to be similar in our testbed. This results in a difference to the measured value of 80 Microseconds (μs) in which the network traffic is handled by our tunneling system, as well as by the router.

This highlights the efficiency of our tunneling system and shows that its implementation and function are well-optimized to affect latency as little as possible.

Packet Processing Delay

During our performance measurements, we not only examined the overall performance of the system but also analyzed the impact on the individual packets in more detail at the same time using the previously mentioned tool, Xdpdump. Of particular interest here was the timestamp of the incoming and outgoing packets, which is captured with nanosecond

precision. By analyzing these timestamps, we were able to measure the processing time of each packet accurately. This allowed us to get a detailed picture of the performance and potential bottlenecks or delays introduced by the tunneling system. Moreover, due to packet data provided by Xdpdump, we were able to look at the times for encapsulation and decapsulation separately and analyze this specifically for each component of our tunneling system.

This specific measurement using Xdpdump offered us the decisive advantage of looking at the time measurements without the disruptive influence of the network. This allowed us to isolate and accurately analyze the pure processing time of the packets. This provided a clearer and unbiased picture of the tunneling system's efficiency and allowed us to evaluate its performance in a controlled environment. It provides us with a deeper insight into the exact operations and times spent processing the network packets and helps us further understand the efficiency and optimization opportunities of the system.

Additionally, we felt it was important to verify whether the use of Xdpdump itself had an impact on latency measurements. Such considerations are critical, as adding monitoring or measurement tools can affect system performance in some cases. However, after careful analysis and repeated testing, we found no measurable impact of Xdpdump on latency. This confirms the reliability of our measurement data and gives us confidence in the accuracy of our analysis results.

Local Node Processing Delay When we examine the processing time for the local node in more detail, a clear difference becomes visible. Time series diagrams provide valuable insights here (cf. Figure 6.4, 6.5). It can be seen that the average time, as well as the spread of the times for the decapsulation of the packets, is considerably higher than for the encapsulation. This suggests that the process of decapsulation is more complex or experiences more internal system delays than encapsulation.

The observed correlation in processing times can be attributed to the specific operation of our implementation. For each packet processed, a data structure is consulted that holds the mapping from the attacker's IP address to the remote node's IP address. For each packet, this structure is checked and updated if necessary. This process involves a write access, which typically takes more time than a read-only access. This makes the decapsulation that involves this allocation check and potential update more time-intensive. This explains the increased scatter and generally longer processing times for decapsulation compared to encapsulation.

The difference between encapsulation and decapsulation becomes particularly evident when the respective distributions are placed next to each other and analyzed. While the encapsulation shows relatively constant values, the distribution of the decapsulation times shows a wider deviation, which clearly illustrates the previously described discrepancy (cf. Figure 6.6).

This observed difference is particularly relevant as it highlights potential areas of optimization in our system and underscores the importance of separate measurements for different components and processes within the tunneling system.

The processing times we observed in our analysis are well below the orders of magnitude

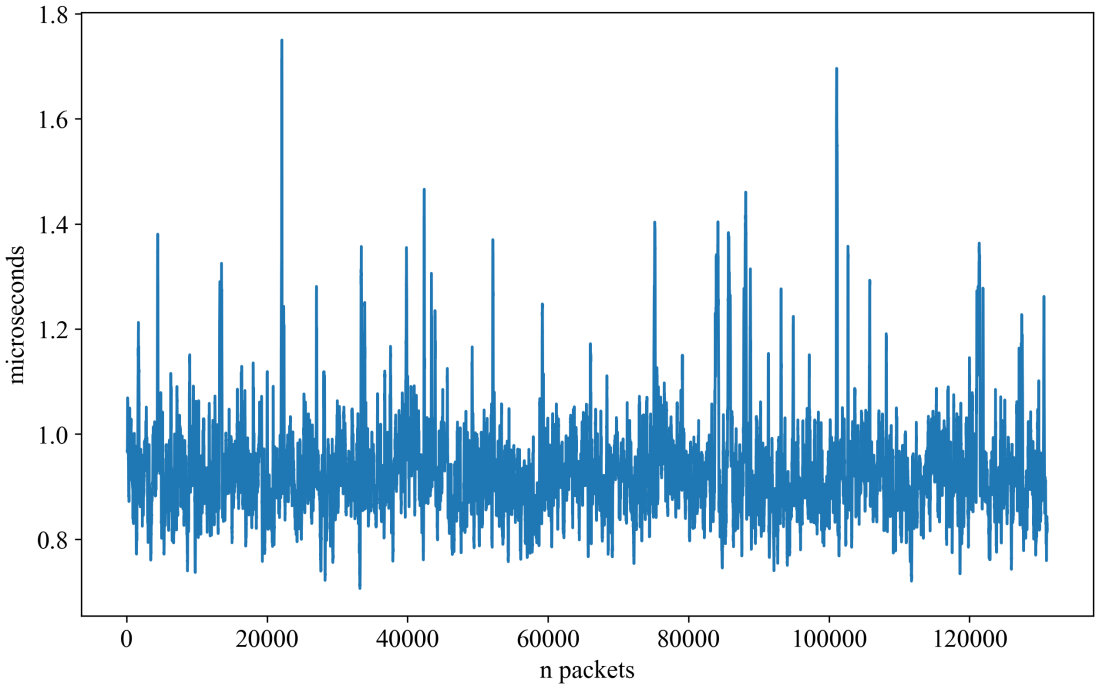

Figure 6.4: Local Node Packet Encapsulation Times Rolling Median (n=100)

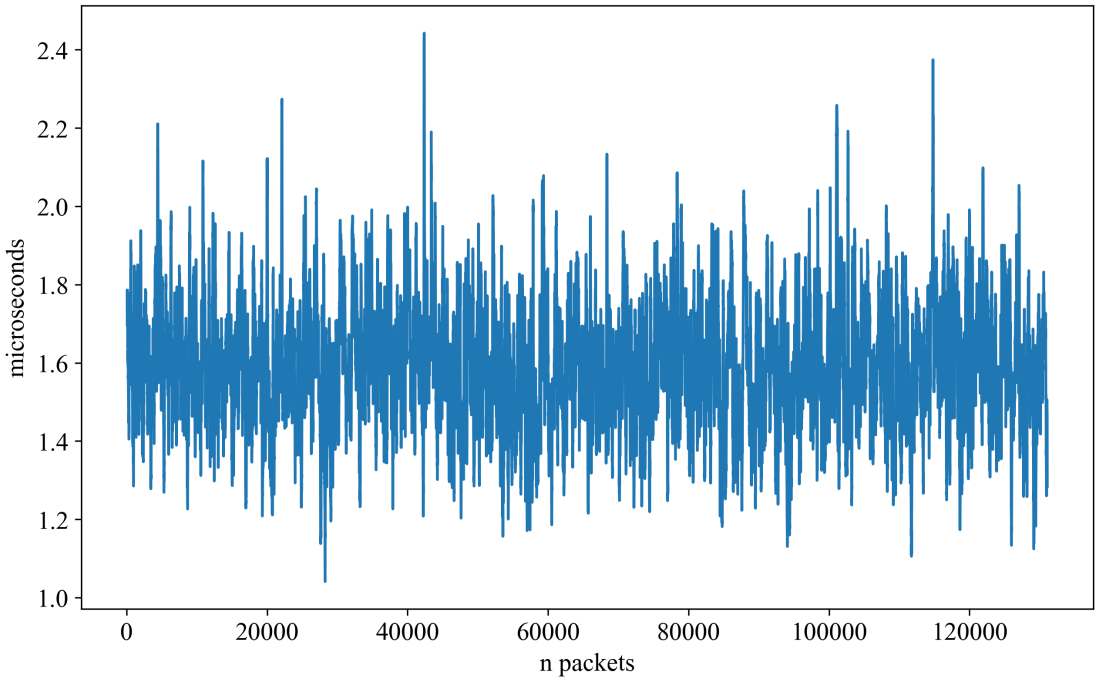

Figure 6.5: Local Node Packet Decapsulation Times Rolling Median (n=100)

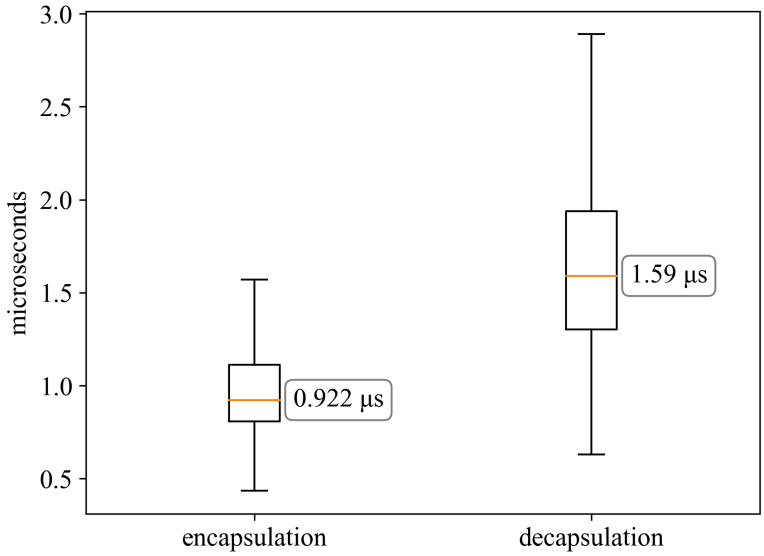

Figure 6.6: Local Node Packet Processing Times Distributions

typically expected for network latency, as estimated previously. From this observation, it is clear that the processing time of the local node does not have a significant impact on the overall context of network latency. This illustrates that despite the additional processing steps introduced by our tunneling system, system performance remains largely unaffected.

Remote Node Processing Delay If we now look at the time series diagrams of the remote node in comparison, an interesting pattern becomes visible. Unlike the local node, where encapsulation and decapsulation had different processing times, the remote node shows remarkable consistency in its processes. Both encapsulation and decapsulation show very similar processing times (cf. Figure 6.7, 6.8). Furthermore, the dispersion of these times is also consistent, indicating that the operations on the Remote Node are very stable and there is little variation (cf. 6.9). This can be attributed to the fact that the tasks performed on the remote node are less complex compared to the local node and that the data structures and algorithms used are well-optimized. In any case, it shows that the remote node is very reliable in its performance and performs its tasks at an expected and consistent speed.

The processing times observed on the remote node are similar to the values we measured for encapsulation on the local node. This is to be expected since the algorithms on both nodes, local and remote, are fundamentally very similar in structure. The most significant difference there is in the computation of the mapping mechanism of the local node. The fact that the times are so close and the only significant difference in the algorithms is this mapping mechanism reinforces the assumption that the additional processing time on the local node is indeed caused by this mapping operation. This finding supports the

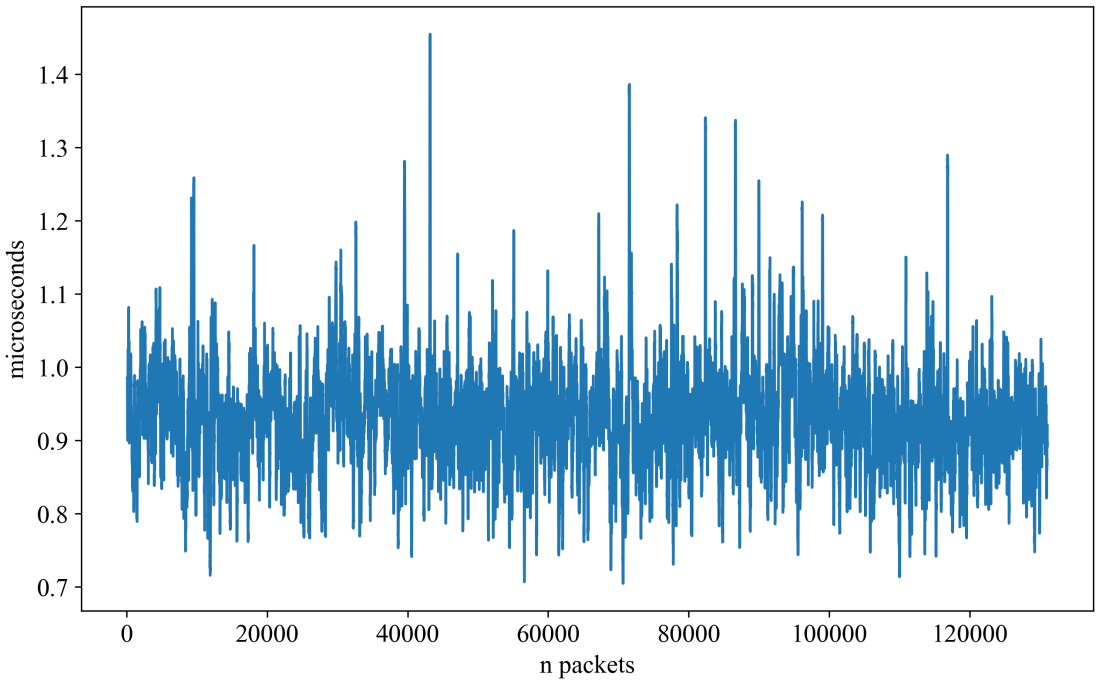

Figure 6.7: Remote Node Packet Encapsulation Times Rolling Median (n=100)

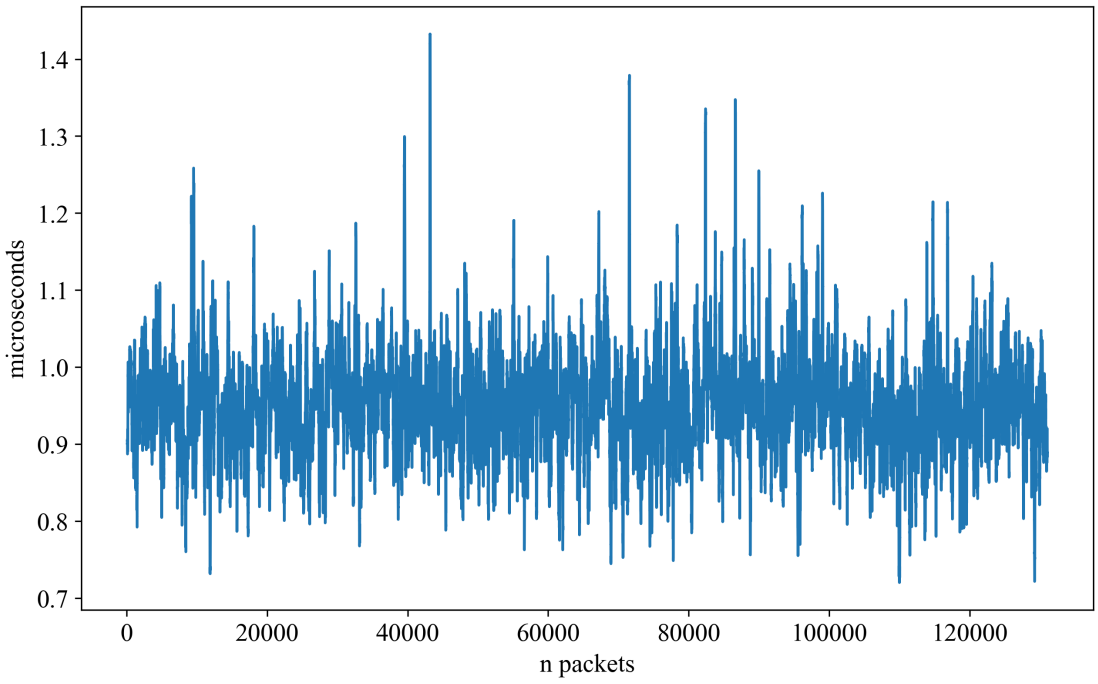

Figure 6.8: Remote Node Packet Decapsulation Times Rolling Median (n=100)

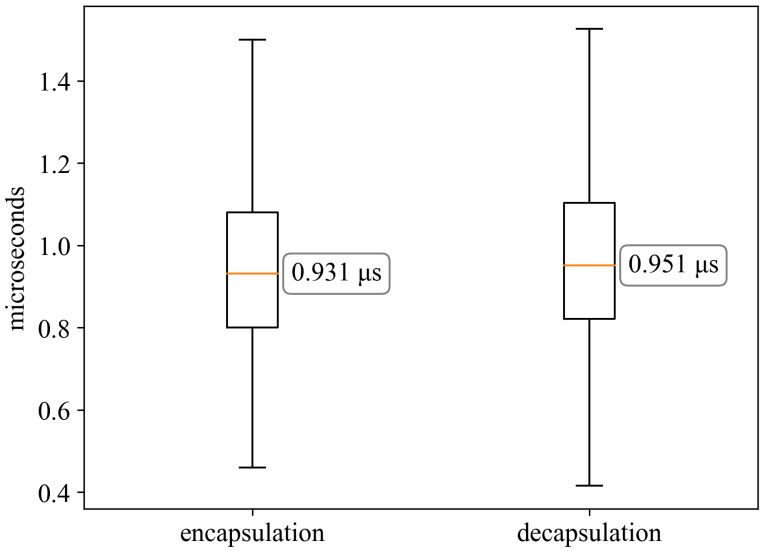

Figure 6.9: Remote Node Packet Processing Times Distributions

hypothesis that the mapping - especially the writing and updating of the mappings - has a significant impact on the performance of the local node.

Ultimately, it can be concluded from the latency analysis that although we have identified a potential optimization opportunity with regard to the mapping mechanism, this only plays a subordinate role in the overall application scenario. Especially when considering the whole scenario in the context of a cloud environment, where latency tends to be higher anyway due to the inherent network structure, it becomes clear that the additional processing times of our tunneling system hardly matter. It is, therefore, essential to focus on the big picture and recognize that, despite the minor potential for optimization, the system already works very efficiently in its current form.

6.2.2 Jitter

Jitter represents the next relevant performance criterion in our analysis. While latency describes the average time it takes a packet to get from one point to another, jitter refers to the variability of these times. In other words, jitter measures the fluctuations or inconsistencies in packet transmission time. In an ideal network, latency would always be constant and jitter accordingly zero. In real networks, however, especially in complex applications and cloud scenarios, these transmission times can vary. Such variations can be caused by a variety of factors, ranging from network load or network configuration, such as incorrectly set buffer sizes, to hardware or software problems. A high jitter value can lead to a degradation of the communication quality, especially in real-time applications. Therefore, it is not only important to look at the latency itself but also the consistency of this latency, which manifests itself in the form of jitter.

Local Node	Encapsulation Time (μs)	Decapsulation Time (μs)
mean	2.0403	2.6599
std	42.2930	29.5332
min	0.4330	0.6300
50%	0.9220	1.5900
90%	1.3839	2.3420
95%	1.6120	2.6834
99%	18.9123	20.9413
max	4,180.7370	4,201.8000

Table 6.1: Local Node Packet Handling Times

Processing Delay Jitter

In our test scenario, we have two approaches to determine the jitter.

On the one hand, we can measure the end-to-end jitter. This gives us information about the total variance of the transmission times from a starting point to an endpoint. Here, we look at all factors that could play a role during the transmission of data packets - from source to destination, including all network elements in between.

Second, we focus on the jitter introduced by our system. This focuses specifically on the variations in processing times within our tunneling system. This is a more detailed investigation that gives us direct indications of how consistent and reliable our application is in handling data packets, regardless of other network elements.

Local Node The analysis processing times of the local node times reveal some significant findings regarding jitter (cf. Table 6.1 [1]). As anticipated by the time series plots presented earlier (cf. Figure 6.4, 6.5), significant jitter is evident within the local node.

When examining the processing times of the local node, several aspects points stand out. Although the average processing times for encapsulation and decapsulation are in the microsecond range, the standard deviations are strikingly high in relation to these average values. This illustrates a volatile and variable processing pattern.

Particularly noteworthy are the extreme maximum values, which are in the thousands of Microseconds. Such high values can lead to significant jitter in practice, especially in real-time applications, and they highlight the need to check system stability and make potential optimizations.

The medians, represented by the 50th percentile values, are closer to the averages, indicating that most packets are processed within an acceptable amount of time. However, the significant deviations in the upper percentage ranges, especially the values near the 99th percentile, indicate that significant delays occasionally occur.

[1] The data shown in the table extracts critical figures from the boxplot shown above (cf. Figure 6.6) and is crucial for our jitter analysis. These figures provide a precise representation of the variability in the latency distribution and allow us to understand and interpret the jitter dynamics of our system more accurately.

The strong fluctuations in processing time are further illustrated by the significant difference between the median and mean values. While the median for encapsulation is 0.9220 μs and for decapsulation 1.5900 μs, the mean values are 2.0403 μs and 2.6599 μs, respectively. This highlights the presence of outliers that significantly affect the average value.

In summary, the analysis shows that although the local node has fast processing times on average, the local node exhibits potentially unsteady behavior and can cause significant jitter, which must be taken into account, especially for time-critical applications.

Remote Node When analyzing the processing times of the remote node with regard to the jitter, the data shows evident stability in the processing times (cf. Table 6.2 [2]). Compared to the local node, the fluctuations in the remote node are remarkably lower. Specifically, the extreme values for encapsulation are lower by a factor of 20 and for decapsulation by a factor of 12 compared to the local node. While both encapsulation and decapsulation times show a standard deviation significantly exceeding the median value, this is still within an acceptable range relative to the average value, especially considering the time unit. This is underlined by comparing the 99th percentile values with the maximum values. Although there are some extreme outliers, these are rare in the overall picture and do not contribute significantly to the overall jitter. Of particular note is the fact that the median times (50th percentile values) for both processes are very close to the average, indicating consistent performance overall. Therefore, this data suggests that the remote node introduces minimal jitter despite minor fluctuations and thus performs very well in terms of stability and reliability.

It also shows that the tunneling system as a whole benefits from the stability and efficiency of the remote node.

Remote Node	**Encapsulation Time (μs)**	**Decapsulation Time (μs)**
mean	1.0931	1.1215
std	1.7935	2.1369
min	0.4600	0.3930
50%	0.9310	0.9510
90%	1.2900	1.3190
95%	1.4480	1.4780
99%	1.9553	1.9760
max	195.6730	331.7320

Table 6.2: Remote Node Packet Handling Times

End to End Jitter

Nevertheless, it is of central importance not only to consider the jitter of the individual components, i.e., the remote and the local node but to focus on that of the overall system.

[2]The data in this table corresponds to the following boxplot (cf. Figure 6.9).

End to End Latency	Attacker-Remote (ms) (No Tunneling)	Attacker-Honeypot (ms) (Tunneling via Remote)
mean	1.1580	3.0516
std	1.3598	1.2318
min	0.6960	2.0900
50%	1.0800	2.7800
90%	1.3400	3.3500
95%	1.5100	4.9300
99%	2.3700	7.8500
max	192.0000	20.7000

Table 6.3: End-to-End Latencies

After all, the time units considered in the analysis are smaller by a factor of 1,000 than the latencies that usually occur in real networks. Therefore, the individual jitter of a single component can easily be ignored in an overall system, which already has higher latencies by nature, and its effect on the entire system is less significant.

The detailed analysis of the end-to-end latencies provides interesting insights into the jitter within our system (cf. Table 6.3 [3]). In a direct comparison of the times without and with tunneling, it is noticeable that the average latencies differ significantly, as previously highlighted. For a direct connection between the attacker and the remote node without tunneling, the median latency is 1.08 ms. The standard deviation of 1.3598 ms and the significant maximum value of 192 ms make it clear that the latencies exhibit considerable fluctuations. Particularly striking here is the significant difference between the 95th percentile (1.51 ms) and the 99th percentile (2.37 ms), which indicates a significant jitter in the highest measured values.

However, if we look at the tunneling connection between the attacker and the honeypot, the median value is significantly higher at 2.78 ms, which scales linearly to the number of network hops, as argued previously. Despite the additional processing time due to tunneling, the associated standard deviation, i.e., the jitter, of 1.2318 ms, is slightly lower than for the direct connection. Here, one could argue that the jitter of the individual system components, such as the local and remote nodes, is no longer as noticeable at the end-to-end level. In fact, one could even argue that the jitter decreases slightly when tunneling is used, even though this difference cannot be considered significant. A possible reason for the slightly lower jitter in our tunneling system could be the use of XDP in our implementation. Using XDP bypasses any queuing mechanisms of the operating system, which leads to more consistent and predictable packet processing and can thus reduce fluctuations in latency.

What is noticeable with tunneling, however, is a wider spread of the observed times. This suggests the presence of less frequent but more significant jitter. This observation fits with the expectation that jitter increases with the number of systems or network

[3] The data in this table corresponds to the following boxplot (cf. Figure 6.3)

hops traversed. Each additional hop or system through which data packets are forwarded introduces some variability in the delay, which can lead to increased jitter.

In summary, although tunneling increases average latency, the associated jitter does not necessarily increase in proportion due to the presence of the tunneling system itself. In fact, it appears that the additional processing steps provided by tunneling may help to dampen the variability in latency somewhat. Nevertheless, the wider spread of times with tunneling shows that overall jitter potentially increases, primarily due to the additional network hops.

Finally, it should be noted that the influence of jitter strongly depends on the type of honeypot device used. Therefore, the influence of jitter on the quality of service cannot be determined across the board.

6.2.3 Throughput

Having examined jitter in detail as a critical performance characteristic of our system, we now turn to another key aspect of network performance: throughput or bandwidth. Throughput indicates how much data can be transmitted over a network in a given amount of time and is thus a direct indicator of the performance of a system or connection. This parameter is especially critical in applications where large amounts of data must be transferred in a short period of time, which we defined as one of the critical requirements. In this section, we will examine the average throughput of our tunneling system and analyze how different load scenarios affect it. In doing so, it is essential to understand whether and to what extent our implementation affects the data traffic and what impact this has on the overall performance of the network.

Measurement Setup

In order to thoroughly test the bandwidth of our system, we decided to use the proven tool iperf3[4], more precisely, version 3.9. Iperf3 is a widely used tool for measuring the maximum TCP and UDP bandwidth and provides precise and reproducible measurement results.

Our experimental setup was carefully planned to simulate the realistic operating conditions of the system. Both the local node and the remote node were activated and set to a stable operating state. The honeypot, which served as the target server, was configured to run iperf3 in server mode. This means that it was ready to accept incoming connection requests and process traffic. However, all individual measurement operations are initiated by the attacker, both for incoming traffic and for outgoing traffic. It is necessary to do so because, in our system, after a reboot, there is no possibility for the honeypot device to address a client on the public internet by itself unless there was incoming traffic before.

However, this does not affect the measurement results since it is possible to determine the direction of the benchmark with iperf3.

[4] `https://iperf.fr/` (Accessed October 2023)

Determining the measurement interval Selecting the right measurement window is critical to the validity and meaningfulness of the data obtained. Measurements that are too short could miss longer-term network fluctuations, while measurements that are too long unnecessarily drag out the test and do not necessarily provide more meaningful results.

Initially, we performed trial measurements that already showed high bandwidth values to determine the optimal benchmark setup. Nevertheless, we were aware that to assess long-term performance, it is essential to consider all buffers in the network. A common concept for estimating this buffer requirement is the Bandwidth Delay Product (BDP). It provides a theoretical value of the volume of data that is "in motion" on the network while data is being transmitted.

Two main components are required to calculate the BDP: the latency (in our case, the end-to-end delay) and the bandwidth. In our scenario, the average end-to-end latency was 2.78 ms as explained earlier. With a maximum expected[5] bandwidth of 250 Megabit per Second (Mbps), the BDP gives a value of 84.84 Kilobyte (KB). This shows how much data could theoretically travel between sender and receiver without any point in the network being saturated, which in turn gives an estimate of how large the packet buffers need to be at each client.

However, the analysis does not end here. The entire network consists of multiple hops, each with its own buffer memory. Therefore, we multiplied the BDP value by the number of hops, resulting in a minimum data volume to be transmitted of 848.4 KB.

We found that the amount of data required to saturate all buffers in the network fully can be transmitted in a very short period of time. In fact, a 30-second measurement window in our scenario was already considered a more than long enough observation period to saturate the network and provide robust data. Therefore, we initially performed and analyzed all measurements in this shorter time period. However, to verify the consistency of our results and to ensure that our observations were not just subject to temporary fluctuations, we extended the measurement interval to 600 seconds. This effectively corresponds to 20 times the duration of the original measurement and thus provides a more complete picture of the performance and stability of our system over a more extended period of time while increasing the validity. [6]

Through this analysis, we concluded that using a 600-second measurement window was more than sufficient not only to ensure that all buffers in our network were utilized entirely but also to give us a reliable estimate of network performance in long-term operation as well as to reduce any fluctuations in performance induced by the virtual testbed. This approach allowed us to strike a balance between test duration and accuracy, making the

[5] The expected bandwidth was determined by preparatory probing measurements taken in preliminary to the comprehensive tests.

[6] We want to highlight that the results of the 30-second interval measurements were similar to the 600-second interval measurements. In addition, to ensure the integrity and validity of our results, we compared the data from several 600-second measurements with the data presented here. These additional measurements provide consistent results that confirm the findings shown in this thesis. Although we do not explicitly present these additional data sets, they serve as an important point of comparison to support the reliability of our conclusions.

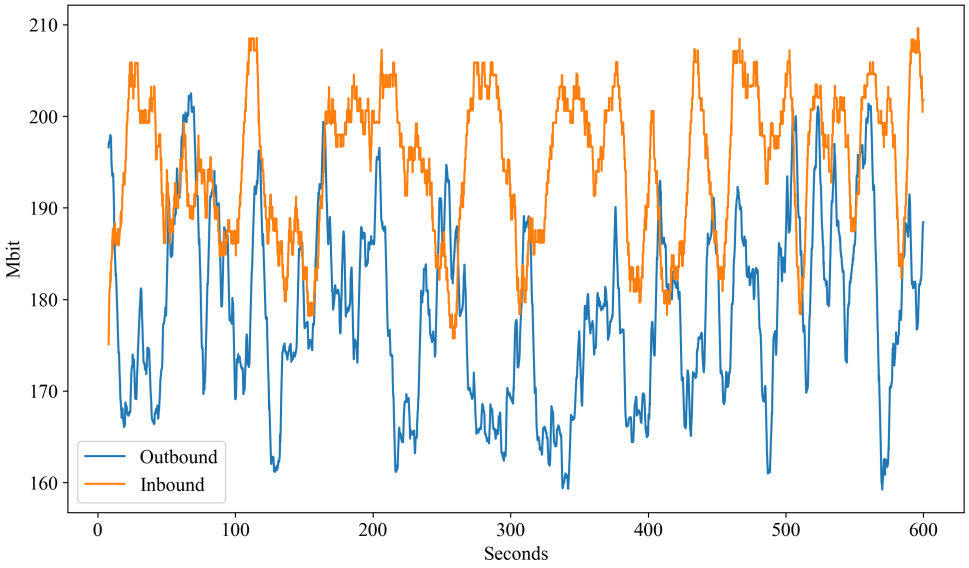

Figure 6.10: TCP Inbound and Outbound Throughput Rolling Mean (n=80)

results obtained both accurate and relevant to real-world applications.

End to End Throughput

Our study mainly aimed at measuring the entire end-to-end bandwidth. The reason for this approach is our belief that this specific measurement is most representative of our scenario. Although different components within a network might have their own specific bandwidth profiles, it is the total throughput rate when all components interact that is most relevant in practical applications.

One Way TCP Bandwidth In our testbed, the selected network topology already represents a minimal topology. Any additional complexity or variation in real network environments would still follow similar basic principles. In this context, the specific bandwidth of a single component or link plays a minor role compared to the overall system throughput. The latter gives us a holistic view of how efficiently data flows through our system and thus provides a definitive indicator of system performance in a real-world deployment scenario. Therefore, we consider measuring the total end-to-end bandwidth to be the most meaningful way to evaluate network performance in our given context.

The figure (cf. Figure 6.10) shows the progression of the measured bandwidth for both incoming and outgoing TCP traffic. It is important to emphasize that incoming and outgoing traffic were captured in separate measurements. One particularly noteworthy detail is that the bandwidth of outgoing traffic is significantly lower than that of incoming traffic.

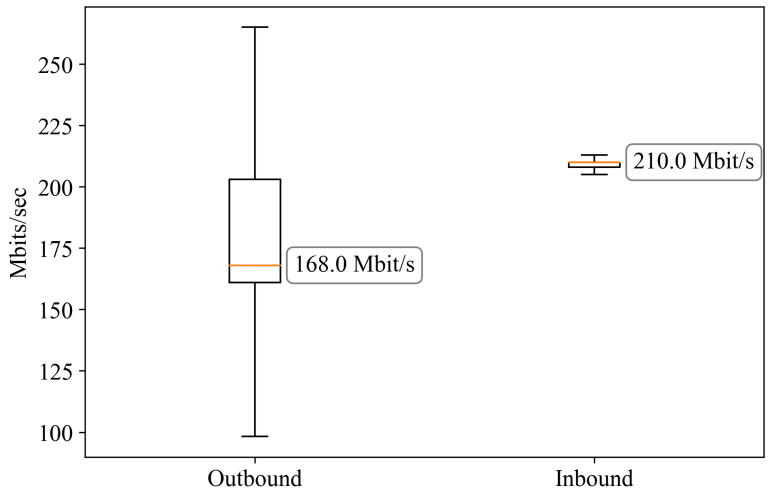

Figure 6.11: TCP Inbound and Outbound Throughput Distributions

A particularly striking aspect of our analysis is that the outgoing traffic, in particular, is characterized by strong fluctuations, with the observed value range extending from 51.7 Mbps to 317 Mbps. In the previous latency analysis, it already became clear that the incoming traffic on the local node, especially decapsulation, was identified as particularly computationally intensive and was subject to strong fluctuations. From this, one might initially assume that it is precisely this incoming traffic that also shows the most substantial bandwidth fluctuations. Surprisingly, however, the opposite is true.

We suspect that the cause of this discrepancy is not to be found in our system but in iperf3 itself. The iperf3 software always keeps a parallel control TCP channel open during the measurement. This is intended to control and monitor the measurement. However, it is possible that this control channel causes marginal interference with the main measurement channel, which could contribute to the observed variations in outgoing traffic.

As already suspected, if we now examine the distribution of the bandwidth values (cf. Figure 6.11), we can clearly see that there are strong differences between incoming and outgoing traffic. It clearly shows that outbound traffic not only has a lower average bandwidth but is also characterized by significant fluctuations. These fluctuations are due to various factors, such as network latencies, buffer capacities, or even specific traffic management algorithms on the server and client side. It highlights the need to monitor bandwidth in both directions for a complete picture of network performance.

Furthermore, the analysis of traffic behavior reveals the characteristic sawtooth pattern (cf. Figure 6.10), especially for incoming data traffic. This pattern arises from the way the congestion control mechanism works. Here, the transmission rate increases until congestion occurs in the network. Then, the mechanism intervenes and reduces the rate and then tries to increase it again. This continuous cycle creates the aforementioned up-and-down pattern, which resembles a sawtooth. It is a powerful illustration of how

protocols such as TCP strive to strike the optimal balance between high throughput rate and network stability. Observing this pattern also reinforces our previous decision to choose a 600-second measurement window, as it illustrates how essential it is to give enough time for such dynamic adjustments in network behavior.

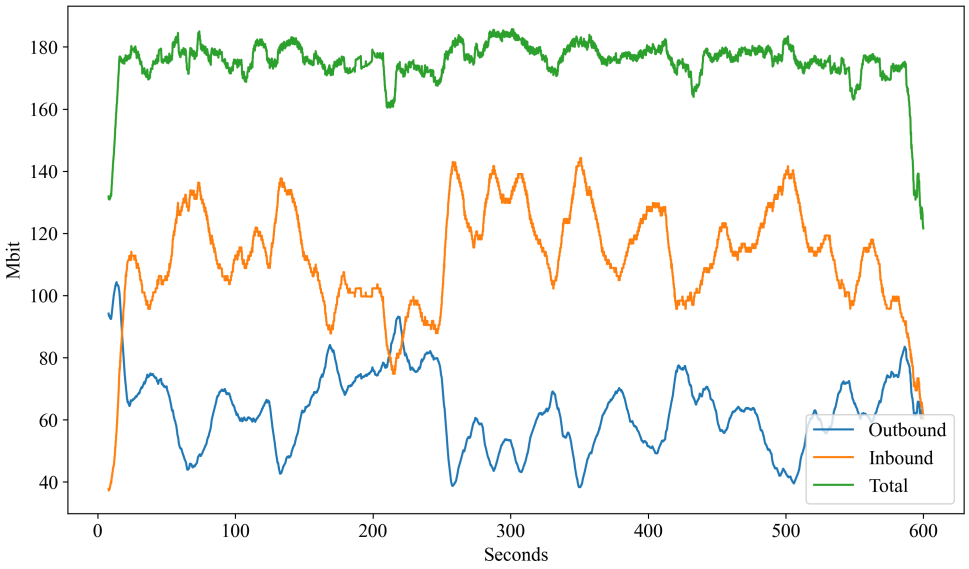

Figure 6.12: TCP Bidirectional Throughput Rolling Mean (n=80)

Two Way TCP Bandwidth After performing the measurements individually for incoming and outgoing traffic in the first run, we repeated the process, this time measuring both traffic directions simultaneously (cf. Figure 6.12). As expected, the results from this measurement showed a significant reduction in bandwidth for both incoming and outgoing traffic. One of the main reasons for this is that two parallel data streams are now being transmitted over the network simultaneously and thus have to share the available bandwidth.

However, it is interesting to note that when the bandwidths of both directions are added together, the total bandwidth is consistent with the values obtained in the previous measurements performed separately. This confirms that the bandwidth degradation in the simultaneous measurements is mainly due to the split between the two streams.

In terms of the distribution of the bandwidths, a similar pattern as in the previous measurements can be seen (cf 6.13). It can be seen that the structure and behavior of the network traffic remain consistent under different conditions.

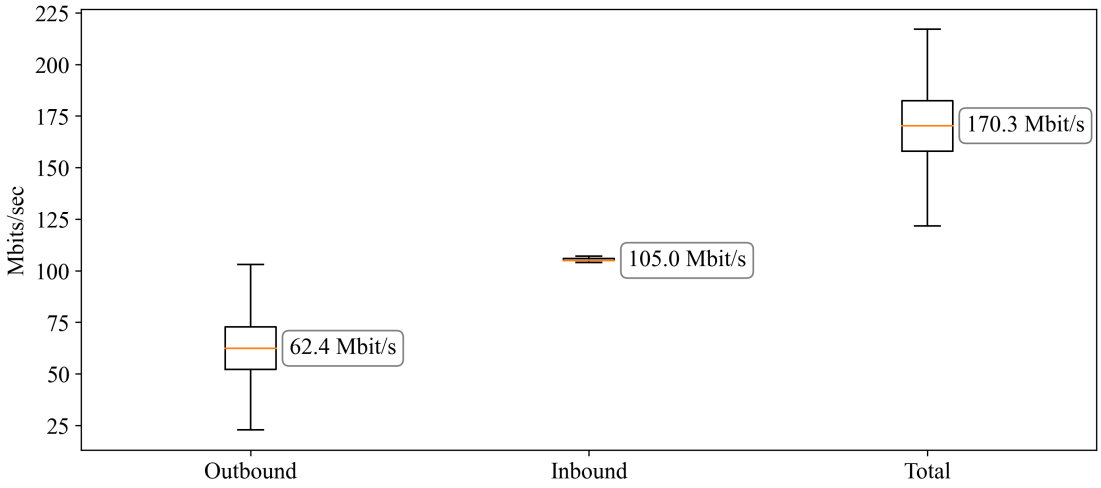

Figure 6.13: TCP Bidirectional Throughput Distributions

UDP Bandwidth Following our TCP bandwidth tests, we additionally performed bandwidth tests with UDP. As expected, the results we obtained here were significantly lower than the values we had previously determined with TCP. It was noteworthy that the bandwidth of the outgoing traffic was again lower than that of the incoming traffic (cf. Figure 6.14).

Some reasons can be given for the lower UDP bandwidth values compared to TCP:

- Packet size: When using iperf3 for UDP testing, UDP packets are generally smaller than TCP packets. This results in a more inefficient use of the available bandwidth since more overhead is incurred due to additional packet headers and the associated increase in the packet rate.
- Control mechanism: Unlike TCP, which has implemented mechanisms such as TCP Congestion Control to control data flows and avoid network congestion, UDP does not provide such mechanisms. Therefore, it is especially important to carefully adjust the UDP data rate and packet size to the specific network conditions. However, we decided to work with the default settings to get a basic impression of UDP performance in our test network.

A closer look at the distribution of UDP bandwidths reveals a pattern that differs from that in the TCP tests (cf. Figure 6.15). In contrast to our TCP observations, the variation in transfer rate of outbound traffic in UDP is smaller than that of inbound traffic and also generally more constant compared to TCP. This could be due to the special nature of UDP and the lack of congestion control, which constantly influences the transfer rate, leading to a more constant output in UDP if no active adjustments are made.

As a result, some noticeable spikes in the transmission rate were observed when analyzing the outgoing traffic (cf. Figure 6.14). We suspect that these spikes are due to

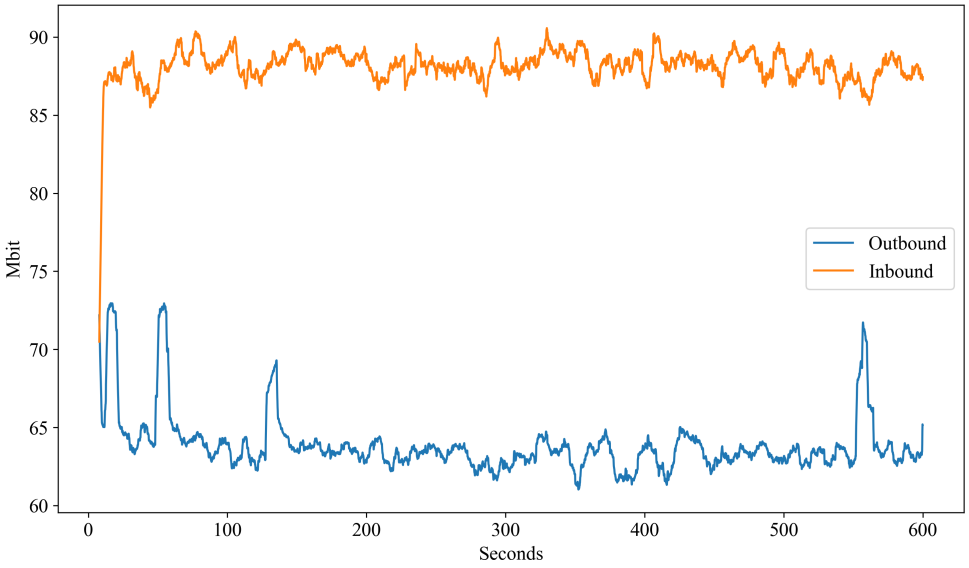

Figure 6.14: UDP Inbound and Outbound Throughput Rolling Mean (n=80)

specific aspects of our testbed setup, which contains some variables that are difficult for us to control and are most likely related to the virtual nature of our testbed.

To complete our series of tests, we also considered performing bidirectional tests with UDP. However, after initial testing, we decided against this approach. The reason for this lies in the nature of UDP itself: Since UDP does not have a built-in congestion mechanism, in our bidirectional tests, we experienced a phenomenon where one random traffic stream always dominated and took up almost all of the available bandwidth. The other stream, on the other hand, suffered significant losses, up to situations where no packets were allowed through at all. These results showed us the limitations of UDP in scenarios with competing data streams and confirmed our decision to focus on unidirectional measurements in order to obtain reliable and interpretable results.

In conclusion, regarding the bandwidth measurements, the bandwidths achieved in our tests should be sufficient in most typical honeypot scenarios. It is important to emphasize that honeypots are usually not designed to handle extremely high traffic loads but rather specialize in detecting and analyzing potential attackers and their methods. In this context, the bandwidth values we measured provide a solid basis for ensuring that the honeypot can work effectively while meeting the requirements of real network environments. It can thus be seen that our system is well equipped to stand up to real-world use, not only in terms of latency but also in terms of bandwidth.

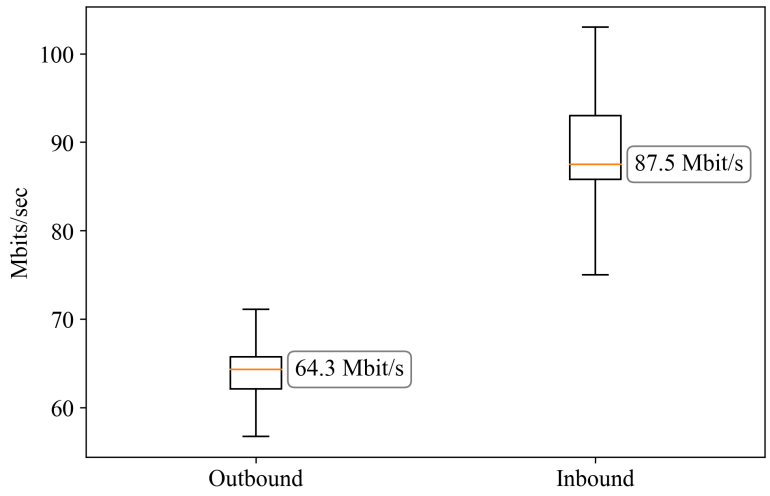

Figure 6.15: UDP Inbound and Outbound Throughput Distributions

6.2.4 Paket Loss Rate

The analysis of the packet loss rate is a crucial criterion in network evaluation since it has a direct influence on the quality and reliability of data transmission. In our evaluation, we found that our tunneling system did not cause any significant packet loss. This was especially confirmed by the ping flooding performed as part of the latency measurement, where responses returned consistently and without significant delay or loss.

However, it is essential to emphasize that packet loss can be observed for both TCP and UDP connections. For TCP, this is because this protocol always tries to use the maximum possible bandwidth and thus reaches the limits of the network, which can lead to losses. With UDP, on the other hand, the transmission must always be precisely matched to the current network conditions in order to function optimally. Due to the lack of a control mechanism with UDP, packet losses can quickly occur here if the data rate or packet size is not adjusted correctly.

Interestingly, we believe that the packet loss observed in our tests is not directly due to our implementation. Instead, we suspect the main reason to be the central node of our virtual network topology - the router node. This effectively has to deal with a packet rate that is two times higher than the measured values. This is because the router has to handle both incoming and outgoing traffic packets, as well as the additional packets caused by tunneling. Thus, the router represents a potential bottleneck that can reach its limits at high data rates, especially in our testbed.

In conclusion, our analysis shows that although packet loss may occur in some situations, in most cases, it is not due to the implementation of our tunneling system. Instead, external factors and network hardware are crucial in ensuring consistent and reliable data transmission.

6.2.5 Profiling

Profiling software, especially at low levels, is often a challenging task. In the case of XDP programs, this is even more difficult. The main reason for this is the nature of XDP itself: It is a relatively new technology that operates at the network driver level and intervenes directly in the kernel data path. While this offers excellent performance benefits, it makes conventional profiling an extremely complex affair.

Detailed analysis of XDP program behavior would require the development or customization of specialized tools and techniques, which would require significant effort in terms of time and resources. Traditional profiling tools are often unable to analyze such deep levels of the system efficiently, and even if they could, interpreting the results would be difficult without specific expertise [26].

It is worth noting that despite the absence of deeper profiling analysis, we have already been able to extract such significant data about our implementation in the context of latency analysis. Compared to other more commonly observed network latencies, our measured times are marginal. This observation supports the assumption that our XDP program operates efficiently and allows network packets to be processed with minimal delay.

In summary, although in-depth profiling analysis could undoubtedly provide further details and insights, the measurements and analysis performed so far are sufficient to draw basic and informative conclusions about the performance of our system. In practice, this means that our system is not only capable of efficiently processing network packets but also that it is competitive and high-performing in the context of real network applications.

6.3 Quality tests

The quality of tunneling is a critical element in evaluating our system, especially in a scenario where the stealth and invisibility of the remote node are of paramount importance. In the context of honeypots, it is essential that attackers cannot tell the difference between a device tunneled through our system and a physical, directly connected device. This ensures that the data collected about potential attackers and their methods is valid and representative. In this section, we will therefore investigate and evaluate the extent to which our tunneling system successfully hides the remote node and whether and what differences can be detected from an attacker's perspective between the tunneled remote node and the actual honeypot device.

To evaluate the quality of our tunneling system in terms of remote node disguise, we have now shifted our attention to fingerprinting tests in addition to the analyses we have performed so far - in particular, stress tests and latency and timing analyses. Fingerprinting allows us to capture detailed information about the system characteristics and behavior of the remote node. This is critical to identify potential discrepancies or detection points that attackers could use to detect the tunnel's presence. These tests allow us to verify the credibility and efficiency of our system in real-world deployments and ensure that the tunneled node appears as authentic as possible.

6.3.1 Measurement setup

To test the quality and efficiency of our tunneling system in terms of its ability to obfuscate the remote node, we constructed a special measurement setup. Our investigation focused on fingerprinting the systems with the common tool Nmap[7], which attackers commonly use to obtain detailed system information.

1. Setting up the honeypot: First, we installed and configured a web server on our honeypot. This web server was used to host a minimal website that served as the target for our tests.

2. Ground truth generation with Nmap: With the tunneling system disabled, we performed Nmap scans on the honeypot from the local node. The goal was to obtain what is known as a ground truth value, which is an unbiased result that shows us the actual characteristics and current state of the honeypot. This data is essential for the subsequent analysis, as it serves as a reference and basis for comparison for the tests with the active tunneling system.

3. Fingerprinting with active tunneling system: After we obtained the ground truth value, we activated the tunneling system. We did not direct the subsequent Nmap scans to the honeypot but to the entrance of the tunneling system. In this way, we wanted to find out if and how the results change when traffic is routed through our tunneling system.

By comparing the Nmap results without and with the tunneling system active, we were able to examine in detail what differences, if any, are detectable from a potential attacker's perspective between direct interaction with the honeypot and interaction via our tunneling system. In doing so, our primary goal was to identify possible deviations or detection points that could provide clues to the existence of the tunneling system.

6.3.2 Methodology for the study of differences using Nmap

To identify potential differences between direct interaction with the honeypot and interaction via our tunneling system, we used different scanning methods. These different approaches were intended to ensure that we had a comprehensive picture of the potential impact of our tunneling system on network traffic. The following list describes the specific measurement methods we used:

- TCP Stealth Scan (SYN Scan): This method takes advantage of the fact that many systems do not generate log entries for a SYN-only request (the first step of a TCP handshake). In a SYN scan, nmap sends a SYN request to the destination port and waits for the response. If a SYN-ACK response is returned, the port is detected as open. It is important to note that the full TCP handshake is never completed in order to remain undetected. We used this method to scan all available ports on the honeypot.

[7] `https://nmap.org/` (Accessed October 2023)

- TCP Connect Scan: In contrast to the stealth scan, the full TCP handshake is performed here. This means that after a successful SYN-ACK, an ACK response is sent from Nmap by the target to complete the handshake. After that, the connection is terminated immediately. This scan type is less unobtrusive as it makes full connections to every port checked. Again, we scanned all available ports on the honeypot.

- UDP Port Scan: TCP and UDP are the two main transport protocols on the Internet. While TCP is connection-based and therefore easier to scan, UDP is connectionless and therefore can be more difficult to scan. In a UDP scan, Nmap sends an empty UDP packet to each destination port. For closed ports, the destination should ideally respond with a "port unreachable" packet. Open ports usually do not respond, which makes them harder to detect. In our investigation, we focused on the most common UDP ports to keep the investigation within a reasonable time frame.

In addition to the port scanning methods outlined above, we also used Nmap for operating system detection. This function in Nmap is based on certain characteristic patterns in network traffic that allow conclusions to be drawn about the underlying operating system. Here, specific packet combinations are sent to the destination, and by analyzing the responses, Nmap can provide an estimate of the operating system in use.

By combining these scanning methods, we wanted to ensure that our tunneling system was examined from different perspectives and that any anomalies or differences from direct access to the honeypot could be detected.

In addition to the Nmap scans, we performed close monitoring of network traffic on the honeypot device in parallel. However, the goal of this monitoring was not to identify direct differences in the scan results. Rather, we were interested in developing a deeper understanding of how network traffic on the honeypot device responded in real-time, especially given the traffic passing through our tunneling system.

During the Nmap scans, Tcpdump ran in the background on the honeypot and recorded the network traffic. This allowed us to observe exactly how the honeypot reacted to the various scanning techniques and whether there were any anomalies or peculiarities in the behavior of the network traffic that might have been influenced by our tunneling system.

This type of monitoring provided us with an additional dimension of understanding and gave us a clearer picture of how tunneling was affecting network interactions overall, independent of the specific scan results. It helped us better understand the overall nature and characteristics of network traffic routed through tunneling.

6.3.3 Results of the quality analysis

After performing the tests and evaluating the results, we were able to determine that our tunneling system worked as expected in terms of port scans. Regardless of whether the tunneling system was active or not, all open and closed ports were correctly identified. This shows that passing traffic through our tunneling system does not change or obfuscate the status of network ports.

Differences were observed when traceroute was used as part of the Nmap scan. However, this was to be expected since traceroute tracks the path of packets through the network, and we measured once from the attacker node and once from the local node which is positioned differently in the network in relation to the honeypot device. With the introduction of our tunneling system, an additional layer was added to the network path, which, therefore does not lead to changes in the results of Traceroute, which we were also able to verify using separate measurements.

The really interesting result concerned operating system detection using Nmap's OSScan. While one might expect slight differences between the TCP Stealth Scan and the full TCP Connect Scan, since the methods interact in different ways, it was surprising to see differences in the operating system detection results depending on our tunneling system. This suggests that our tunneling system, while correctly forwarding network traffic, introduces subtle changes in packet interactions that affect the fingerprints interpreted by Nmap.

It is important to emphasize that these differences are not necessarily a security issue, but they do show that adding a tunneling layer, even if it operates transparently, can have subtle effects on network traffic.

The differences in operating system detection results could in fact be due to the additional packet delays caused by the tunneling system. Nmap's OSScan uses various heuristics that rely on the timing and order of packet responses to create fingerprints of operating systems. Any additional latency introduced by tunneling can thus affect the results and lead to divergent detection results.

It is important to understand that the additional delays added by the tunneling system may not be problematic in a real deployment scenario. When the remote node is placed in a cloud environment, traffic is inevitably routed through multiple network nodes and devices before reaching the attacker. These additional hops and associated delays, along with other network factors such as jitter and packet loss, can affect overall latency. Thus, the delays introduced by tunneling would only be a small part of the total latency and could be masked by other network variables. In practice, an attacker might have difficulty measuring the exact times and distinguishing between the real hardware and the hardware routed through the tunneling system. Therefore, the observed delay, although technically interesting, should not be considered a significant obstacle in most real-world scenarios.

6.4 Deployment feasibility

Previously, we have extensively explored the development and evaluation of our tunneling system. Now, we turn our attention to the specific technical challenges and details that arise when integrating this system into existing network infrastructures. This section focuses specifically on the technical aspects of network integration, including the specific requirements and potential obstacles that must be addressed. In doing so, we aim to provide a clear and concise overview of how the system can be deployed and operated in real network environments and what challenges we have encountered that may have a significant impact on deployability.

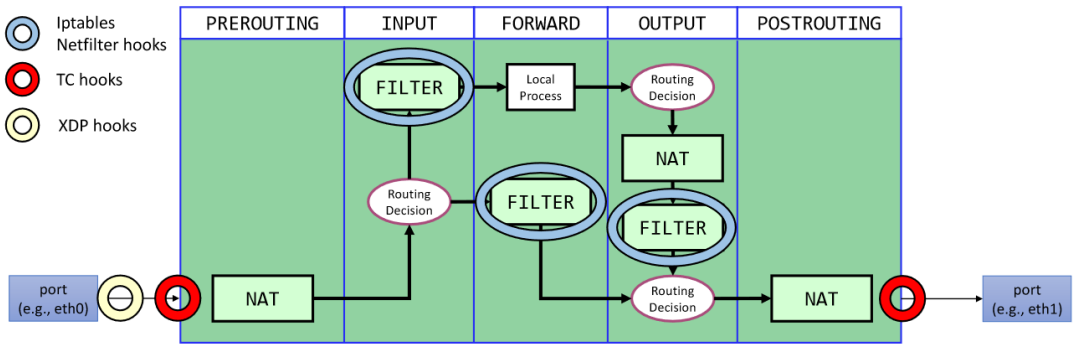

Figure 6.16: Linux Kernel Traffic Processing Hooks [7]

In some network environments, the complexity arises that our program must communicate with multiple network interfaces. This may be the case, for example, when traffic between remote nodes and local nodes is rerouted through existing infrastructure facilities, such as a VPN channel. Since our system is primarily designed to work with a single network interface, such scenarios may create the need for either the local or remote node to have the XDP program interact with two different network interfaces.

This added complexity of addressing multiple network interfaces introduced several difficulties. To address this problem, we considered running XDP on a dummy virtual interface. This would allow us to bypass the challenge of directly interacting with multiple physical network interfaces by shifting the complexity to a virtual interface. However, this solution is not without its own technical hurdles and introduces its own unique challenges to the implementation process.

A key issue that arose during our implementation concerned the coordination of network traffic between the physical network interfaces - those interfaces over which Internet traffic is intended to flow - and the virtual dummy interface running our XDP tunneling system.

Our original idea was to use iptables[8] to redirect incoming traffic to the dummy interface. However, with this approach, we found that the XDP ingress hook, the point of engagement where XDP interacts with the traffic flow, was not triggered. The reason for this is that forwarding of traffic with iptables in the kernel occurs after the execution of the XDP hook [7] (cf. Figure 6.16). This hierarchy of processing in the kernel led to the problem that we could not redirect incoming traffic as intended to our tunneling system.

Another problem arose when trying to direct outbound traffic from the XDP interface through iptables. Since iptables require direct access to the interface and it is "blocked" to some extent by XDP, we found that iptables cannot capture and forward outbound traffic to a different interface. In particular, it appeared that iptables was not correctly detecting outbound traffic generated by XDP, which caused further complexities in correctly redirecting traffic, which makes sense considering that our XDP program completely

[8] `https://linux.die.net/man/8/iptables` (Accessed October 2023)

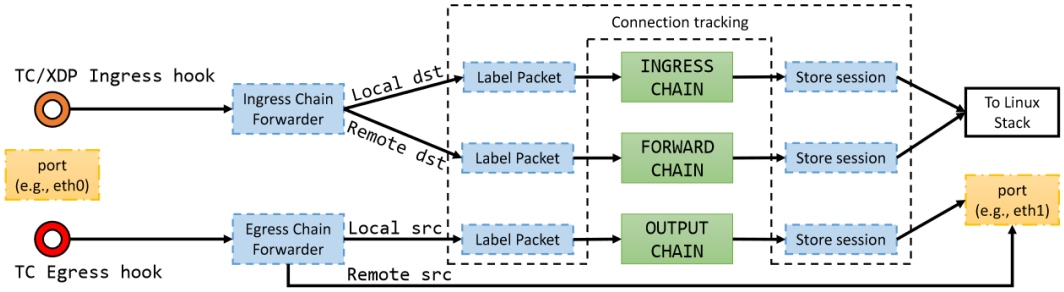

Figure 6.17: Linux Kernel Network Data Plane Structure [7]

bypasses all network traffic processing by the operating system (cf. Figure 6.17), as our system already transmits traffic during the execution of the XDP ingress hook.

Considering the previous difficulties and in order to make our XDP program compatible with multiple Linux interfaces, we decided to take an alternative approach. Our next attempt was to use a Virtual Ethernet Device (VETH) interface pair. The concept behind this solution is that a pair of VETH interfaces is created, with each interface acting as the counterpart of the other. Data sent to one of these interfaces will appear on the other and vice versa.

For the deployment of our application, we used one interface of this pair to implement conventional network rules using the tools at our disposal. This means that incoming and outgoing traffic could be handled using standard network tools on this interface. At the same time, we ran our XDP tunneling program on the counterpart of the interface pair. This allowed us to process traffic specific to our tunneling system while taking advantage of normal network tool functionality on the partner interface. This approach allowed us to bypass the complexity and limitations that had previously been created by using iptables and direct XDP integration with the physical network interfaces.

Although the approach of using a Virtual Ethernet interface pair looked promising on paper, we encountered a number of issues during its implementation. Initially, there were challenges in setting up the virtual interfaces to ensure that they did not use the loopback adapter, in which case the network traffic would not trigger the XDP hook. Once this obstacle was overcome, we were faced with other technical difficulties, including the need to configure manual ARP entries to ensure proper network communications.

Once these hurdles were overcome and the network was functional, we were able to confirm that the XDP hook was indeed triggered when traffic was sent through the VETH interface and correctly processed by our application. However, we encountered a new, unexpected problem: traffic generated by XDP and sent to the VETH pair's partner interface was not routed back to the original VETH interface as expected. The exact cause of this behavior is unclear, but it appears that the XDP implementation is somehow interfering with the mechanism by which the VETH interfaces are implemented in Linux. We suspect that the XDP program is generating the outgoing traffic in such a way that it is not recognized by the VETH pair's partner interface. This phenomenon presents a

significant technical barrier that requires further investigation.

During our evaluation, we found that despite intensive efforts and exploration of several approaches, integration of our system in environments with more complex network requirements remains problematic as our implementation eliminates the possibility of using common network management tools. While our application works reliably in certain scenarios, its limitations have become apparent in more complex network structures. This highlights the need for further research and possible adjustments to the design or implementation to ensure broader applicability and integration into more diverse network environments.

6.5 Security evaluation

One of the main objectives of the security assessment of any network system is to verify its resistance to attacks and malicious traffic. The system presented in this paper uses the XDP framework, which brings a number of security benefits.

First of all, XDP programs are executed inside an in-kernel Virtual Machine. This has the major advantage of running code in an isolated environment, greatly reducing the risk of unwanted interaction with other system components. In addition to this execution environment, XDP performs strict pointer boundary checks before loading any program into the Linux kernel. These checks ensure that all accesses to memory are within the allowed limits, effectively preventing special, malicious traffic from successfully performing, for example, buffer overflows or similar attacks. With these measures in place, it is virtually impossible to compromise our system directly through targeted network traffic.

However, one of the biggest security challenges is not directly in dealing with network traffic but in the way the local node is positioned in the network. This is especially key in the context of honeypots, where malicious and potentially dangerous traffic is often expected. The main goal is to ensure that this local node is embedded in a way that not only captures the desired traffic but also provides a secure environment that protects its surroundings from potential risks. With this in mind, it is important to look not only at the software itself but also at the bigger picture - the entire network topology and underlying protections and configurations, especially considering the hurdles encountered during deployment in more complex network topologies. Therefore, to mitigate the risk for the operator and to reduce the attack surface, it is advisable to isolate these two devices in the network or even create a separate, dedicated network for them. Such isolation significantly reduces the risk of unwanted security breaches that could result from external attacks on other parts of the network.

While our implementation has been robustly developed and tested, we must also consider potential vulnerabilities that are beyond our direct control. Such a risk could be introduced by other programs and services running in combination with our system. One particular feature that could serve as an attack vector is the necessary IP whitelisting of our application. This feature allows only certain trusted IP addresses to pass through our system, which is required to maintain control over the machine.

An attacker could attempt to bypass the whitelisting security measures by swapping his own IP address with one of a trusted network device, a process known as IP address spoofing. This would allow the attacker to attempt to push unauthorized traffic through our system as if it were from a trusted source.

However, while IP address spoofing is a potential threat, we consider the associated risk to be relatively low. A primary reason for this assessment is that a successful spoofing attempt will not result in any network traffic flowing back to the attacker as long a potential attacker does not have control of the surrounding network infrastructure. In practice, this means that establishing a functional TCP connection, which would be necessary for many more complex attacks, is not possible. The lack of a back channel makes it significantly more difficult for attackers to perform advanced attacks or exploits.

In conclusion, although XDP provides a high level of security against direct attack attempts, it is paramount to consider the surrounding factors and ensure that the entire system is robust and resilient against a variety of threat vectors.

7 Discussion and Future Work

In the course of this research project, we have been intensively involved in the development and evaluation of our tunneling system. The insights gained have not only demonstrated the potential of our approach but also identified areas where there is room for further investigation and optimization. While our results appear applicable and robust in practice, it is crucial to consider the continuous development and adaptation of such systems.

In this chapter, we aim to provide deeper insights into some of the key aspects of our work that emerged during the evaluation. These include both our observations and reflections on the potential limitations of our system. Furthermore, we would like to provide a perspective on what steps could be taken in future work to enhance further the performance, security, and applicability of the system.

It is essential to emphasize that in the ever-evolving world of cybersecurity, no system can be considered final or perfect. Therefore, this section is not only a discussion of current results but also an invitation for continued research and development in this exciting and critical area.

7.1 Main results

During the course of our investigation and evaluation of the tunneling system, we were able to achieve a number of key findings that both underscore the performance of our system and provide insight into potential areas for improvement and enhancement.

- Performance and latency: Our evaluations have shown that our system, despite the introduction of an additional tunneling component, has acceptable latency. This highlights the system's ability to be deployed in real network environments without significantly impacting network traffic. However, it is always important to keep in mind that it is inevitable to hide the additional network latency, which can have an impact on the overall system in certain conditions.

- Tunneling bandwidth: Despite the challenges of implementing the tunneling system, the results show that the bandwidth is sufficient for typical honeypot scenarios.

- Security evaluation: Our system shows robust defenses against direct attacks. The implementation via XDP provides a strong line of defense, especially by executing programs in the in-kernel VM and using pointer boundary checks. Nevertheless, we also identified potential vulnerabilities such as IP whitelisting and its vulnerability to IP address spoofing.

- Tunneling quality: The results of the fingerprint and other tests showed that, from an attacker's perspective, there is no difference between the actual honeypot device and the remote node when our tunneling system is active, and all network traffic is correctly routed in both directions.
- Operating system detection: Here we encountered interesting discrepancies resulting from packet delays due to tunneling. However, this is probably not a major problem in practical scenarios, as many factors can affect packet timings.
- Deployment feasibility: The investigation into the usability of our system revealed significant challenges in integrating it into complex network environments, especially in conjunction with multiple Linux network interfaces and VETH adapters. Despite the potential of the XDP framework, further research and adaptations are needed to overcome the identified limitations and ensure broader applicability.

In summary, these main results have increased our confidence in the effectiveness and robustness of our system. However, they have also highlighted areas where further investigation and adjustments could be made to increase overall efficiency and safety.

7.1.1 Influence of the virtual test bed on the measurement results

Using a virtual test bed to evaluate our system brings both benefits and challenges. A crucial understanding of this environment is indispensable to correctly classify the results and understand their relevance to real-world applications.

A key aspect of using a virtual test bed is the absence of direct interaction with the network driver plane when it comes to XDP. XDP was originally designed to handle traffic at the earliest point of the network stack, directly at the driver level. This allows for maximum efficiency and minimum latency. However, since our virtual testbed cannot operate directly at this level, the XDP programs in our environment are potentially less efficient than in a native system with hardware support as the XDP support is emulated at the operating system level instead.

At first glance, this might appear to be a decisive disadvantage. However, we look at this in a different light: Our evaluation in such a setup gives us valuable insights into a possible worst-case scenario. If our system delivers acceptable and satisfactory results even under these conditions, then it could perform even better in real environments with hardware support for XDP.

In addition, there is another aspect to consider. In many public cloud environments, there might not be hardware support for XDP. This limitation makes our investigation and evaluation in a virtual test bed particularly relevant. Cloud service providers often rely on virtualization to deliver services, and the lack of hardware support for specific functions, such as XDP, is not uncommon. Therefore, our testbed also represents the reality of many public cloud infrastructures to some extent.

The choice of our specific test bed offered us another decisive advantage: absolute control over the entire network and all its components. In an environment where precision and reproducibility are critical, such control cannot be underestimated. It allowed us to isolate

and control all variable factors, from network latency to specific traffic patterns. This ensured that all of our tests were conducted under consistent and repeatable conditions, which in turn led to reliable and meaningful results.

In an external test bed, such as those offered by public cloud providers, such granularity of control would not have been possible. Even when highly configurable environments are offered, there are still aspects that are hidden from the end-user or cannot be directly influenced. For example, network performance fluctuations, background traffic, or the exact hardware configuration and performance in a public cloud environment cannot always be fully controlled or even viewed.

By having complete control of our own test bed, we were able to ensure that external factors were minimized and that our results directly reflected the performance and characteristics of our system, free of the potential glitches or unknowns that might occur in a less controlled environment.

In conclusion, although the use of a virtual testbed has unavoidable effects on the measurement results, these effects, in some way demonstrate the robustness of our system. They show that it can perform even under less-than-ideal conditions and provide guidance for future implementations in real network environments.

7.1.2 Limitations

In scientific studies and technical evaluations, it is essential to recognize and acknowledge the possible limitations and constraints of the system under study and the methodology used. Our study and our developed system are no exception. While we endeavored to conduct a comprehensive analysis, there are still some aspects that deserve attention:

- Virtual Test Bed: As discussed earlier, the use of a virtual test bed has performance implications, particularly with respect to XDP implementation. Although we have presented arguments for the merits of this decision, this remains a limitation that must be considered when interpreting the results.

- Scalability: Our tests were conducted in a controlled environment with defined parameters. How the system performs on a larger scale or under extreme traffic was not extensively tested, in part because the testbed itself could skew the measurement results due to the computational overhead required.

- Real network traffic: The synthetic test scenarios we used to simulate typical network scenarios cannot cover all possible real scenarios. Especially in case of anomalies or rare traffic patterns, different results might occur. We focused our tests primarily on the correct functioning of traffic forwarding, but attacks in the context of computer networks are often not typical network traffic. Extensive testing of this type is beyond the scope of this work.

- Security evaluation: Our security considerations focused mainly on the attack opportunities we identified on a theoretical basis. However, there is always the

potential of unknown threats or zero-day exploits that are beyond the scope of our evaluation.

- MTU issue and encapsulation: Another notable limitation of our system concerns the Maximum Transmission Unit. The introduction of the encapsulation header in the tunneling process reduces the effective MTU for through traffic. If incoming packets already match the default MTU, this could potentially lead to connection failures. While mechanisms such as Path MTU Discovery (PMTUD) exist to work around such issues, we have found that PMTUD does not always work reliably in practice. This means that in certain network configurations or under certain conditions, connection problems could occur that are due to MTU restrictions.

- Virtual Networking Compatibility: The difficulties in integrating our system into complex network environments open up opportunities for future work and extensions. In particular, developing approaches to better support multiple network interfaces and optimize the interaction of XDP and VETH adapters would be of great value. This could help to significantly increase the applicability and flexibility of our tunneling system in diverse network landscapes.

In conclusion, despite these limitations, our system and evaluation provide valuable insights and results. Nevertheless, it is important to consider these limitations when evaluating and applying our system.

7.2 Potential extensions and Future Research

In our research, we have laid a solid foundation for the tunneling system and its applications in network security. But as with many technological developments, our approach opens doors to further extensibility and in-depth research.

A key issue to be considered in future work is the optimization of MTU management. Our observations have shown that MTU management, especially in the context of packet encapsulation, needs further investigation. However, we were able to identify two potential solutions to address the challenges associated with encapsulation and effective MTU sizing. One approach would be to use IP fragmentation. By splitting larger packets into smaller fragments, the MTU size could be maintained without the packets losing size. While this offers a way around the MTU problem, it also brings its own challenges, particularly with regard to the order and integrity of the packet fragments, as well as potential security issues. Another approach could be to adopt solution approaches from established tunneling technologies such as WireGuard. WireGuard and similar technologies have already developed mechanisms to deal with MTU problems, especially in the context of encapsulation. Studying and possibly adopting their techniques could not only help solve the MTU problem but also provide other valuable insights for the development of our system. In addition, the integration of best practices from existing solutions could be a valuable addition to our system, not only in the context of MTU but also in other areas.

The difficulties we experienced in implementing multi-interface support reveal a clear gap that needs to be addressed in future research. Improved compatibility of our XDP implementation with common virtual networking techniques could significantly extend the applicability of our tunneling system in diverse network configurations. A deeper analysis of the interaction between these technologies, accompanied by the adaptation of our current implementation approach, could prove fundamental for the deployment of our system in more complex network environments.

Apart from the improvement of shortcomings we have identified so far, there are also many exciting opportunities to add new features to our system.

A very interesting area for future enhancements is the implementation of a hole-punching algorithm in our system. This would bypass the need to open ports in a firewall manually, thus increasing overall security and usability. Holepunching techniques could allow network connections to be made through NATs or firewalls without the need for explicit configurations. This would further increase the flexibility of our system and simplify its deployment in a variety of network topologies.

There is also the possibility of going beyond pure IPv4 support. A useful extension to our system would, therefore be to integrate IPv6 processing capabilities. Such an extension would ensure that our system not only meets current but also future network requirements and continues to provide a robust and reliable solution for diverse deployment scenarios. This is particularly interesting since, on the one hand, the use of IPv6 can provide future security, but on the other hand, IPv6 also opens up completely new attack vectors due to the significant difference between IPv4 and IPv6.

A promising extension to increase the reliability and efficiency of the system would be to establish a control channel between the remote node and the local node. Such a channel would allow health checks to be performed to monitor the operational status and performance of both nodes regularly. In addition, useful system statistics and management information could be transmitted in real-time through this channel. This would not only simplify the monitoring and management of the system but also help to identify potential problems at an early stage and react accordingly.

Another interesting approach to optimizing the system is checksum caching. In many network applications, checksums are essential to ensure the integrity of the transmitted data. Since in our case the checksum in the IP header is often identical between the local and remote nodes, caching these checksums could save considerable computing time. This could be particularly useful when the available computing power is limited by other extensions or requirements. However, the caching system would need to be carefully designed and tested to ensure that there are no unexpected side effects or security issues.

Apart from that, an integration of our tunneling system into larger network management or monitoring systems could be another interesting area of research. This could make our system an integral part of a holistic solution for network security and performance.

In addition, advanced profiling techniques could provide deeper insights into the performance of our system under various conditions. Finally, extending support to other network protocols could significantly broaden the system's range of applications.

In conclusion, although our system already provides promising results, there is still

room for further development and adaptation. The above suggestions and considerations could serve as a guide for future research in this area.

7.3 Answering of Research Questions

In this section, we return to the initial research questions posed during our study. Although many of these questions have already been addressed and answered extensively and implicitly throughout our investigation and discussion, we would now like to take the time to answer these questions explicitly and specifically. In doing so, we draw on all the evidence and data we have gathered to date to ensure a conclusive and coherent picture and to fill in any gaps in our arguments.

RQ4 What types of attacks and threats can be routed through tunneling solutions, and how effective are these solutions compared to other measures?

First of all, our discussion shows that by using our tunneling solution, network traffic is forwarded. This means that theoretically, all types of attacks are possible that are based on the transport protocols, i.e., TCP, UDP, and ICMP can be forwarded. This presents the tunneling solution as quite effective and versatile in terms of the variety of threats that can be dealt with.

However, a major concern with our tunneling solution is that there may be disclosure of the tunneling system if, as a result of an attack, the honeypot system connects to a third, previously unknown communication partner. Outbound traffic for which no mapping to a remote node exists will be routed through the last used remote node, which might in certain circumstances not be the one an attacker might suspect if concurrently a second attack from a different remote node is in progress. It is therefore critical to monitor carefully and, if necessary, restrict the outbound communication flows from the honeypot.

RQ5 How can the security of the tunneling software itself be ensured to prevent it from becoming the target of attacks, thereby compromising the integrity of the honeypot system?

To answer this question, we rely primarily on the inherent security of XDP and eBPF, respectively. These frameworks are known for their high security and efficiency, as they work directly within the Linux kernel and perform extensive checking of bytecodes before they are executed. This checking ensures that the eBPF code is safe and has no unwanted side effects. Nevertheless, as with any software, there is always the potential risk of zero-day exploits. These are security vulnerabilities that are not yet known to the developers and, therefore have not yet been patched. An attacker who knows about and exploits such a gap could theoretically compromise the system. It is, therefore, crucial to stay up to date with security updates and patches and to actively monitor the system to detect any unusual activity.

Furthermore, it is crucial to properly secure the local network where the local node and honeypot are located. This includes not only securing the communication channels and access points but also ensuring the anonymity of the operator of the system. If an

attacker can draw conclusions about the identity or location of the operator, this could compromise the integrity of the honeypot system.

RQ6 How can the results of this study help improve cybersecurity, and what other research topics and applications arise from the results?

First, the results of this study help improve cybersecurity by presenting a detailed investigation and implementation of our novel tunneling solution for honeypots. By effectively forwarding network traffic, including common TCP, UDP, and ICMP protocols from multiple remote nodes concurrently, we expand the range of potential attacks that can be observed, thus increasing the effectiveness of honeypots as early warning systems and research tools. The issues discussed in the main results and limitations, particularly the challenges related to MTU and the impact of delay due to tunneling on operating system detection, provide further starting points for research and optimization. These issues not only highlight the need for further fine-tuning of the proposed system but also point to potential research directions that could focus on improving the efficiency and robustness of such tunneling solutions.

In addition, the results open doors for numerous other applications and research topics, especially in the area of network tunneling optimization, integration with other security- and networking tools, and adaptation to new and emerging threats.

8 Conclusion

This thesis provides a deep dive into the topic of tunneling of network traffic from and to honeypots. This system helps increase the efficiency and effectiveness of honeypots while ensuring security and anonymity by disguising the true honeypot device. Our investigation and implementation demonstrated how tunneling techniques can be used to deceive potential attackers, significantly facilitating the collection of valuable data for security research.

A concise aspect of our research was the critical examination of existing approaches and technologies in the context of an extensive literature review. This approach allowed us to identify the strengths and weaknesses of existing solutions, evaluate possible implementation approaches, and integrate these findings into our own development processes. This review helped us identify recurring patterns and potential stumbling blocks in tunneling system implementations and related issues, allowing our work to benefit from previous approaches and recent advances in the field.

A central component of our work was an in-depth analysis of the theoretical basis for algorithmic processing of network packets. Our research showed that tunneling is not merely a practical tool but an inherent necessity for effectively delivering network traffic in the context of our work. By examining the mechanisms and intricacies of tunneling in this light, we were able to develop a deep understanding of its role in modern network security. This knowledge enabled us to design a powerful tunneling system and emphasize the importance of tunneling in a broader cybersecurity context.

In our work, we have gone into detail about the implementation process using XDP and have also shed light on the development of our test bed, which is an integral part of our investigation. We were particularly interested in highlighting successful approaches and those that proved less effective. This holistic approach should give future research a clear direction and make avoidable hurdles visible from the very beginning.

Through extensive testing and analysis, we confirmed our proposed system's functionality and outlined its limitations and challenges. Issues regarding the MTU, the operating system detection, and the compatibility with specific network configurations helped deepen our understanding of the issue's complexity.

The results of our work illuminate not only the potential but also the weaknesses of our approach. They point out directions for future research and improvements based on the findings presented here. Of particular note is the ability to transfer the insights gained in this context to other security domains and applications.
In conclusion, this work aims to contribute to cybersecurity research and lays the foundation for future research and development in tunneling systems implemented with XDP. We hope the findings and techniques presented here will help make the digital world a safer place.

Bibliography

[1] Abranches, M., Michel, O., Keller, E., Schmid, S.: Efficient Network Monitoring Applications in the Kernel with eBPF and XDP. In: 2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN). pp. 28–34 (Nov 2021). https://doi.org/10.1109/NFV-SDN53031.2021.9665095

[2] Amazon: Compute – Amazon EC2 Instance Types – AWS (2023), https://aws.amazon.com/ec2/instance-types/

[3] Aqun, Z., Yuan, Y., Yi, J., Guanqun, G.: Research on tunneling techniques in virtual private networks. In: WCC 2000 - ICCT 2000. 2000 International Conference on Communication Technology Proceedings (Cat. No.00EX420). vol. 1, pp. 691–697 vol.1 (Aug 2000). https://doi.org/10.1109/ICCT.2000.889294

[4] asudbring: Azure Virtual Network FAQ (Mar 2023), https://learn.microsoft.com/en-us/azure/virtual-network/virtual-networks-faq

[5] Baidya, S., Chen, Y., Levorato, M.: eBPF-based content and computation-aware communication for real-time edge computing. In: IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). pp. 865–870. IEEE (2018). https://doi.org/10/gr3w2w

[6] Belkhiri, A., Pepin, M., Bly, M., Dagenais, M.: Performance analysis of DPDK-based applications through tracing. Journal of Parallel and Distributed Computing **173**, 1–19 (Mar 2023). https://doi.org/10.1016/j.jpdc.2022.10.012, https://www.sciencedirect.com/science/article/pii/S0743731522002271

[7] Bertrone, M., Miano, S., Risso, F., Tumolo, M.: Accelerating Linux Security with EBPF Iptables. In: Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos. pp. 108–110. SIGCOMM '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3234200.3234228, https://doi.org/10.1145/3234200.3234228

[8] CAIDA Organization: Trace Statistics for CAIDA Passive OC48 and OC192 Traces (Sep 2010), https://www.caida.org/catalog/datasets/trace_stats/, section: catalog

[9] Chen, T., Gao, X., Chen, G.: The features, hardware, and architectures of data center networks: A survey. Journal of Parallel and Distributed Computing **96**, 45–74 (Oct 2016). https://doi.org/10.1016/j.jpdc.2016.05.009, https://www.sciencedirect.com/science/article/pii/S0743731516300399

[10] Clemente, P., Lalande, J.F., Cornabas, J.R.: HoneyCloud: Elastic Honeypots - On-attack Provisioning of High-interaction Honeypots. In: Proceedings of the International Conference on Security and Cryptography. pp. 434–439. SciTePress - Science and Technology Publications, Rome, Italy (2012). https://doi.org/10.5220/0004129604340439, http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0004129604340439

[11] Cohen, F.: Managing network security — The unpredictability of defence. Network Security **1998**(4), 12–14 (Apr 1998). https://doi.org/10.1016/S1353-4858(00)87598-4, https://www.sciencedirect.com/science/article/pii/S1353485800875984

[12] Cohen, F.: The Use of Deception Techniques: Honeypots and Decoys. Handbook of Information Security **3**(1), 646–655 (2006)

[13] Coonjah, I., Catherine, P.C., Soyjaudah, K.M.S.: Experimental performance comparison between TCP vs UDP tunnel using OpenVPN. In: 2015 International Conference on Computing, Communication and Security (ICCCS). pp. 1–5 (Dec 2015). https://doi.org/10.1109/CCCS.2015.7374133

[14] David Wragg: Unimog - Cloudflare's edge load balancer (Sep 2020), http://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/

[15] Deering, S.E., Mogul, J.: Path MTU discovery. Request for Comments RFC 1191, Internet Engineering Task Force (Nov 1990). https://doi.org/10.17487/RFC1191, https://datatracker.ietf.org/doc/rfc1191, num Pages: 19

[16] Derek Phanekham, Rick Jones: Using netperf and ping to measure network latency (Jun 2020), https://cloud.google.com/blog/products/networking/using-netperf-and-ping-to-measure-network-latency

[17] Diorio, R.F., Timóteo, V.S.: A Platform for Multimedia Traffic Forwarding in Software Defined Networks. In: Proceedings of the 21st Brazilian Symposium on Multimedia and the Web. pp. 177–180. WebMedia '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2820426.2820464, https://doi.org/10.1145/2820426.2820464

[18] eBPF.io authors: eBPF Applications Landscape, https://ebpf.io/applications/

[19] Facebook: facebookincubator/katran (Aug 2023), https://github.com/facebookincubator/katran, original-date: 2018-04-27T21:49:35Z

[20] Freitas, E., Filho, A.T.d.O., Carmo, P.R.X.d., Sadok, D.F.H., Kelner, J.: Takeaways from an experimental evaluation of eXpress Data Path (XDP) and Data Plane Development Kit (DPDK) under a Cloud Computing environment. Research, Society and Development **11**(12), e26111234200–e26111234200 (Sep 2022). https://doi.org/10.33448/rsd-v11i12.34200, https://rsdjournal.org/index.php/rsd/article/view/34200

[21] Google: CPU platforms | Compute Engine Documentation (Sep 2023), https://cloud.google.com/compute/docs/cpu-platforms

[22] Google: MTU considerations | Cloud VPN (Aug 2023), https://cloud.google.com/network-connectivity/docs/vpn/concepts/mtu-considerations

[23] Guezzaz, A., Asimi, A., Sadqi, Y., Asimi, Y., Tbatou, Z.: A new hybrid network sniffer model based on Pcap language and sockets (Pcapsocks). International Journal of Advanced Computer Science and Applications **7**(2) (2016). https://doi.org/10/gr3w28

[24] Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V., Zeiger, F., Frank, R., Menth, M.: A survey on data plane programming with P4: Fundamentals, advances, and applied research. Journal of Network and Computer Applications **212**, 103561 (Mar 2023). https://doi.org/10.1016/j.jnca.2022.103561, https://www.sciencedirect.com/science/article/pii/S1084804522002028

[25] IANA: Service Name and Transport Protocol Port Number Registry (2023), https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

[26] Jones, Z.H.: Performance analysis of XDP programs. In: 2021 Large Installation System Administration Conference (LISA). USENIX Association (Jun 2021)

[27] Kofod-Petersen, A.: How to do a Structured Literature Review in computer science. Ver. 0.2. October **1** (Oct 2014)

[28] Lahey, K.: TCP Problems with Path MTU Discovery. Request for Comments RFC 2923, Internet Engineering Task Force (Sep 2000). https://doi.org/10.17487/RFC2923, https://datatracker.ietf.org/doc/rfc2923, num Pages: 15

[29] Laki, S., Horpácsi, D., Vörös, P., Kitlei, R., Leskó, D., Tejfel, M.: High speed packet forwarding compiled from protocol independent data plane specifications. In: Proceedings of the 2016 ACM SIGCOMM Conference. pp. 629–630. SIGCOMM '16, Association for Computing Machinery, New York, NY, USA (Aug 2016). https://doi.org/10.1145/2934872.2959080, https://dl.acm.org/doi/10.1145/2934872.2959080

[30] Li, T., Farinacci, D., Hanks, S.P., Meyer, D., Traina, P.S.: Generic Routing Encapsulation (GRE). Request for Comments RFC 2784, Internet Engineering Task Force (Mar 2000). https://doi.org/10.17487/RFC2784, https://datatracker.ietf.org/doc/rfc2784, num Pages: 9

[31] Liatifis, A., Sarigiannidis, P., Argyriou, V., Lagkas, T.: Advancing SDN from OpenFlow to P4: A Survey. ACM Computing Surveys **55**(9) (2023). https://doi.org/10.1145/3556973, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85143388359&doi=10.1145%2f3556973&partnerID=40&md5=9a1d941b95f3792ce190201038ef7ab1

[32] Linguaglossa, L., Rossi, D., Pontarelli, S., Barach, D., Marjon, D., Pfister, P.: High-speed data plane and network functions virtualization by vectorizing packet processing. Computer Networks **149**, 187–199 (Feb 2019). https://doi.org/10.1016/j.comnet.2018.11.033, https://www.sciencedirect.com/science/article/pii/S1389128618312957

[33] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., Wright, C.: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. Request for Comments RFC 7348, Internet Engineering Task Force (Aug 2014). https://doi.org/10.17487/RFC7348, https://datatracker.ietf.org/doc/rfc7348, num Pages: 22

[34] Mairh, A., Barik, D., Verma, K., Jena, D.: Honeypot in Network Security: A Survey. In: Proceedings of the 2011 International Conference on Communication, Computing & Security. pp. 600–605. ICCCS '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1947940.1948065, https://doi.org/10.1145/1947940.1948065

[35] Mallory, T., Kullberg, A.: Incremental updating of the Internet checksum. Request for Comments RFC 1141, Internet Engineering Task Force (Jan 1990). https://doi.org/10.17487/RFC1141, https://datatracker.ietf.org/doc/rfc1141, num Pages: 2

[36] mamccrea: Azure VM sizes - General purpose - Azure Virtual Machines (Aug 2022), https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-general

[37] Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M.: Creating complex network services with eBPF: Experience and lessons learned. In: IEEE Int. Conf. High Perform. Switch. Rout., HPSR. vol. 2018-June. IEEE Computer Society (2018). https://doi.org/10.1109/HPSR.2018.8850758, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85056447646&doi=10.1109%2fHPSR.2018.8850758&partnerID=40&md5=35b2465bf5daca9a37edd460fe3eb867

[38] Nawrocki, M., Wählisch, M., Schmidt, T.C., Keil, C., Schönfelder, J.: A Survey on Honeypot Software and Data Analysis (Aug 2016), http://arxiv.org/abs/1608.06249, arXiv:1608.06249 [cs]

[39] Networks, P.: Why does the GRE tunnel not come up in Azure Cloud - Knowledge Base - Palo Alto Networks (Oct 2018), https://knowledgebase.paloaltonetworks.com/KCSArticleDetail?id=kA10g000000PNAvCAO&lang=en_US%E2%80%A9

[40] Nikita Shirokov, Ranjeeth Dasineni: Open-sourcing Katran, a scalable network load balancer (May 2018), https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

[41] Nurhaida, I., Ngadiyono: Quality of Service for Traffic Monitoring System Based on Static Routing Using EoIP Tunnel over IPSec. In: Proceedings of the 2019 Asia Pacific Information Technology Conference. pp. 91–99. APIT 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314527.3314543, https://doi.org/10.1145/3314527.3314543

[42] Ogudo, K.A.: Analyzing Generic Routing Encapsulation (GRE) and IP Security (IPSec) Tunneling Protocols for Secured Communication over Public Networks. In: 2019 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD). pp. 1–9 (Aug 2019). https://doi.org/10.1109/ICABCD.2019.8851004

[43] Osinski, T., Tarasiuk, H., Chaignon, P., Kossakowski, M.: A Runtime-Enabled P4 Extension to the Open vSwitch Packet Processing Pipeline. IEEE Transactions on Network and Service Management **18**(3), 2832–2845 (2021). https://doi.org/10.1109/TNSM.2021.3055900, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85100740997&doi=10.1109%2fTNSM.2021.3055900&partnerID=40&md5=d67d5649bac49347e7605313eee7add4

[44] Rijsinghani, A.: Computation of the Internet Checksum via Incremental Update. Request for Comments RFC 1624, Internet Engineering Task Force (May 1994). https://doi.org/10.17487/RFC1624, https://datatracker.ietf.org/doc/rfc1624, num Pages: 6

[45] R.T. Braden, D.A. Borman, C. Partridge: Computing the Internet checksum. Request for Comments RFC 1071, Internet Engineering Task Force (Sep 1988). https://doi.org/10.17487/RFC1071, https://datatracker.ietf.org/doc/rfc1071, num Pages: 24

[46] Sharaf, H., Ahmad, I., Dimitriou, T.: Extended Berkeley Packet Filter: An Application Perspective. IEEE Access **10**, 126370–126393 (2022). https://doi.org/10.1109/ACCESS.2022.3226269, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85144028617&doi=10.1109%2fACCESS.2022.3226269&partnerID=40&md5=ddca00c5b332689bfc6416d90b415f5b

[47] Shen, Y., Zhang, Q.f., Wang, Y.f., Li, W.j., others: A multi-tunnel VPN concurrent system for new generation network based on user space. In: 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications. pp. 1334–1341. IEEE (2012). https://doi.org/10.1109/trustcom.2012.41

[48] Sikos, L.F.: Packet analysis for network forensics: A comprehensive survey. Forensic Science International: Digital Investigation **32**, 200892 (2020). https://doi.org/10.1016/j.fsidi.2019.200892

[49] Terin Stock: High Availability Load Balancers with Maglev (Jun 2020), http://blog.cloudflare.com/high-availability-load-balancers-with-maglev/

[50] Thomas Ptacek: BPF, XDP, Packet Filters and UDP (Oct 2020), https://fly.io/blog/bpf-xdp-packet-filters-and-udp/

[51] Tran, V.H., Bonaventure, O.: Beyond socket options: Towards fully extensible Linux transport stacks. Computer Communications **162**, 118–138 (Oct 2020). https://doi.org/10.1016/j.comcom.2020.07.036, https://www.sciencedirect.com/science/article/pii/S014036642031848X

[52] Valencia, A.J., Zorn, G., Palter, W., Pall, G.S., Townsley, M., Rubens, A.: Layer Two Tunneling Protocol "L2TP". Request for Comments RFC 2661, Internet Engineering Task Force (Aug 1999). https://doi.org/10.17487/RFC2661, https://datatracker.ietf.org/doc/rfc2661, num Pages: 80

[53] Van Tu, N., Yoo, J.H., Hong, J.K.: Building Hybrid Virtual Network Functions with eXpress Data Path. In: Lutfiyya H., Diao Y., Zincir-Heywood N., Badonnel R., Madeira E. (eds.) Int. Conf. Netw. Serv. Manag., CNSM. Institute of Electrical and Electronics Engineers Inc. (2019). https://doi.org/10.23919/CNSM46954.2019.9012730, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85081982503&doi=10.23919%2fCNSM46954.2019.9012730&partnerID=40&md5=9f72574f1fdc6c1af423e0fda0fb1c9d

[54] Vieira, M., Castanho, M., Pacífico, R., Santos, E., Câmara, E., Vieira, L.: Fast packet processing with EBPF and XDP: Concepts, code, challenges, and applications. ACM Computing Surveys **53**(1) (2020). https://doi.org/10.1145/3371038, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85079572561&doi=10.1145%2f3371038&partnerID=40&md5=ff564b94cfa31c849985ed61e8350d5e

[55] Wang, H., Wu, B.: SDN-based hybrid honeypot for attack capture. In: 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC). pp. 1602–1606. IEEE (2019). https://doi.org/10.1109/itnec.2019.8729425

[56] Wu, Z., Xie, M., Wang, H.: Design and implementation of a fast dynamic packet filter. IEEE/ACM Transactions on Networking **19**(5), 1405–1419 (2011). https://doi.org/10.1109/TNET.2011.2111381, https://dl.acm.org/doi/10.1109/TNET.2011.2111381

[57] Xu, Q., Wong, M., Wagle, T., Narayana, S., Sivaraman, A.: Synthesizing safe and efficient kernel extensions for packet processing. In: SIGCOMM - Proc. ACM SIGCOMM Conf. pp. 50–64. SIGCOMM '21, Association for Computing Machinery, Inc, New York, NY, USA (2021). https://doi.org/10.1145/3452296.3472929, https://www.scopus.com/inward/record.uri?eid=2-s2.0-85113203595&doi=10.1145%2f3452296.3472929&partnerID=40&md5=2009e95599442f0308c95b3871a4efee

[58] Zhang, F., Zhou, S., Qin, Z., Liu, J.: Honeypot: a supplemented active defense system for network security. In: Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies. pp. 231–235 (Aug 2003). https://doi.org/10.1109/PDCAT.2003.1236295

[59] Zhang, K., Bi, J., Wang, Y., Zhou, Y., Liu, Z.: Tunneling over IP Based on Match-Action Table in Software Defined Networks. In: Proceedings of the 13th International Conference on Future Internet Technologies. CFI 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3226052.3226054, https://doi.org/10.1145/3226052.3226054

[60] Zhang, X., Cui, L., Wei, K., Tso, F.P., Ji, Y., Jia, W.: A survey on stateful data plane in software defined networks. Computer Networks **184**, 107597 (Jan 2021). https://doi.org/10.1016/j.comnet.2020.107597, https://www.sciencedirect.com/science/article/pii/S1389128620312305

Acronyms

μs Microseconds. 78, 84, 85

ACM Association for Computing Machinery. 9, 10, 12

API Application Programming Interface. 14, 21

ARP Address Resolution Protocol. 62, 63, 100

BCC BPF Compiler Collection. 50, 66, 67

BDP Bandwidth Delay Product. 88

BMv2 Behavioral Model v2. 14

BPF Berkeley Packet Filter. 11, 16, 17

CAIDA Center for Applied Internet Data Analysis. 39, 40

CPU Central Processing Unit. 14, 20, 74

DBLP Digital Bibliography & Library Project. 10–12

DF Don't Fragment. 70

DPDK Data Plane Development Kit. 14, 15, 20, 21, 24, 25

eBPF Extended Berkeley Packet Filter. 11, 15, 16, 20, 21, 24, 25, 27, 39, 50, 52, 53, 62, 63, 66–68, 108

GB Gigabyte. 74

Gbit Gigabit. i, iii

Geneve Generic Network Virtualization Encapsulation. 15

GRE Generic Routing Encapsulation. 6, 15, 17, 19–21

IANA Internet Assigned Numbers Authority. 58

ICMP Internet Control Message Protocol. 39–41, 57, 64, 70, 71, 108, 109

IEEE Institute of Electrical and Electronics Engineers. 9–12

IP Internet Protocol. 1, 6, 19, 27, 30, 31, 33–37, 53, 54, 56–58, 60–65, 67, 68, 79, 101–103, 106, 107

IPsec Internet Protocol Security. 6, 18, 21

IPv4 Internet Protocol Version 4. ix, 39, 40, 64, 107

IPv6 Internet Protocol Version 6. 39, 40, 107

ISP Internet Service Provider. 54

IT Information Technology. 5, 7

KB Kilobyte. 88

L2TP Layer 2 Tunneling Protocol. 6, 18–21

LibPcap Packet Capture Library. 17

LLVM Low Level Virtual Machine. 50

LTS Long Term Support. 46, 74

MAC Media Access Control. 33, 54–60, 62, 67

Mbps Megabit per Second. i, iii, 88, 90

ms Milliseconds. 76, 78, 86, 88

MTU Maximum Transmission Unit. 68–71, 75, 106, 109, 111

NAPI New API. 14

NAT Network Address Translation. 107

NFV Network Function Virtualization. 14

NS3 Network Simulator 3. 44

OSI Open Systems Interconnection. 5, 6, 18, 19, 21, 54

OVS Open vSwitch. 15, 20, 21

P4 Programming Protocol-Independent Packet Processors. 14, 15, 20, 21, 24, 25, 27

PMTUD Path MTU Discovery. 70, 71, 106

PPTP Point-to-Point Tunneling Protocol. 6, 18

RAM Random Access Memory. 74

RFC Request for Comment. 64, 65, 70

RQ Research Question. 2, 24

SDN Software Defined Networking. 14, 15

SSL Secure Socket Layer. 6

TCP Transmission Control Protocol. xi, 6, 18, 39–41, 57, 64, 71, 87, 89–92, 94, 96–98, 102, 108, 109

TLS Transport Layer Security. 6

TTPs tactics, techniques, and procedures. 2

UDP User Datagram Protocol. xi, 18, 19, 39–41, 54, 57, 58, 60, 64–67, 87, 92–94, 97, 108, 109

UTC Coordinated Universal Time. 40

VETH Virtual Ethernet Device. 100, 104, 106

VM Virtual Machine. i, iii, 1, 3, 17–19, 21–25, 43–45, 74, 75, 101

VPN Virtual Private Network. 6, 17, 18, 20, 21, 49, 99

VXLAN Virtual Extensible LAN. 15, 17, 19–21

XDP eXpress Data Path. i, iii, 11, 16, 20, 21, 24, 25, 27, 39, 45, 50–53, 59, 62–64, 66–68, 70, 86, 95, 99–108, 111