Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Prototyische Implementierung und Evalutation eines asynchronen, performance-orientieren SYN-Portscanners in Rust

Bachelorarbeit

von

Lennard Alexander Dubhorn

Matrikelnummer: s0592852

Fachbereich 4 – Informatik, Kommunikation und Wirtschaft –
der Hochschule für Technik und Wirtschaft Berlin

zur Erlangung des akademischen Grades
Bachelor of Engineering (B. Eng.)
im Studiengang
Wirtschaftsinformatik

Tag der Abgabe: 06.02.2025

Erstgutachten:	Prof. Dr.-Ing. Alexander Stanik
Zweitgutachten:	Dr.-Ing. Ingmar Poese

Inhaltsverzeichnis

Kurzfassung

Diese Arbeit beschreibt die Erstellung einer internetfähigen Steuerung für elektrische Verbraucher. Anforderungen an die Steuerung werden nach dem Kano-Modell definiert. Über eine Nutzwertanalyse werden vorhandene Techniken und Standards bewertet. Exemplarisch wird die Lösung mit dem größten Nutzwert implementiert.

Ein Zigbit-Modul, bestehend aus einem AVR Mikrocontroller und einem IEEE 802.15.4 Funkchip, bildet die Basis für die Hardware. Zusammen mit einem selbst dimensioniertem Kondensatornetzteil wird das Modul in einem Steckdosengehäuse verbaut.

Um zukunftssicher zu sein, wird das Protokoll IPv6 eingesetzt. Die Adaptionsschicht übernimmt das Protokoll 6LoWPAN. Das verwendete Betriebssystem Contiki besitzt eine fertige Webserver-Applikation, die für die eigenen Zwecke angepasst wird. Das Protokoll IEC 60870-5-104 wird neu implementiert. Es basiert auf dem TCP/IP-Modell und wird vor allem im Umfeld von Energieleitsystemen eingesetzt. Es eignet sich besonders für einen automatisierten Zugriff.

Über eine öffentliche Adresse des IPv6-Tunnelbrokers SixXS ist die Steuerung weltweit erreichbar und der elektrische Verbraucher kann über einen Webbrowser oder von einem Energieleitsystem ein- und ausgeschaltet werden.

Die Anforderungen nach dem Kano-Modell wurden nahezu vollständig erfüllt. Die Implementierung eines Webservers und einer IEC 60870-5-104 Applikation ist mit den gegebenen limitierten Ressourcen möglich. Anwendungsmöglichkeiten für die Steuerung liegen im Bereich eHome und Smart Grid.

Abstract

This Master Thesis describes the implementation of a solution to control and monitor electric consumers via the Internet. Needs of this solution are defined by use of the Kano model. Existing technologies and standards are benchmarked by means of a cost-utility analysis. The solution that scores the highest value of benefit will be implemented typically.

A Zigbit Module forms the basis of the hardware. It bundles an AVR microcontroller and an IEEE 802.15.4 transceiver. Together with a self-dimensioned capacitive power supply it is mounted in a socket housing.

To be future-proof, the IPv6 protocol is used. The 6LoWPAN protocol handles the adaptation layer. Contiki is used as operating system. It is delivered with a ready-to-use web server application which is customized for the own purposes. The IEC 60870-5-104 protocol is implemented from scratch. It is based on TCP/IP and is used in the field of energy management systems. It is particularly suitable for automated access.

Via a public address given by IPv6 tunnel broker SixXS the solution is accessible worldwide. The electric consumer can be switched on and off by the means of a web browser or an energy management system.

The needs according to the Kano model are almost completeley achieved. It is possible to implement a solution consisting of a web server and IEC 60870-5-104 application in ressource constraint environments. Possible applications for such a solution are in the field of home automation and smart grid.

Kapitel 1: Einleitung

1.1 Motivation und Einführung in das Themengebiet

Das Scannen von Netzwerken oder gar dem gesamten Internet macht einen nicht zu vernachlässigenden Teil des Datenverkehrs im IPv4-Adressraum aus. So waren 98 Prozent des unaufgeforderten TCP-Verkehrs im Jahre 2024 weltweit auf `SYN`-Scans zurückzuführen [1]. Bekannte *Open-Source*-Internetscanning-Projekte wie ZMap, welches über 10 Jahre stetig weiterentwickelt wurde [2] oder Masscan [3] sind dazu in der Lage [4] den gesamten IPv4-Adressraum in der Größenordnung von Minuten zu scannen. Das Scannen von Netzwerken nach offenen Ports ermöglicht es Organisationen Schwachstellen ausfindig zu machen, bevor Angreifer es tun. Außerdem lassen sich durch das breitflächige Scannen von ausgewählten Adressräumen oder dem gesamten IPv4-Raum Informationen über Trends und Veränderungen dieser ableiten. Cyberangriffe haben Auswirkungen auf den Ruf und die finanzielle Stabilität von Unternehmen [5]. Die gegenwärtig hohen Angriffszahlen zum Beispiel bei *Denial-of-Service*-Angriffen [6] unterstreichen die Wichtigkeit.

Bisherige Hochleistungsscanner, wie die soeben genannten, wurden überwiegend in C entwickelt [4][3][7][8]. C ist häufig die Standardwahl für maschinennahe Anwendungen, da sie zum einen ein niedriges Level an Abstraktion und zum anderen hochperformant sein kann [9]. Allerdings ist C anfällig für menschengemachte Fehler [10] wie doppelte Speicher-Freigaben, Zugriffe auf bereits freigegebenen Speicher und Pufferüberläufe welche teils zu Speicherbeschädigungen und Sicherheitslücken führen können [11] [12]. Andere Sprachen wie zum Beispiel Go, Java oder Python lösen diese Probleme durch automatische Speicherverwaltung, insbesondere durch *Garbage Collection* und weitere Techniken. Diese Sprachen sind allerdings im Vergleich zu Sprachen ohne automatischer Speicherverwaltung wie C weniger performant [11].

Rust hingegen schneidet in Vergleichen bezüglich der *Performance* auf ähnlichem Niveau wie C ab, bringt gleichzeitig aber das höchste Sicherheitsniveau der genannten Sprachen mit, indem es Speicherfehler weitestgehend verhindert [11][13]. Außerdem unterstützt Rust Konzepte von Sprachen hoher Abstraktionsebene, wie beispielsweise die der funktionalen Programmierung oder Objektorientierung [13], während zudem in der zuletzt zitierten Untersuchung, auch die Anzahl der Zeilen niedriger als im Vergleich zu dem in C geschriebenen Code ist.

Bisher fehlt eine fundierte Untersuchung darüber, ob Rust als moderne Sprache, welche Sicherheitsgarantien, *High-Level*[1] Konzepte und *Performance* vereint, in Kombination mit aktuellen Linux-Schnittstellen, in der Lage ist, eine konkurrenzfähige Alternative zu gängigen Hochleistungsscannern, welche überwiegend in C geschrieben sind, darzustellen. Es ist ungeklärt, ob der potenzielle *Performance*-Unterschied gering genug ist, um durch die gewonnene Sicherheit kompensiert zu werden, weshalb diese Arbeit an diesem Punkt ansetzt.

1.2 Zielsetzung und Forschungsfrage

In dieser Arbeit wird ein prototypischer `SYN`-Portscanner zum breitflächigen Scannen von Netzwerken in Rust entwickelt. Der Fokus des Scanners liegt auf einer hohen *Performance* sowie hohen Effizienz, weshalb die Architektur teilweise asynchron gestaltet und leistungsfähige Linux-Schnittstellen wie `AF_PACKET`, `AF_XDP` und `eBPF` verwendet werden. Anschließend wird dieser bezüglich ausgewählter *Performance*-Metriken mit einer repräsentativen Auswahl an bestehenden Scannern verglichen und die Ergebnisse daraufhin evaluiert.

Es ergibt sich folgende Forschungsfrage: Inwieweit kann ein in Rust implementierter asynchroner `SYN`-Scanner hinsichtlich des Durchsatzes und der Ressourceneffizienz mit etablierten Hochleistungsscannern konkurrieren und durch sprach-eigene Sicherheitsgarantien eine tragfähige Alternative für den produktiven Einsatz darstellen?

1.3 Abgrenzung des Themas

Die Scanning-Methode beschränkt sich explizit auf das `SYN`-Scanning. Es ist die de facto Standard Methode und weist in Tests den niedrigsten Einfluss auf das Zielsystem, sowie die kürzeste Scan-Dauer auf [14].

Bei der in dieser Arbeit entwickelten Implementierung handelt es sich um einen horizontalen Scanner (siehe 2.2). Anders als beispielsweise beim regulären `SYN`-Scan des Tools Nmap [15], welcher in der Regel vertikal erfolgt. Vertikales Scanning ist für Netzwerk- beziehungsweise Internetscanner weniger relevant, da dabei das individuelle Ziel im Vordergrund steht.

Zusätzliche Mechanismen zur Verschleierung des Scans oder weiterführende Maßnahmen zur Treffererhöhung werden in dieser Implementierung rudimentär behandelt, da der Fokus auf der Nutzung von Rust, sowie der Entwicklung eines *Performance*-orientieren Netzwerkscanners liegt. Da der normale Ablauf des `SYN`-Scans bereits grundlegende Mechanismen diesbezüglich mitbringt [14], sind diese Gebiete für die Beantwortung der Forschungsfrage nicht notwendig. Außerdem beschränkt sich diese Arbeit auf den IPv4-Adressraum, da dies genügt, um der Forschungsfrage nachzugehen.

[1] Auf hoher Abstraktionsebene

Kapitel 2: Theoretische Grundlagen

In diesem Kapitel werden die nötigen Grundlagen zum Verständnis des Port-*Scannings* in Form von `SYN`-Scans, sowie das nötige Wissen über Netzwerkkommunikation, die genutzten Technologien und Linux-Schnittstellen vermittelt. Des Weiteren wird auf asynchrone Programmierung eingegangen, sodass ein Verständnis für das nachfolgende Konzept der Implementierung gegeben ist.

Anschließend werden die zum Vergleich genutzten Scanner vorgestellt und eingeordnet. Auch Rust und dessen Besonderheiten werden genauer vorgestellt.

2.1 Grundlagen der Netzwerkkommunikation

Bei der Kommunikation in TCP/IP [1] Netzwerken dient das IP-Protokoll und die IP-Adressen der Identifikation der Maschine im Netzwerk, während die genaue Adressierung der spezifischen Anwendungen durch sogenannte Ports bzw. der sogenannten Portnummer bestimmt wird [16]. Die Portnummer ist ein 16-Bit-Wert und kann somit zwischen jeweils einschließlich 0 und 65535 liegen [17, S. 107]. Einige Portnummern sind fest vergeben oder für bestimmte Anwendungen registriert [18], was es ermöglicht, gezielt nach bestimmten Anwendungen zu scannen. Der gesamte Kommunikations-Endpunkt wird *Socket* genannt [19, S. 1149].

2.1.1 Ports

Ports können in verschiedene Zustände eingeordnet werden. Für diese Arbeit ist nur die Unterscheidung zwischen offen und geschlossen/gefiltert relevant.

- **Offen:** Eine Anwendung lauscht auf dem Port und akzeptiert eingehende valide TCP oder UDP Anfragen [15].
- **Geschlossen / Gefiltert:** Der mit dem Port verbundene Service ist zwar ansprechbar, aber akzeptiert keine eingehenden Verbindungen / Es gibt lediglich eine ICMP (Fehler) Antwort oder gar keine, da beispielsweise kein Service für diesen Port existiert [15].

[1] Eine grundlegende Kenntnis über das TCP/IP-Modell wird angenommen

2.1.2 Transmission Control Protocol (TCP)

Das *Transmission Control Protocol* operiert in der Transportschicht des TCP/IP-Modells und ist eines der meistgenutzten Transportprotokolle des Internets [20, S. 71]. Es gewährleistet eine zuverlässige, verbindungsorientierte Datenübertragung zwischen den Prozessen der *Hosts*. Die ursprüngliche Spezifikation erfolgte im RFC 793 [21], welches durch RFC 9293 [22] konsolidiert wurde. Für die Entwicklung eines `SYN`-Scanners sind insbesondere der Aufbau des TCP-Headers und der Mechanismus des Verbindungsaufbaues entscheidend.

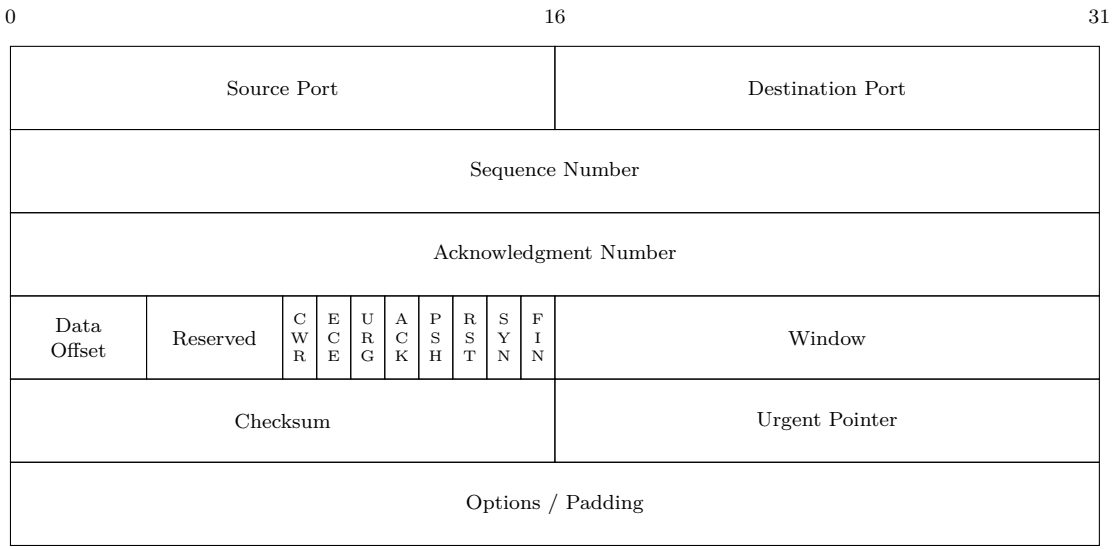

Abbildung 2.1: Aufbau des TCP-Headers nach RFC 9293 [22].

Da das TCP-Protokoll Daten als *Stream* statt einzeln (*Message*) versendet, wird vorher eine Verbindung in einem sogenannten *Three-Way-Handshake* aufgebaut [20] S71/72. Bei diesem werden TCP-Pakete mit jeweils unterschiedlichen Werten in den *Control Bits* (*Flags*) des TCP-Headers nach dem in 2.2 beschriebenen Muster ausgetauscht.

2.2 Portscanning

Portscanning, als Art des Netzwerkscannings, ist eines der fundamentalen Verfahren in der Netzwerksicherheit zum Auffinden von potenziellen Schwachstellen [16]. Ein Portscanner verschickt Pakete an ein Zielsystem und zieht anhand der Antworten, oder auch ausbleibenden Antworten, Rückschlüsse auf den Zustand des Systems. Das Ziel ist die Identifikation von offenen Ports bzw. aktiven Diensten, was als erster Schritt für weiterführende Sicherheitsanalysen oder aber auch Angriffe dienen kann [23, S. 4-3].

Beim Scannen von Ports können grundsätzlich zwei strategische Ausrichtungen unterschieden werden:

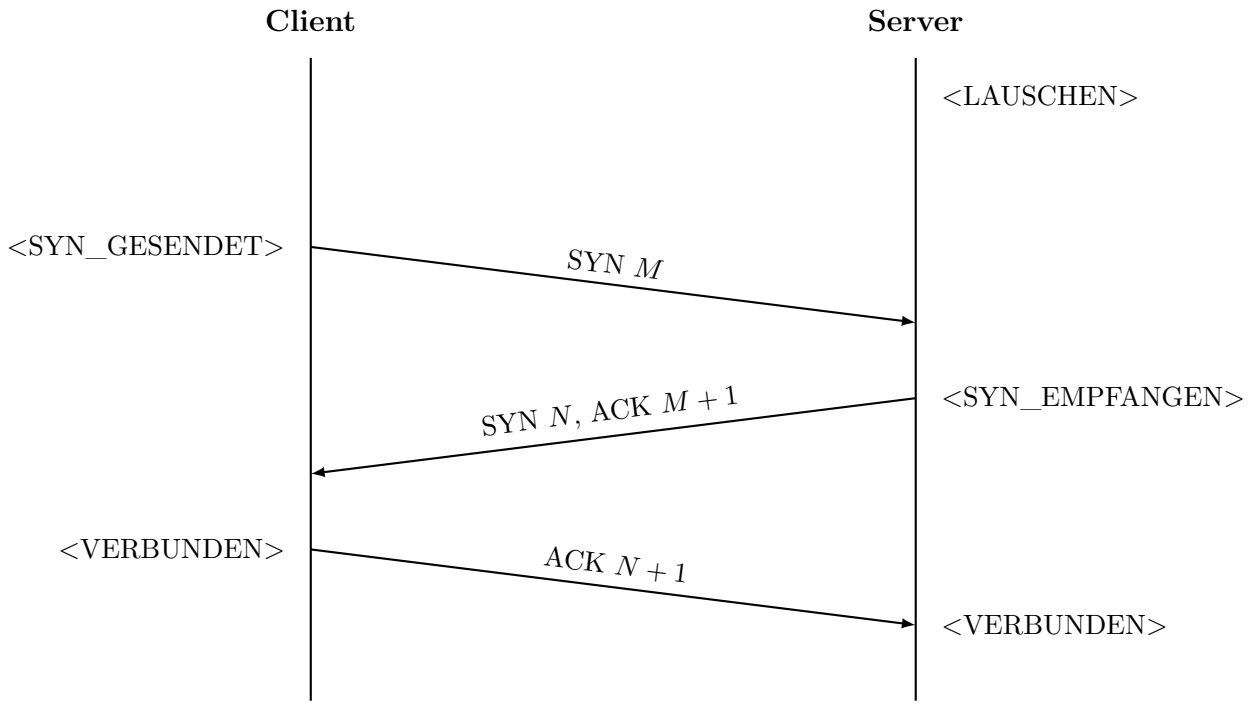

Abbildung 2.2: *Three-Way-Handshake* zum Aufbau einer TCP-Verbindung [20].

- **Vertikales Scannen:** Hierbei wird ein einzelner Ziel-*Host*[2] auf eine Vielzahl von Ports (oft alle 65535) gescannt, um ein möglichst vollständiges Profil möglicher Schwachstellen des Zielsystems zu erlangen und somit eher für das *Penetration Testing* geeignet ist.
- **Horizontales Scannen:** Ein sehr großer Adressbereich, beispielsweise das komplette IPv4-Internet wird gescannt. Dafür ist die Anzahl der zu scannenden Ports sehr klein oder auf einen einzigen beschränkt. Dies bietet die Möglichkeit wertvolle Daten über Trends oder die Verbreitung von Schwachstellen zu untersuchen [4].

2.2.1 SYN-Scanning

Für einen *Half-Open* `SYN`-Scan wie er in dieser Arbeit behandelt wird, sind lediglich die ersten beiden Schritte des in 2.2 dargestellten Verbindungsablaufes relevant. Es wird davon gebraucht gemacht, dass bereits und ausschließlich eine `SYN/ACK`-Antwort den Port als offen klassifiziert, was den weiteren Verbindungsaufbau irrelevant macht [15]. Um den Scan zu verschleiern und den Verbindungsversuch trotz dessen sauber abzuschießen, kann anschließend noch ein Paket, bei welchem die `RST`-Flag der *Control Bits* gesetzt ist gesendet werden [15].

Um antwortende *Hosts* effizient zu identifizieren, wird das Prinzip des `SYN`-Cookies adaptiert [4], welches ursprünglich als Abwehrmechanismus gegen *Denial-of-Service*-Angriffe

[2]Teilnehmer im Netzwerk, der über eine IP-Adresse adressierbar ist.

spezifiziert wurde [24, S. 8]. Dafür werden verbindungsspezifische Informationen unter Verwendung eines *Hash*-Algorithmus (z. B. *Keyed SipHash* TODO opt. Quelle) kodiert und als *Sequence Number* in den TCP-Header des ausgehenden `SYN`-Pakets eingetragen. Antwortet ein Ziel-*Host* mit einem `SYN-ACK`-Paket, so enthält dessen *Acknowledgment Number* gemäß TCP-Spezifikation den inkrementierten Wert der ursprünglichen *Sequence Number*. Die Validierung lässt sich abstrahiert wie folgt beschreiben:

```
is_valid = hash(value_0, value_1, ..., secret) == answer.ack_num - 1
```

Die Validierung der Antwort erfolgt somit rein mathematisch und benötigt keine Speicherung in einer lokalen Zustandstabelle. Dies erwirkt eine sowohl zeitliche als auch logische Entkopplung von Sende- und Empfangsprozessen, was wiederum eine asynchrone Architektur ermöglicht.

Entscheidend für den Scanner sind demnach die in 2.2.1 aufgeführten Header Felder.

Header Feld	**Beschreibung**
Source Port	Beschreibt den genutzten Port des Ausgangsdienstes.
Destination Port	Beschreibt den zu scannenden Port des Zielsystems.
Sequence Number	Wird zur Speicherung des `SYN`-Cookies genutzt.
Acknowledgement Number	Wird zum Abrufen des `SYN`-Cookies genutzt.
Control Bits (*Flags*)	Wird für die verschiedenen Phasen des Verbindungsaufbaues angepasst oder ausgelesen.

Tabelle 2.1: Relevante TCP-Header Felder

2.3 Schnittstellen zur Paketverarbeitung unter Linux

Um einen performanten Scanner zu bauen, müssen die genutzten Technologien zum einen für die Netzwerkprogrammierung geeignet und zum anderen hohe Sende- und Empfangsraten zulassen, während möglichst wenig Rechenressourcen verbraucht werden.

2.3.1 Linux

Linux ist ein *Open-Source*-Betriebssystem-Kernel [19, S. 1], welcher aufgrund neuartiger Subsysteme (wie beispielsweise `eBPF` (siehe 2.3.3) oder `XDP` (siehe 2.3.4)), eine programmierbare Paketverarbeitung nahe an der Hardware ermöglicht. Dies ist für die Entwicklung eines Hochleistungsscanners von großem Vorteil.

Ein zentrales Konzept zum Verständnis der *Performance*-Grenzen ist die Unterscheidung zwischen *User Space* und *Kernel Space* im Linux Ökosystem [19, S. 23]:

- ***Kernel-Space*:** Hier läuft der Kern des Betriebssystems mit vollem Zugriff auf die Hardware und den Speicher. Treiber und der Netzwerk-Stack operieren auf dieser Ebene.

- ***User-Space*:** Hier laufen reguläre Anwendungen in isolierten Speicherbereichen. Diese haben keinen direkten Zugriff auf den *Kernel Space*.

Die Kommunikation zwischen diesen Ebenen erfolgt über *System Calls* [19, S. 44]. Jeder Wechsel (*Context Switch*) zwischen *User-* und *Kernel Space*, sowie das Kopieren von Daten zwischen diesen Speicherbereichen, erzeugt *Overhead*. Beim Versenden und Empfangen sehr vieler Pakete summiert sich dieser *Overhead*, da jedes Paket im Normalfall sowohl *Kernel Space*, als auch *User Space*, durchschreitet. Dies belastet die CPU und wird für den Durchsatz zum Flaschenhals [25].

2.3.2 *Raw-Sockets* und *Address Families*

Als Endpunkt für die Kommunikation werden *Sockets* genutzt [26]. Die traditionelle Netzwerkprogrammierung unter Linux abstrahiert die Komplexität der Netzwerkprotokolle wie TCP. So übernimmt der Kernel dabei vollständig den *Three-Way-Handshake* und die Zustandsverwaltung [19, S. 1158]. Für einen `SYN`-Scanner ist dies ungeeignet, da der Scanner lediglich das initiale `SYN`-Paket senden und die Antwort registrieren will, ohne eine vollwertige Verbindung aufzubauen, welche Ressourcen im Kernel binden würde.

Raw-Sockets erlauben der Anwendung, Netzwerkpakete unter Umgehung bestimmter *Layer* des Kernel-Stacks zu senden und zu empfangen [27]. Der Entwickler muss die Protokoll-Header selbst konstruieren. Dies ist für *Half-Open* Portscanner essenziell, um individuelle Pakete zu generieren, ohne dass der Kernel automatisch in den Verbindungsaufbau eingreift.

Die *Address Families* definieren dabei die Interpretation der Adressen und die Ebene des Zugriffs [28]. Der Linux-Kernel stellt diverse Address-Familien bereit. Zum Verständnis, im Rahmen dieses Projektes, sind folgende Varianten von zentraler Bedeutung:

- **`AF_INET` (Netzwerk-Ebene):** Diese Familie operiert auf Layer 3 der IP-Ebene [27]. Bei Nutzung von *Raw-Sockets* fügt der Kernel standardmäßig den IP-Header hinzu und übernimmt das vollständige Routing zur korrekten Netzwerkschnittstelle [19, S. 1202].

- **`AF_PACKET` (Sicherungsschicht):** Diese Familie ermöglicht direkten Zugriff auf Layer 2 (Ethernet-Ebene). Anwendungen erzeugen vollständige *Ethernet-Frames* und haben somit die volle Kontrolle. Das Versenden oder Empfangen von Paketen erfordert jedoch weiterhin die Allokation von Kernel-internen Datenstrukturen [29].

- **`AF_XDP` (Hochperformant):** Hierbei handelt es sich um eine speziell für Hochleistungsanwendungen optimierte *Address Family*. Sie ermöglicht das Senden und Empfangen von Paketen unter Umgehung des regulären Kernel-Netzwerkstacks. Dabei ist zwischen dem universell verfügbaren *Copy-Mode*, in welchem Daten zwischen Kernel und User Space kopiert werden und dem Treiber-abhängigen *Zero-Copy-Mode*, in welchem Daten direkt in den Speicher der Anwendung geschrieben werden zu unterscheiden [25].

2.3.3 Erweiterte Berkeley Packet Filter (`eBPF`)

Ursprünglich als *Berkeley Packet Filter* (`BPF`) für Werkzeuge wie `tcpdump` entwickelt, um Pakete effizient zu filtern [30], wurde die Technologie erweitert, sodass grundlegend neue Möglichkeiten außerhalb des reinen filtern von Paketen erschlossen wurden.

`eBPF` ist eine im Linux-Kernel integrierte virtuelle Maschine (VM), die es erlaubt, benutzerdefinierten *Bytecode* sicher und effizient im *Kernel*-Kontext (siehe 2.3) auszuführen, ohne Kernel-Module schreiben oder den Kernel neu kompilieren zu müssen [31]. `eBPF`-Programme werden zur Laufzeit durch einen *JIT-Compiler* (*Just-In-Time*) in native Maschinensprache übersetzt. Ein *Verifier* stellt vor der Ausführung sicher, dass der Code sicher ist [32]. So werden Fehler wie beispielsweise Endlosschleifen oder falsche Speicherzugriffe vermieden.

Für einen `SYN`-Scanner ist `eBPF` nützlich, da es ermöglicht, eingehende Antwortpakete (`SYN-ACK`) extrem früh zu filtern und an den *User Space* weiterzuleiten, bevor teure Speicherstrukturen des Kernels angelegt werden. So werden nur relevante Daten an den *User Space* weitergereicht.

2.3.4 eXpress Data Path (`XDP`)

`XDP` definiert eine limitierte Ausführungsumgebung für `eBPF`-Programme, die direkt im Kontext des Netzwerktreibers ausgeführt werden. Dies ermöglicht eine programmierbare und hochperformante Paketverarbeitung direkt im Betriebssystemkern. Im Gegensatz zu früheren Ansätzen, die den Kernel vollständig umgehen (z.B. `DPDK`), integriert sich `XDP` kooperativ in den bestehenden Stack. [25]

Ein `XDP`-Programm kann Pakete verwerfen (`XDP_DROP`), an den regulären Netzwerkstack weiterleiten (`XDP_PASS`), über dieselbe Schnittstelle zurücksenden (`XDP_TX`) oder an eine andere CPU bzw. einen *Userspace-Socket* umleiten (`XDP_REDIRECT`) [25][32].

Die Effizienz von `XDP` resultiert aus der Positionierung im Datenpfad. In herkömmlichen Linux-Netzwerkarchitekturen durchläuft ein Paket nach dem Empfang durch die Netzwerkkarte den gesamten Netzwerk-Stack. Erst danach erreichen die Daten den *User Space*. Dies erfordert CPU und Speicher- aufwendige *Context Switches* (*Context Switches*) zwischen *Kernel-* und *User-Mode*, sowie die Allokation komplexer Metadatenstrukturen (eines

`sk_buff` [3]) [25][33]. `XDP` greift vor dieser Allokation ein (siehe Abbildung 2.3). Tests zeigen, dass `XDP` auf einem einzelnen CPU-Kern bis zu fünfmal mehr Pakete pro Sekunde verarbeiten kann als der Standard Linux-Stack [25].

Die *Performance* und Verfügbarkeit von `XDP` hängen vom verwendeten Betriebsmodus ab. Nach Zhang et al. [33] und Vieira et al. [32] lassen sich drei Modi unterscheiden:

- ***Native Mode* (*Driver Mode*):** Dies ist der Standardmodus für Hochleistungsanwendungen. Das `XDP`-Programm wird direkt im Netzwerkkartentreiber ausgeführt. Die Verarbeitung erfolgt nach dem *DMA*-Transfer (*Direct Memory Access*) in den *Ring-Buffer*, aber vor der `sk_buff`-Allokation. Dies erfordert explizite Unterstützung durch den Treiber der Netzwerkkarte.

- ***Offloaded Mode* (*Hardware Mode*):** Hierbei wird das `eBPF`-Programm vom Kernel auf die Netzwerkkarte ausgelagert und direkt auf der Hardware ausgeführt. Dies bietet die höchste *Performance*, da die Host-CPU vollständig von der Paketverarbeitung entlastet wird, setzt aber die Nutzung einer sogenannten *Smart NIC* Netzwerkkarte voraus.

- ***Generic Mode* (*SKB Mode*):** Dieser Modus dient der Kompatibilität. Wenn ein Treiber `XDP` nicht nativ unterstützt, führt der Kernel das `XDP`-Programm im Netzwerkstack des Kernels aus. Zwar gehen hier die massiven *Performance*-Vorteile der Speicherersparnis verloren, jedoch wird sichergestellt, dass `XDP`-Anwendungen auf jeder Hardware funktionsfähig bleiben.

2.4 Die Programmiersprache: Rust

Rust ist eine multiparadigmatische Systemprogrammiersprache, die ursprünglich von Mozilla Research entwickelt wurde. Der Hauptfokus der Sprache ist die Sicherheit. Doch auch *Performance* und Nebenläufigkeit sind immer weiter in den Fokus gerückt [34]. Dabei schafft es Rust als erste Sprache Speichersicherheits-Konzepte von Sprachen hoher Abstraktionsebene mit der Entscheidungsfreiheit über die Ressourcenverwaltung von Sprachen niedriger Abstraktionsebene zu vereinen [35].

2.4.1 Konzepte und Besonderheiten

Sprachen hoher Abstraktionsebene bedienen sich häufig einer automatisierten Speicherverwaltung mithilfe eines *Garbage Collectors*, um Speicherfehler, welche das Sicherheitsniveau einer Sprache maßgeblich bestimmen [11], zu vermeiden. Rust hingegen nutzt ein einzigartiges Modell, welches durch drei zentrale Konzepte bestimmt wird:

[3] *Socket Buffer*

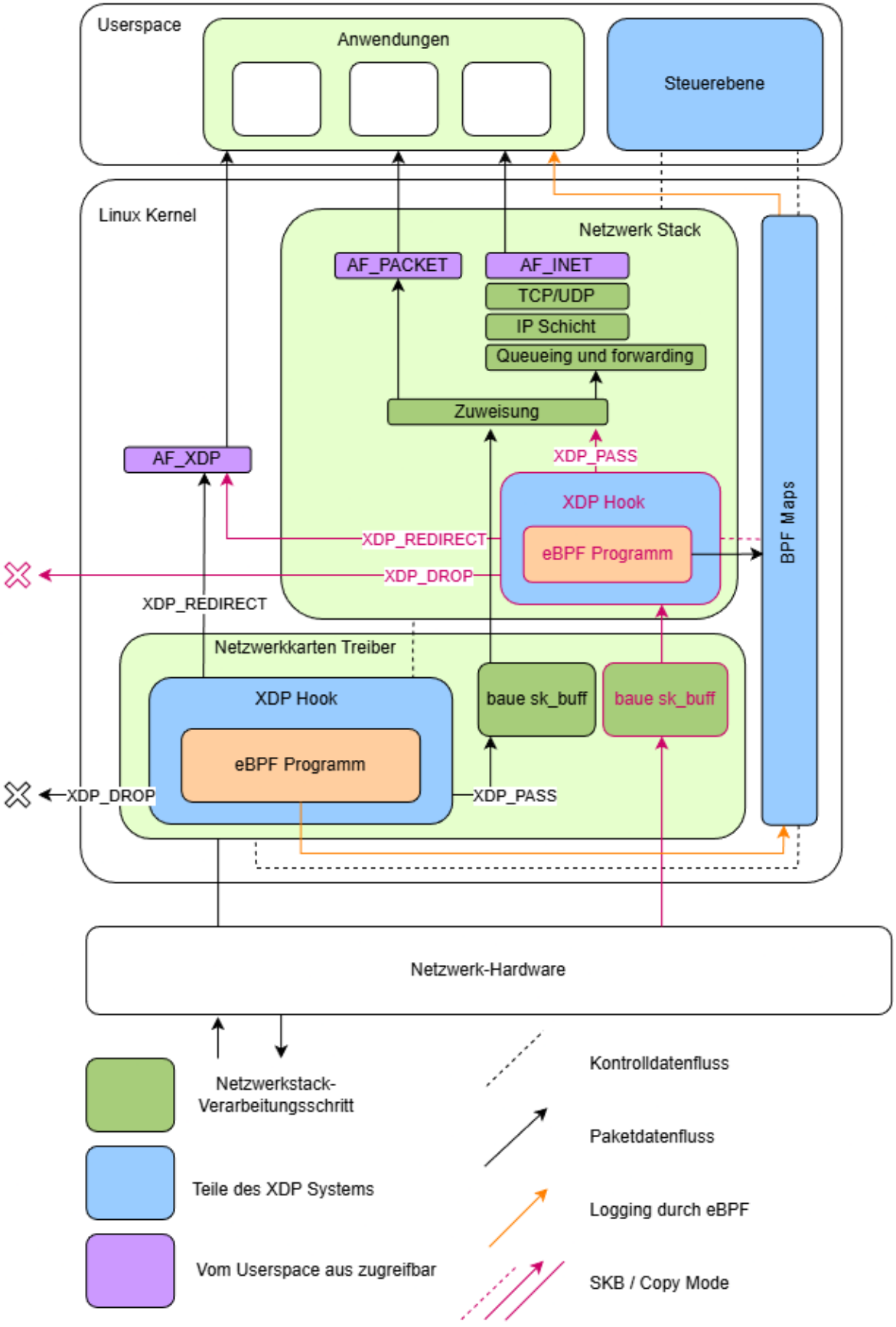

Abbildung 2.3: Der Empfangspfad durch den Kernel bei der Nutzung von `XDP` und `eBPF` (vereinfacht). Orientiert an Høiland et al. [25].

- ***Ownership*:** Jede Variable hat einen *Owner* (Besitzer). Wird der Besitzer gelöscht, wird auch die Variable gelöscht. Die Variable kann nur einen Besitzer haben [36]. Das bewirkt, dass der Programmierer sich nicht um das Freigeben des Speichers kümmern muss.
- ***Borrowing*:** Um eine Variable als Referenz in mehreren Kontexten nutzen zu können, ohne dessen *Owner* zu wechseln, gibt es das *Borrowing* Konzept, mit folgenden Regeln: es kann entweder eine veränderbare oder mehrere unveränderbare Referenzen einer Variable geben [34].
- ***Lifetimes*:** Jede Referenz in Rust besitzt eine Lebensdauer (*Lifetime*), welche den Gültigkeitsbereich definiert, in dem die Referenz valide ist. Meist implizit vom Compiler abgeleitet, verhindern *Lifetimes* falsche Zugriffe, indem sie sicherstellen, dass die referenzierten Daten mindestens so lange existieren wie die Referenz selbst [34].

Die Einhaltung dieser Regeln wird zur Kompilierzeit vom *Borrow-Checker* verifiziert. Außerdem hat Rust noch weitere Konzepte zur Steigerung der Sicherheit. Budgen et al. [34] führen weitere Konzepte wie das *Bounds Checking*, welches auf ungültige Indexzugriffe prüft oder die Nutzung von `Option`s, welche Zugriffe auf nicht initialisierte Werte vermeiden, indem sie eine Struktur, welche entweder den gewünschten Wert x als `Some(x)` oder `None` enthält, zurückgeben auf. Außerdem muss *Pointer*-Arithmetik in sogenannte `unsafe`-Blöcke ausgelagert werden. Diese dienen als eine Art Umgehungsmöglichkeit der anderen Konzepte. Sie lösen im Fehlerfall eine Programm-beendenden `panic` aus, welche verhindert, dass das Programm in einem undefinierten Zustand weiter läuft.

Trotz der Möglichkeit und Notwendigkeit, `unsafe`-Blöcke zu nutzen (beispielsweise für hardwarenahe Operationen oder der Arbeit mit C-Bibliotheken), was dem Sicherheitskonzept der Sprache widerspricht, sind laut Jung et al. „zahlreiche wichtige Rust Bibliotheken“ [36] sicher, da sie die `unsafe`-Blöcke korrekt kapseln [36].

Durch das Zusammenspiel dieser Konzepte können Speicherfehler verschiedenster Art bereits zur Kompilierzeit vermieden werden, was Rust zur sichersten unter den derzeit gängigen Sprachen macht [11]. Allerdings müssen deshalb auch einige Regeln bei der Programmierung beachtet und eingehalten werden, weshalb der Sprache eine steile Lernkurve zugeschrieben wird [37].

2.4.2 Asynchrone Programmierung und *Performance* von Rust

Die im letzten Abschnitt genannten Konzepte schließen auch *Data Races*, welche beim Zugriff mehrerer *Threads* [4] auf den gleichen Speicher entstehen können, bereits zur Kompilierzeit aus [13]. Dies macht *Data Races* zu einer häufigen Fehlerquelle in der asynchronen Programmierung [34]. Die Beseitigung derer, macht Rust zu einer guten Wahl für sowohl nebenläufige als auch parallele Programmierung.

[4] Untergeordnete Arbeitseinheiten eines Prozesses

Nebenläufige und Parallele Programmierung

Die nebenläufige Programmierung, welche in Rust durch das `async/await`-Modell umgesetzt wird, befasst sich mit der logischen Strukturierung von Software in unabhängige Kontrollflüsse. Diese agieren zeitlich verschränkt, wobei der primäre Zweck nicht die gleichzeitige Ausführung, sondern die Entkopplung von Aufgaben ist. So werden Ressourcen effektiv genutzt, indem sie während möglicher Wartezeiten z.B. bei *I/O*-Operationen[5], für andere Prozesse freigegeben werden. Dadurch kann die Effizienz und die Responsivität des Systems erhöht werden [38] [39].

Die Parallelität hingegen, bezieht sich auf die tatsächliche physikalische Ausführung mehrerer Aufgaben zum gleichen Zeitpunkt [38], was eine entsprechende *Multi-Core*-Hardware voraussetzt [40]. Der Vorteil der Parallelität liegt in der Leistungssteigerung und der Maximierung des Datendurchsatzes bei rechenintensiven Problemen [38][39].

Rusts Konzepte zur *Performance*-Steigerung

Neben den möglichen *Performance*-Vorteilen durch die nebenläufige Programmierung, welche aber letztendlich dem Programmierer überlassen ist, bietet die Sprache ihre größten internen *Performance*-Vorteile durch die *Zero-Cost Abstraction*. Das Konzept der *Zero-Cost Abstraction* [37], welches auch in der Sprache C++ Anwendung findet, kann nach Bjarne Stroustrup wie folgt beschrieben werden: „Was man nicht nutzt, dafür bezahlt man nicht. Was man nutzt, könnte man selbst nicht besser per Hand codieren“ [41].

Dazugehörige Konzepte sind beispielsweise die Eliminierung von Laufzeit-*Overhead* durch die Vermeidung eines zur Laufzeit arbeitenden *Garbage Collectors* [11] [37], die *Monomorphisierung* um die Typen oder Größen generischer Strukturen wie z.B. `Vec` [6] oder `Option` nicht mehr während der Laufzeit bestimmen zu müssen [37], oder die Bereitstellung eigener Iteratoren, welche die Leistung manuell geschriebener Schleifen oft übertrifft [37].

Diese Konzepte und vor allem die Prüfung der in 2.4.1 vorgestellten Konzepte zur Kompilierzeit, führt dazu, dass Rust in Benchmarks gängige Sprachen wie Java, Python, oder Go übertrifft und sogar mit der Geschwindigkeit von C konkurriert [13] [34] [37].

[5] Input-/Output-Operationen
[6] Vektor, ähnlich einer Liste

Kapitel 3: Stand der Technik

In diesem Kapitel wird im ersten Schritt die historische Entwicklung des horizontalen Netzwerkscannings betrachtet. Daraufhin werden in der Forschung etablierte Scanner und alternative Ansätze vorgestellt und diskutiert, und die resultierenden Nachteile betrachtet. Es folgt ein kurzer Überblick über weitere relevante Forschungsarbeiten zu relevanten Forschungssträngen sowie die Auswahl der Scanner, welche später als Vergleichsobjekte dienen.

3.1 Historische Entwicklung des horizontalen Netzwerkscannings

Ursprüngliche Netzwerkscanner wie Nmap [15] wurden primär für die vertikale Analyse einzelner *Hosts* oder kleiner Netzwerke konzipiert. Sie arbeiten teils Zustands-behaftet, was bedeutet, dass für jede ausgesendete Anfrage ein eigener Eintrag im Arbeitsspeicher verwaltet wird, um den Verbindungsstatus abzubilden. Bei Internet-weiten Scans führt dieser Ansatz jedoch schnell zur Erschöpfung der Systemressourcen und limitiert die Scan-Geschwindigkeit drastisch. Ein vollständiger Scan des Internets benötigte mit diesen Methoden oft Wochen oder Monate [4].

Der entscheidende Durchbruch gelang 2013 mit der Veröffentlichung von ZMap durch Durumeric et al. Mithilfe eines radikalen Architekturwechsels hin zum zustandslosen Scanning konnte die Geschwindigkeit so weit gesteigert werden, dass 97 % der theoretischen Geschwindigkeit vom Gigabit-Ethernet erreicht wurden. Dies ermöglichte erstmals Scans des gesamten IPv4-Adressraums in unter 45 Minuten von einem einzelnen Rechner aus [4]. Spätere Arbeiten, wie Zippier ZMap, optimierten diesen Ansatz weiter, um auch bis zu 10-Gbps Leitungen auszulasten [42] und somit die anhaltende Relevanz des ZMap-Projektes zu unterstreichen.

3.1.1 Der Standard Scanner: ZMap

In der wissenschaftlichen Literatur gilt ZMap [4] als der De-facto-Standard und als das primäre Vergleichsobjekt für internetweite Scans. In einer Retrospektive aus dem Jahr 2024 stellen Durumeric et al. fest, dass ZMap die Art und Weise, wie Internetmessungen durchgeführt werden, fundamental verändert hat. Mit über 1.200 wissenschaftlichen Zitationen

und der Nutzung als Basis für kommerzielle Sicherheitsanalysen (z. B. Censys) ist es das am weitesten verbreitete Werkzeug seiner Art [2].

Der Kern der Leistungsfähigkeit von ZMap lässt sich auf drei wesentliche Implementierungsentscheidungen zurückführen:

- **Effiziente *I/O*-Schnittstellen:** ZMap nutzt standardmäßig `AF_PACKET` in Kombination mit *Memory Mapping* (`mmap`), um den *Overhead* des Kopierens zwischen Kernel und *User-Space* zu reduzieren. Zwar zeigten Erweiterungen wie Zippier ZMap [42], dass durch spezialisierte Treiber wie `PF_RING ZC` (*Zero Copy*) noch höhere Geschwindigkeiten möglich sind, jedoch weisen die Autoren darauf hin, dass solche externen Treiber oft Wartungsprobleme und Inkompatibilitäten mit sich bringen. Daher setzt die aktuelle Version von ZMap primär auf universell verfügbare Linux-Schnittstellen, auch wenn diese *Performance*-technisch limitiert sind [2].
- **Zustandslose Architektur:** ZMap nutzt das Prinzip der `SYN`-Cookies, um keinen Zustand für ausgehende Verbindungen im Arbeitsspeicher halten zu müssen.
- **Adressgenerierung mittels zyklischer Gruppen:** ZMap nutzt zyklische multiplikative Gruppen modulo p (wobei p eine Primzahl $> 2^{32}$ ist). Dies ermöglicht eine pseudozufällige Permutation des gesamten IPv4-Adressraums, was nötig ist, um Zielnetzwerke nicht zu überlasten. [4].

3.2 Alternative Implementierungsansätze

Neben der reinen *Socket*-Programmierung und klassischen *Raw-Sockets* haben sich weitere, teils modernere Technologien und Scanner-Architekturen aufgetan. Beispielsweise mithilfe von:

- ***Kernel-Bypass* mit `DPDK`:** Das Data Plane Development Kit (`DPDK`) erlaubt es Anwendungen, die Netzwerkkarte direkt aus dem *User-Space* anzusprechen und den Kernel komplett zu umgehen. Abu Bakar und Kijsirikul zeigen, dass `DPDK`-basierte Scanner extrem hohe Raten erzielen können [43]. Der Nachteil ist jedoch die hohe Komplexität, die exklusive Belegung von CPU-Kernen und die schwierige Integration in bestehende Systemumgebungen [25].
- **Eigener TCP-Stack im *User-Space* (Masscan):** Der Scanner Masscan [3] umgeht den Flaschenhals des Betriebssystems mithilfe eines eigenem, TCP-Stack im *User-Space*. Dies erlaubt es dem Scanner, die Statusverwaltung und das Timing von Paketen komplett unabhängig vom Kernel-*Scheduler* zu steuern. Der *Scheduler* ist normalerweise dafür verantwortlich, die Rechenzeit der CPU auf die laufenden Prozesse zu verteilen [19, S. 737]. Die dabei entstehenden Kontextwechsel können das präzise Timing von Hochleistungsanwendungen stören. Dadurch kann Masscan deutliche *Performance*-Gewinne gegenüber ZMap erreichen [44].

- ***Hardware-Offloading* und *SmartNICs*:** Um die CPU des *Host*-Systems zu entlasten, lagert IMap die Scan-Logik direkt auf die Netzwerkhardware aus. Durch den Einsatz von programmierbaren Switches oder *SmartNICs* können Pakete bereits auf der Netzwerkkarte generiert und Antworten gefiltert werden, bevor sie überhaupt die CPU erreichen. Dies erfordert jedoch spezialisierte Hardware. In dieser Untersuchung wurde mit Raten von 40Gbps getestet, wobei jedoch Raten von einem Terabit oder mehr laut Li et al. theoretisch möglich wären. [8]

3.3 Weitere relevante wissenschaftliche Arbeiten

Der Forschungsstand zu horizontalen Hochleistungs-Netzwerkscannern wurde bereits in den vorherigen Sektionen 3.1 und 3.2 aufgegriffen. Ergänzend dazu ist noch anzubringen, dass nach bestehenden Ansätzen zur Kernel-Umgehung die Arbeit zu `XDP` (*eXpress Data Path*) einen Paradigmenwechsel darstellt. Zuvor genutzte Umgehungsstrategien waren unter anderem Netmap [45], welches bereits 2012 Konzepte wie *Shared Memory*[1] und *Zero-Copy* einführte, aber den Netzwerkstack eher ersetzte, statt ihn zu komplementieren, oder `DPDK` (siehe 3.2). Høiland-Jørgensen et al. zeigen, dass mit `XDP` durch eine programmierbare Paketverarbeitung im Kernel-Treiber eine mit `DPDK` vergleichbare *Performance* erreicht werden kann, ohne die Integration in das Betriebssystem aufzugeben [25].

Zwei Studien vergleichen `SYN`-Scanner [44] [46], fokussieren sich aber eher auf die Trefferrate, welche in der Arbeit nicht priorisiert wird (siehe 1.3). Außerdem ist die Gestaltung der Testumgebung bei beiden Arbeiten nicht auf Hochleistungs-Szenarien ausgelegt.

Die Eignung von Rust für hochperformante Netzwerkprogrammierung wird in mehreren wissenschaftlichen Arbeiten evaluiert. Sagramoni et al. schrieben eine Netzwerkbibliothek in Rust und verglichen sie mit der ursprünglichen C-Bibliothek [47]. Gonzalez et al. entwickelten einen UDP Treiber für Linux und verglichen diesen mit einem ähnlichen C-Treiber [48]. Moon et al. erstellten einen *NAT* (Network Address Translator) und testeten den Durchsatz [49]. Alle kommen zu dem Ergebnis, dass Rust sich für die *Low-Level* Netzwerkprogrammierung gut eignet und eine minimal niedrigere *Performance* verglichen mit der aktuellen Standardsprache in diesem Bereich - C, aufweist. Emmerich et al. verglichen Rust mit einer Vielzahl von anderen Sprachen, indem sie einen Netzwerktreiber in jeder der untersuchten Sprachen schrieben und diese anschließend miteinander verglichen. Dabei stellte sich Rust aufgrund seiner Sicherheitsgarantien und Performanz als erste Wahl für zukünftige Treiber-Projekte heraus [50].

Weitere Arbeiten beschäftigen sich mit dem Thema Sicherheit von Rust verglichen mit anderen Programmiersprachen und kamen zum einheitlichen Ergebnis, dass Rust umfangreiche Sicherheitsgarantien mitbringt, die man so von keiner anderen gängigen Sprache erhält [34] [11] [51].

[1] Zwischen *User-Space* und *Kernel-Space* geteilter Speicher

3.4 Vergleichsobjekte für die Evaluation

Um die Eignung der Implementierung in seiner Funktion als Performanz-orientierter `SYN`-Scanner aussagekräftig evaluieren zu können, wird er im Evaluationsteil der Arbeit mit folgenden Scannern verglichen

- **Wissenschaftlicher Standard:** Als primäres Vergleichsobjekt dient ZMap [4], da dieser die historische Basis des Internetscannings darstellt (siehe 3.1.1). Da ZMap in C geschrieben ist und auf klassischen Linux-Schnittstellen basiert, dient er als *Baseline*, um zu untersuchen, ob die im Lösungsansatz gewählte Kombination aus Rust und `XDP` trotz der Sicherheitsgarantien mit der etablierten Referenz konkurrieren kann.
- **Performance-Referenz:** Ergänzend wird Masscan [3] herangezogen. Dieser gilt durch seinen eigenen *User-Space*-Stack (siehe 3.2) als einer der schnellsten verfügbaren Scanner. Dieser Vergleich ist sinnvoll, um die Effizienz der `XDP`-basierten Lösung gegenüber einem hochoptimierten C-Ansatz einzuordnen.

3.5 Nachteile bisheriger Ansätze

3.5.1 Nachteile C-basierter Ansätze

Obwohl ZMap und ähnliche Hochleistungsscanner (wie Masscan oder `DPDK`-Scanner) extrem effizient sind, basieren sie fast ausschließlich auf der Programmiersprache C. Diese technologische Monokultur bringt jedoch signifikante Nachteile mit sich.

Ein zentraler Nachteil ist die fehlende intrinsische Speichersicherheit von C [12]. Da die Sprache dem Entwickler die volle Verantwortung für die Speicherverwaltung überträgt, führen menschliche Fehler häufig zu schwerwiegenden Sicherheitslücken. Schwachstellen wie *Buffer Overflows* in C/C++-basierten Systemen zählen nach wie vor zu den häufigsten Ursachen für Sicherheitslücken [10].

Darüber hinaus geht die Leistungsfähigkeit von C oft zu Lasten der Wartbarkeit und Entwicklungseffizienz [13]. Um maximale Durchsatzraten zu erzielen, sind in C häufig komplexe, manuelle Optimierungen notwendig. Costanzo et al. heben hervor, dass die Entwicklung von korrektem und effizientem C-Code im Vergleich zu Rust-Code einen signifikant höheren Programmieraufwand erfordert, insbesondere wenn komplexe Nebenläufigkeit umgesetzt werden soll [13]. Selbst die Autoren von ZMap sagen in ihrer Retrospektive explizit, dass sie für eine heutige Implementierung ihres Scanners Rust wählen würden, um die Wartbarkeit und Sicherheit der Codebasis langfristig zu gewährleisten [2].

Ein weiteres wesentlicher Nachteil bisheriger Hochleistungsansätze (wie `DPDK` oder `PF_RING`) ist ihre fehlende Integration in den Linux-*Mainline-Kernel*. Sie erfordern oft proprietäre Treiber oder Kernel-Module, die das Sicherheitssystem des Kernels umgehen und bei Updates zu Inkompatibilitäten führen können [2]. Durumeric et al. merken an, dass bei der

Einführung und Wartung von `PF_RING` für ZMap über die Jahre eine erhebliche Hürde darstellte [2]. `XDP` füllt diese Lücke, indem es *High-Performance*-Paketverarbeitung direkt im Kernel ermöglicht, ohne dessen Sicherheit und Kompatibilität zu kompromittieren.

3.5.2 Lösungsansatz

Die Nutzung von Rust stellt einen vielversprechenden Lösungsansatz dar, da sie Speichersicherheit bereits zur Kompilierzeit garantiert, in der Lage ist, eine mit C vergleichbare Geschwindigkeit zu erreichen, und durch sein striktes Typ- und Besitzmodell ganze Klassen von Fehlern (wie *Data Races*) eliminiert (siehe 2.4). Zusätzlich löst die Nutzung moderner Kernel-Technologien wie `XDP` und `eBPF` der Nachteil der fehlenden Kernelintegration für die Hochleistungspaketverarbeitung.

Die Kombination dieser Technologien und Werkzeuge stellt in dem Kontext des horizontalen High-Speed-Netzwerkscannings eine Forschungslücke dar, die in dieser Arbeit untersucht wird.

Kapitel 4: Anforderungsanalyse und Methodik

Dieses Kapitel definiert die funktionalen und nicht-funktionalen Anforderungen an den zu entwickelnden Portscanner, beschreibt das gewählte Vorgehensmodell zur Umsetzung in Rust und legt das Untersuchungsdesign für die anschließende Evaluation fest.

4.1 Anforderungsanalyse

4.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen definieren das Verhalten des Systems und die logischen Operationen, die der Scanner ausführen muss, um einen korrekten `SYN`-Scan durchzuführen.

- **/F-01/ Konstruktion valider TCP-`SYN`-Pakete:** Das System muss in der Lage sein, rohe TCP-Pakete so zu konstruieren, dass *IP*-Header und *TCP*-Header (inklusive `SYN`-Cookie) korrekt manuell gesetzt und die Prüfsummen valide berechnet werden, damit sie vom Zielsystem als legitime Verbindungsanfragen akzeptiert werden.
- **/F-02/ Senden von Paketen:** Das System muss in der Lage sein, die konstruierten TCP-Pakete über die Netzwerkschnittstelle an definierte Zielsysteme zu versenden.
- **/F-03/ Empfang von Paketen:** Das System muss in der Lage sein, eingehende Netzwerkpakete unabhängig vom Sendeprozess abzufangen und zur Auswertung bereitzustellen.
- **/F-04/ Zustandsloses Scanning:** Die Sende- und Empfangskomponenten dürfen keine statusbehaftete Kommunikation über die Zielsysteme austauschen. Die Zuordnung muss ausschließlich über Informationen im Paket-Header erfolgen.
- **/F-05/ Validierung eingehender Antworten:** Die Empfangskomponente muss eingehende `SYN-ACK`-Pakete validieren. Dafür muss der Hash-Werte des `SYN`-Cookies korrekt erstellt und mit dem aus der *Acknowledgement Number* extrahierten Wert verglichen werden.

- **/F-06/ Schließen der Verbindung:** Nach der Identifikation eines offenen Ports muss der Scanner ein `RST`-Paket senden, um die halboffene-Verbindung auf dem Zielsystem sauber zu beenden.

- **/F-07/ Endausgabe:** Es muss eine Endausgabe in einer Datei oder dem *Standard Output* geben in welcher die ausgewerteten Scanergebnisse bestehend aus IP-Adresse und Ziel-Port der offenen Zielsysteme enthalten sind.

- **/F-08/ Durchsatzlimitierung:** Das Programm muss in der Lage sein, eine angegebene Durchsatzrate (in Byte pro Sekunde) nicht zu überschreiten, sodass eine konsistente *Performance*-Messung möglich ist.

- **/F-9/ Eingabeschnittstelle:** Das Programm muss die zu scannenden Ziel-IP-Adressen aus dem *Standard Input* des Programmes entnehmen, um in die Infrastruktur des Unternehmens, welches diese Arbeit begleitet, zu passen.

4.1.2 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen stellen Qualitätsanforderungen dar und leiten sich primär aus technischen Randbedingungen und der Verwendung von Rust ab, welche sich aus dem Forschungsziel ergeben.

- **/NF-01/ Maximierung des Durchsatzes:** Das System soll in der Lage sein, die verfügbare Bandbreite einer Standard-Gigabit-Schnittstelle vollständig auszunutzen.

- **/NF-02/ Asynchrone Architektur:** Die Implementierung muss auf einem asynchronen Programmiermodell basieren, um durch nicht-blockierende I/O-Operationen eine hohe Nebenläufigkeit zu gewährleisten.

- **/NF-03/ Nutzung moderner Kernel-Mechanismen:** Zur Evaluation der Forschungsfrage müssen Linux-native Schnittstellen zur hochperformanten Paketverarbeitung wie `AF_XDP` oder `eBPF` verwendet werden.

- **/NF-04/ Speichersicherheit:** Die Implementierung soll die Sicherheitsgarantien von Rust wahren. `unsafe`-Blöcke können genutzt werden (z.B. in der Interaktion mit Kernel-APIs) sollten aber möglichst vermieden oder durch die Nutzung von externen Bibliotheken ersetzt werden, da diese intern `unsafe`-Blöcke häufig sicher kapseln [36].

- **/NF-05/ Minimale Ressourcennutzung:** Der CPU- und Arbeitsspeicherverbrauch soll im Verhältnis zum erzielten Durchsatz minimiert werden.

- **/NF-06/ Technologische Einschränkung:** Das Programm darf ausschließlich Technologien verwenden, die im Linux-Kernel-Ökosystem verfügbar sind, um Abhängigkeiten von Drittanbieter-Treibern zu vermeiden.

4.1.3 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen definieren Qualitätsmerkmale (wie Performance) sowie technische Randbedingungen, die sich aus dem Forschungsziel ergeben.

4.2 Vorgehensmodell der Entwicklung

Für die Realisierung wird ein evaluationsgetriebener, prototypischer Ansatz gewählt. Aufgrund der Komplexität asynchroner Netzwerkprogrammierung, sowie der Vielzahl möglicher Technologien, empfiehlt es sich, verschiedene Wege auszuprobieren. Dies hilft bei der Schaffung einer realistischen Vergleichsbasis zwischen den verschiedenen Technologien und bietet die Möglichkeit, Ausweichlösungen bei Problemen wie z. B. *Performance*-Engpässe zu finden oder die tatsächliche Notwendigkeit komplexer Technologien zu validieren. Die Entwicklung teilt sich in 2 Hauptphasen:

1. **Basisimplementierung:** In der Basisimplementierung wird ein vollumfänglicher `SYN`-Scanner implementiert. Dies soll zum einen als *Proof of Concept*, und zum anderen als erste Vergleichsgrundlage dienen. Für das Senden, werden bereits die Möglichkeit des Sendens mit `AF_PACKET` oder `AF_XDP` und jeweils auch in *Batches*[1] oder Einzelpaketen implementiert. Für das Empfangen wird der *Crate*[2] `pcap` genutzt, welcher das Empfangen von Paketen abstrahiert, allerdings nicht den Netzwerkstack des Linux-Kernels umgeht.

   Außerdem wird die für die nötige Struktur um die Komponenten zu vernetzen, sowie die Pakete korrekt zu erstellen und verarbeiten implementiert, sodass das Programm am Ende einen funktionierenden `SYN`-Scan vollführen kann. Zusätzlich wird auch hier schon auf die möglichst performante Umsetzung der gesamten Struktur, spezifischer Umsetzungen und der Wahl von *Crates* Wert gelegt.

2. **Optimierung des Empfangspfades durch Nutzung von `eBPF`:** Basierend auf den Messergebnissen der ersten Phase wird die Empfangskomponente hier grundsätzlich verändert, um eine hochperformante, sowie effizienten Paketempfang und Paketauswertung zu gewährleisten. Dafür wird ein `eBPF` in Verbindung mit einem `RingBuf` zur Protokollierung in Verbindung mit einem `XDP`-Programm statt des `pcap`-*Crates* genutzt. maybe TODO "bla bla pcap kann ab gewisser Geschwindigkeit nicht mithalten oder pcap ist zu ineffizient und/oder rst werden immer automatisch gesendet". Außerdem werden weitere, kleinere *Performance*-steigernde Maßnahmen addressiert.

[1] Gruppen von Paketen
[2] Bezeichnung für Bibliothek im Rust-Ökosystem

4.3 Untersuchungsdesign

Es wird im Folgenden erklärt, wie *Performance* im Kontext eines `SYN`-Scanners zu definieren ist. Anschließend werden die in dieser Arbeit zur Evaluation genutzten Metriken erklärt und daraufhin die Anforderungen an die Testumgebung vorgestellt und erläutert.

4.3.1 Metriken

Der Begriff „*Performance*-Effizienz“ wird gemäß der Norm *ISO/IEC 25010* als die Fähigkeit eines Produkts, seine Funktionen innerhalb festgelegter Zeit- und Durchsatz-Parameter zu erfüllen und dabei die Ressourcen unter den gegebenen Bedingungen effizient zu nutzen, verstanden [52].

Basierend auf der Definition werden folgende Metriken zur Quantifizierung herangezogen, wobei die Paketrate den Durchsatzparameter und die CPU-Auslastung, sowie RAM-Verbrauch die Ressourcennutzung darstellen:

- **/M-01/ Paketrate:** Die Paketrate wird in Pakete pro Sekunde *PPS* dargestellt und beschreibt die durchschnittliche Anzahl der erfolgreich an den Netzwerkadapter übergebenen Pakete pro Sekunde. Da die Scanner nur sehr kleine Pakete verschicken ist die Paketrate in *Performance*-orientierten Projekten dieser Art der limitierende Faktor **TODO**, weshalb sie maßgeblich als Metrik für die *Performance* dient.
- **/M-02/ CPU-Auslastung:** Die CPU-Auslastung von hochperformanten Netzwerkanwendungen findet maßgeblich im *User-Space*, im *Kernel-Space* und im *SoftIRQ*[3] statt **TODO**. deshalb sollte alle drei Bereiche betrachtet werden. Gemessen wird die prozentuale Auslastung der CPU-Kerne in Bezug auf diese Metriken.
- **/M-03/ RAM-Verbauch:** Der RAM-Verbrauch als zweiter primärer Teil der Ressourcenmetriken wird in Megabyte (MB) angegeben und stellt den Anteil des physisch durch den Scanner belegten Arbeitsspeicher dar.

4.3.2 Geplante Testszenarien

Um die in Abschnitt 4.1 definierten Anforderungen zu validieren, werden drei Testszenarien definiert:

1. **/S-01/ Anforderungsvalidierung:** Um die allgemeine Funktionsweise und Stabilität des Scanners zu validieren, soll ein Szenario mit einer künstlich limitierten, aber signifikanten Senderate durchgeführt werden. Ziel ist der Nachweis, dass der Scanner über einen längeren Zeitraum stabil arbeitet und die funktionalen Anforderungen

[3] TODO erklären

erfüllt. Dies dient auch als Referenzmessung, um sicherzustellen, dass funktionale Anforderungen in späteren Tests nicht explizit getestet werden müssen.

2. **/S-02/ Ermittlung der Performanzgrenzen:** In diesem Szenario wird jegliche künstliche Drosselung aufgehoben. Das Ziel ist es, die maximalen Durchsatzraten zu ermitteln. Hierbei wird geprüft, wie effizient die Ressourcen unter Volllast genutzt werden, um die nicht funktionalen Anforderungen zu untersuchen.

3. **/S-03/ Simulation unter realen Parametern und *Features*:** Um die Vergleichbarkeit zu praxisrelevanten Szenarien zu erhöhen, sollen Parameter gewählt werden, die für echte Internetscans typisch sind. Dies soll die *Performance*-Effizienz unter möglichst realen Bedingungen testen. Dabei wird auch ein Augenmerk auf die Nutzung von *Features* gelegt, die im Kontext eines realen Scans von Nutzen sind. Beispielsweise zur Verschleierung des Scans.

Kapitel 5: Konzeption und Implementierung

In diesem Kapitel wird zuerst das Konzept zur Erfüllung der Anforderungen vorgestellt. Anschließend wird die konkrete Umsetzung in den beiden Schritten der Basisimplementierung, sowie im Optimierungsschritt, sowie die Veränderungen, die im Laufe der Entwicklung stattgefunden haben, dargelegt und erklärt. Daraufhin werden die Schritte zur Qualitätssicherung, welche im Laufe der Entwicklung genutzt wurden aufgezeigt und schlussendlich der konkrete Aufbau für die finale Testumgebung beschrieben.

5.1 Projektstruktur Basisimplementierung

Das Rust-Projekt hat folgende Ordnerstruktur:

Codeauszug 5.1: Ordnerstruktur des SYN-Scanners (gekürzt)

```
/scanner
    /src
        /bin
            mock_programm.rs
        /scan utils
            /capturing_packets
                bucket.rs
                receiver.rs
            /emitting_packets
                assembler.rs
                rate_limiter.rs
                sender.rs
            /job_controlling
                parser_std_in.rs
                scan_job.rs
            /shared
                helper.rs
                types_and_config.rs
        main.rs
    Cargo.toml
/xdp-common
    /src
```

```
23         lib.rs
24 /xdp-ebpf
25     /src
26         main.rs
27 ...
```

In jedem Ordner ist eine `mod.rs` Datei zu finden, welche hier zugunsten der Lesbarkeit entfernt wurde. Diese Dateien dienen dazu, ein Verzeichnis als Rust Modul zu definieren und die darin genutzten Dateien für den Compiler sichtbar zu machen. Die `Cargo.toml` ist für die Verwaltung der externen Bibliotheken zuständig.

Die Verzeichnisse sind nach Aufgabenbereich gegliedert, um eine übersichtliche Gesamtstruktur zu haben und klar zeigen zu können, welches Verzeichnis für welche Aufgabe zuständig ist. Inhaltlich relevant sind vor allem `emitting_packets`, welches die Paketbearbeitung, das *Rate Limiting* und das Versenden übernimmt. Außerdem `mock_programm.rs`, in welchem die Paketrohlinge erstellt und alle Daten zur Konfiguration eingegeben werden. Für die Aufgabe des Empfangens und Auswerten der Antwortpakete sind zum einen `xdp-ebpf`, `xdp-common` und `capturing_packets` zuständig. Letzteres beinhaltet die Logik zum Empfangen der Daten im *User Space*, welche vom `XDP`-Programm übermittelt wurden und der darauffolgenden Duplikatsentfernung. Die Verzeichnisse `xdp-ebpf`, `xdp-common` beschreiben das `eBPF`-Programm, welches Antwortpakete abfängt, auswertet und nur die relevanten Informationen an den *User Space* weiterleitet. Die Komponente `job_controlling` ist für die Funktionsfähigkeit des Programmes auch essenziell, hat aber hauptsächlich die Aufgabe, die anderen Komponenten korrekt zu vernetzen.

5.1.1 Logische Komponenten des Scanners

Basierend auf dieser Struktur zeigt die Abbildung 5.1 die logischen Komponenten des `scanner`-Verzeichnisses und deren Interaktionen:

Das Verzeichnis `shared` ist dort nicht aufgeführt, da es lediglich der Steigerung der Übersichtlichkeit dient und helfende Funktionen, sowie Typenbeschreibungen enthält, die mehrfach im Projekt genutzt werden. Somit ist es für die logische Darstellung irrelevant. Es ist zudem wichtig zu erwähnen, dass auch bei Kontrollflüssen (z.B. Starten einer Komponente) einmalig Daten übertragen werden können. Datenflüsse hingegen stehen für einen mehrfach geschehenden Datenaustausch.

In dem Diagramm ist zu erkennen, das zwischen der `emitting_packets`-Komponente und der `capturing_packets`-Komponente kein Datenfluss besteht, sondern lediglich das Signal zum Beenden des Scans ausgetauscht wird. Daran ist das zustandslose Design zu erkennen, welches die Anforderung /F-04/ erfüllt.

In der Abbildung 5.1 wird der Weg der Pakete durch den Netzwerkkartentreiber und die Trennung der Zuständigkeiten von *User Space* und Linux-Kernel nicht explizit behandelt.

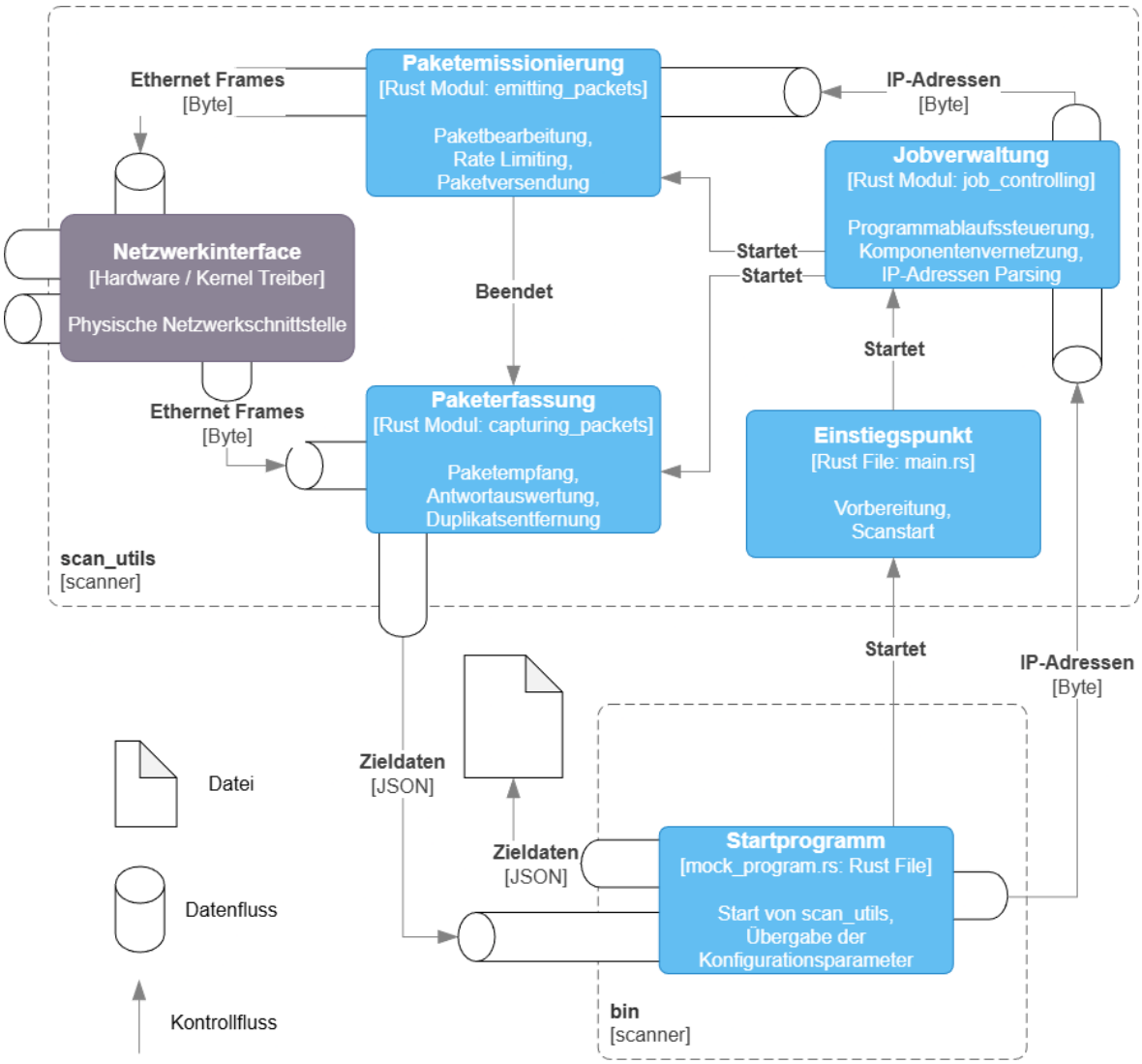

Abbildung 5.1: Diagramm logischer Komponenten des `scanner`-Verzeichnisses (vereinfacht)

Um nun aber die Funktion des `eBPF`-Programmes, welches in der Projektstruktur unter den Verzeichnissen `xdp-ebpf` und `xdp-common` zu finden ist, zu verbildlichen, wird in Abbildung 5.2 der Datenfluss zwischen Scanner-Programm und Netzwerkkarte verdeutlicht.

Das Diagramm zeigt mögliche Pfade, die ein Paket durchläuft, wenn es entweder gesendet oder empfangen wird. Dabei werden auch die unterschiedlichen XDP-Modi (siehe 2.3.4) beachtet. Im Sendeprozess (von der Anwendung zur Netzwerk-Hardware) wird mithilfe von `AF_XDP` der Netzwerk-Stack des Kernels je nach *Socket*-Konfiguration (*Copy* oder *Zero-Copy*) vollständig oder zum Großteil übersprungen. Die `AF_PACKET`-Variante hingegen durchläuft immer einige wenige Schritte des regulären Netzwerkstacks. Im Empfangsprozess ist zu sehen, dass das `eBPF`-Programm je nach Modus (*SKB Mode* oder *Driver Mode*) im Treiber der Netzwerkkarte oder direkt zu Beginn des Kernel-Netzwerkstacks ausgeführt wird. In beiden Fällen werden dort die eingehenden Pakete zuerst untersucht und je nachdem was das Ergebnis der Untersuchung ist, direkt verworfen, an den Netzwerkstack weitergeleitet, oder verändert und an den Treiber oder direkt an die Netzwerkkarte zum Versenden zurückgegeben. So werden alle, oder im Falle des *SKB Mode* fast alle, Schritte des regulären Netzwerkstacks eingespart. Die Ergebnisse der Untersuchung im `eBPF`-Programm werden bei validen Paketen in eine `eBPF Map`, in diesem Fall einem `RingBuf` geloggt. Dies hat den Vorteil, dass nur die relevanten Inhalte des Pakets (IP-Adresse, Port) statt des ganzen Paketes übermittelt werden müssen. Außerdem hat das *User Space*-Programm direkten Zugriff auf den `RingBuf` und kann die Daten somit ohne Umwege abgreifen.

Auf diese Art und Weise kann der SYN-Scanner die Verarbeitungsschritte sowohl beim Senden als auch beim Empfangen von Paketen auf ein Minimum reduzieren, was große Performanzchancen mit sich bringt.

5.1.2 Übersicht genutzter *Crates*

Für die Umsetzung der Komponenten sind folgende genutzte *Crates* aufgrund ihres Einflussreichtums hervorzuheben:

5.2 Implementierung und Funktionsweise der Komponenten

Um die Funktionsweise des Programmes im Detail zu erklären, werden im Folgenden die einzelnen Quelldateien vorgestellt und Besonderheiten bezüglich performanzsteigernden oder ressourcensparendenden Maßnahmen erläutert. Der Relevanz für das Projekt des SYN-Scanners entsprechend, werden manche Themen mehr und manche weniger ausführlich behandelt.

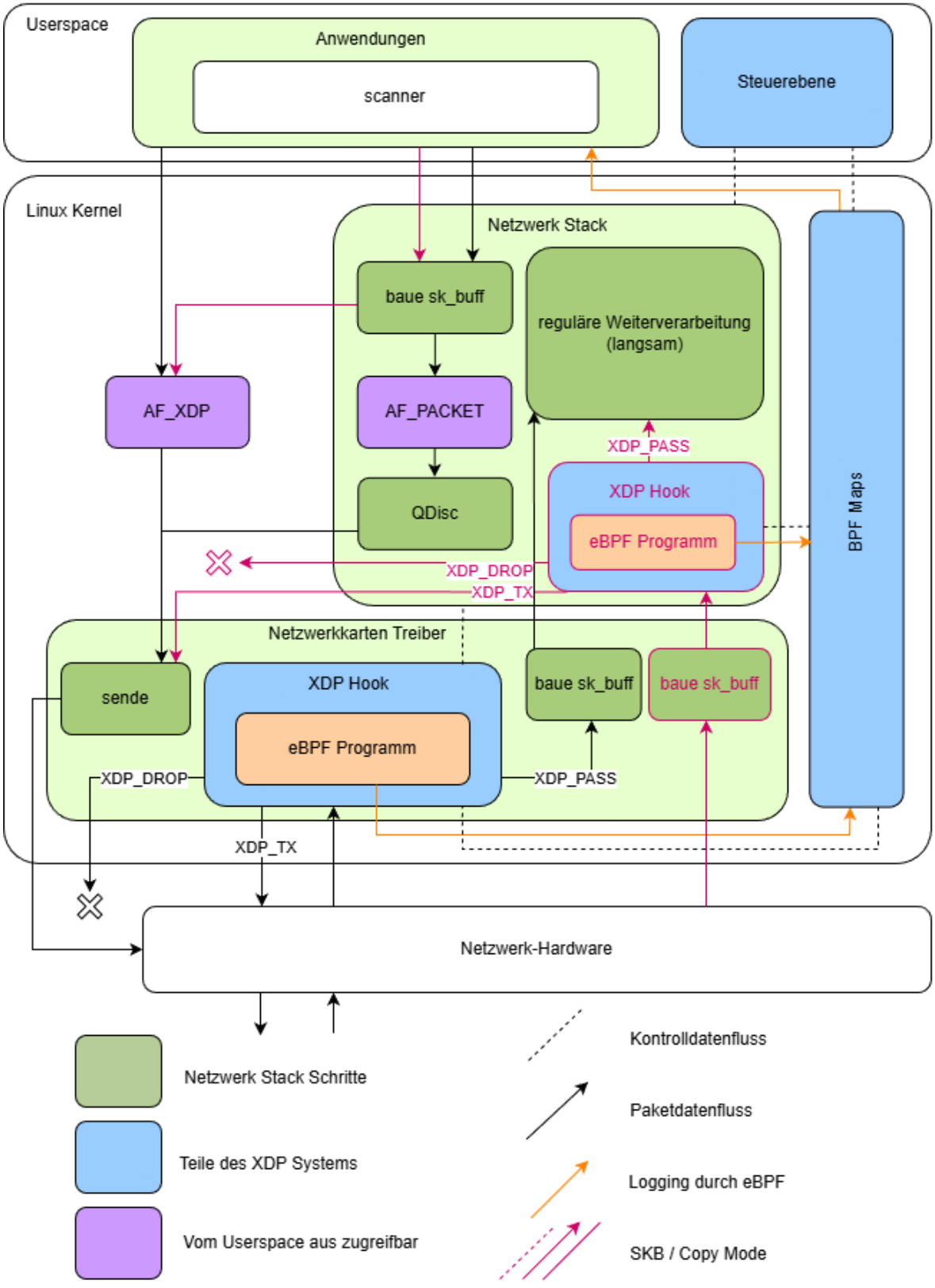

Abbildung 5.2: Weg der Pakete durch den Linux Kernel (vereinfacht)

[H]

Crate	Version	Nutzung
`tokio`	1.47.1	Nutzung für asynchrone Komponenten, Kommunikation über *Channels*, Parsen des *Standard Input* und Starten mehrerer asynchron laufender *Tasks*
`nix`	0.30.1	Erstellen der `AF_PACKET`-Schnittstelle und Versenden darüber
`xdp-socket`	0.1.4	Erstellen der `AF_XDP`-Schnittstelle und Versenden darüber
`aya`	0.13.1	Stellt Werkzeuge und Strukturen für die Erstellung und Nutzung von `eBPF`-Programmen zur Verfügung
`dashmap`	6.1.0	Stellt für asynchrone Nutzung optimierte *HashMaps* bereit und führt das *Locking* selbstständig durch
`pnet`	0.35.0	Nutzung als abstrahiertes Netzwerkwerkzeug zur Erstellung der Paket-*Templates*
`network_types`	0.1.0	Parsen der *Header*-Strukturen aus rohem Speicherbereich, ohne diese zu kopieren

Tabelle 5.1: Genutzte *Crates*

5.2.1 Paketemissionierung (`emitting_packets`)

Die Paketemissionierung umfasst den Prozess der Durchsatzlimitierung, der Paketbearbeitung und des Paketversandes inklusive den dafür benötigten Vorbereitungsschritten.

Effizienzsteigernde Maßnahmen

Um möglichst ressourcensparend zu arbeiten, werden die Ziel-IP-Adressen in *Batches*, also Ansammlungen von Paketen übertragen. Dies reduziert die Anzahl der benötigen *Context Switches* drastisch indem nicht für jede IP-Adresse ein eigener *System Call* [1] getätigt werden muss.

Rate Limiter (`rate_limiter.rs`)

Wie in der Abbildung 5.3 zu sehen, führt der *Rate Limiter* (`rate_limiter.rs`) dem Namen entsprechend die Funktion der Durchsatzlimitierung (Anforderung /F-08/) aus. Zuerst nimmt er die zu scannenden IP-Adressen vom *Parser* (`parser_std_in`) entgegen, bestimmt die Puffergröße anhand der in dieser Sekunde bereits gesendeten Datenmenge (TODO wahlweise kleinen Flowchart), füllt einen Puffer und erstellt für jeden Puffer einen `tokio` *Task* mit einem *Assembler* (`assembler.rs`). Wenn der *Parser* alle IP-Adressen geparst hat, und der *Rate Limiter* alle verarbeitet hat, wird der gleiche Prozess für die restlichen Zielports durchgeführt, mit dem entscheidenden Unterschied, dass nun auf den internen

[1] TODO

Puffer an IP-Adressen, welche zuvor gespeichert wurden zugegriffen wird, was den CPU-Verbrauch potenziell verringert, da die Adressen nun nicht mehr geparst und weitergeleitet werden müssen.

`tokio` *Tasks* oder auch *Green-Threads* sind kleine Ausführungseinheiten, ähnlich eines Betriebssystem-*Threads*, bloß dass diese durch die `tokio`-eigene Laufzeitumgebung verwaltet werden. Sie sind sehr leichtgewichtig, da sie keine *Context Switches* benötigen und erlauben asynchrone Ausführung mehrerer *Tasks*, da sie, statt wie Betriebssystem-*Threads* zu blockieren, die Ressourcen für andere *Tasks* freigeben und somit Nebenläufigkeit ermöglichen [53]. Diese Nebenläufigkeit wird hier genutzt, um entsprechend der aktuellen Senderate *Assembler* zu erzeugen, die nicht den gesamten Betriebssystem-*Thread* blockieren, wenn die Pakete eines *Assemblers* nicht zuerst vom *Sender* entgegengenommen werden. Stattdessen wartet jeder *Assembler*, ohne andere Teile der Software zu beeinträchtigen. So wird sicher gestellt, dass immer genügend Pakete für den *Sender* bereitstehen. Die Puffergröße eines *Assemblers* wird bei Beginn des Programmes abhängig von der Durchsatzlimitierung und der *Batch*-Größe rechnerisch ermittelt

Assembler (`assembler.rs`)

Die Rolle des *Assemblers* ist recht simpel: Jeder *Assembler* iteriert über die ihm verfügbaren IP-Adressen, füllt *Templates* mit der Ziel-IP-Adresse, dem Ziel-Port, sowie der *Sequence Number* und berechnet die Checksummen des *IP-* und *TCP-Header* neu. Dies dient zur Erfüllung der Anforderung /F-01/. Die *Sequence Number* wird wie folgt berechnet:

$$\text{ISN} = \text{SipHash}_K(\texttt{src\_ip}, \texttt{dst\_ip}, \texttt{src\_port}, \texttt{dst\_port}) \tag{5.1}$$

wobei:

- **ISN:** die berechnete 32-Bit initiale *Sequence Number* (SYN-Cookie).
- **K:** ein geheimer, zufälliger 128-Bit Schlüssel, der beim Start des Scanners generiert wird.
- **`src_ip`, `dst_ip`:** die Quell- und Ziel-IP-Adressen der Verbindung.
- **`src_port`, `dst_port`:** die zugehörigen TCP-Quell- und Ziel-Ports.

Die Pseudozufallsfunktion SipHash eignet sich hervorragend, da sie speziell für hohe *Performance* bei kurzen Eingabedaten entwickelt wurde, aber einer *Hashing*-Funktion entsprechend bei gleichem Input immer den gleichen Wert zurückgibt [54]. Damit dies konsistent funktioniert, muss allerdings ein geheimer Schlüssel genutzt werden, welcher der Paketemissionierungs- sowie der Paketerfassungskomponente bekannt ist. In den *Templates* sind die restlichen Werte bereits vorhanden. Die Änderungen werden direkt auf Byte-Ebene umgesetzt, da die Feldzuweisungen der *Header*-Felder immer gleich sind [22] [21]. Somit

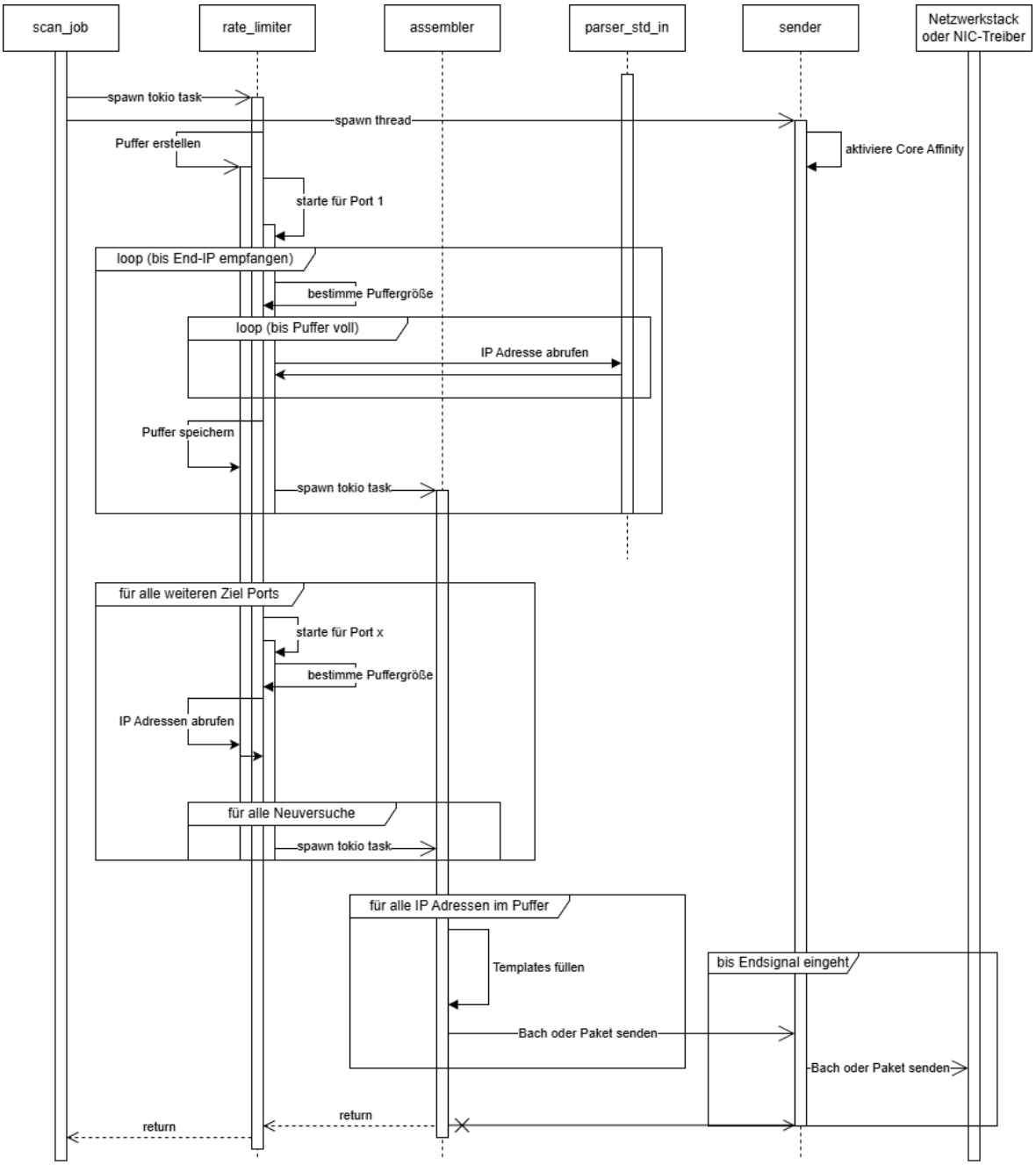

Abbildung 5.3: Ablaufes und Funktionsweise der `emitting_packets`-Komponente (vereinfacht)

können vollständige Pakete in sehr wenigen Schritten und ohne aufwendiges *Parsing* oder gar kompletter Neuerstellung genutzt werden. Diese Pakete werden anschließend je nach Konfiguration einzeln oder in *Batches* an den *Sender* weitergeleitet.

Sender (`sender.rs`)

Der *Sender* agiert im Kontrast zu den anderen Subkomponenten in einem eigenen Betriebssystem-*Thread*. Das hat den Grund, dass er somit die komplette Kapazität des *Threads* alleine ausnutzen kann und bezüglich CPU-Auslastung möglichst wenig mit anderen Prozessen konkurrieren soll, um möglichst performant zu sein. Um diesen Effekt zu verstärken wird außerdem der `core_affinity` *Crate* genutzt (siehe 5.1). Der Sender läuft in einer ständigen Schleife bis die *Channels* zum Erhalt der Pakete geschlossen werden. Je nach Konfiguration sendet er *Batches* oder einzelne Pakete über die jeweilige Schnittstelle. Die *Socket*-Schnittstelle welche zum Versenden und somit zur Erfüllung der Anforderung /F-02/ verwendet wird, wird beim Start des Senders initialisiert.

5.2.2 Ergebnisverarbeitung (`capturing_packets`)

In der Ergebnisverarbeitung werden die durch den `eBPF` vorgeprüften Daten der validen Antworten entgegengenommen und einer Duplikatsprüfung unterzogen. Anschließend werden die endgültig korrekten Ergebnisse ausgegeben.

Receiver (`receiver.rs`)

In früheren Iterationen des Programmes lief der *Receiver* ebenso wie der *Sender* in einem eigenen Betriebssystem-*Thread*, um möglichst viel Leistung nutzen zu können und *Context Switches* zu vermeiden. Die Nutzung von `pcap` stellte die abstrahierte Netzwerkschnittstelle zur Erfüllung der Anforderung /F-03/ dar. Mit `pcap` muss sich der Programmierer nicht manuell um *Sockets* oder der Kommunikation mit dem Netzwerk-Stack kümmern. Ein weiterer Vorteil ist die einfache Nutzung eines *Berkley Packet Filters* (BPF), mit welchem man Pakete an einem frühen Zeitpunkt im Netzwerk-Stack filtern kann. Wenn nun ein Paket empfangen wurde, wurden dessen *Header*-Felder mit `etherparse`, einer Ethernet-*Parsing*-Bibliothek extrahiert und analog zum aktuellen Vorgehen auf Duplikate geprüft.

Obwohl die Handhabung mit `pcap` entwicklerfreundlich ist, wurde es letztendlich durch den `eBPF`-Ansatz verdrängt, da die *Performance* in ersten Tests nicht den Ansprüchen dieses Projektes genügte. Dies ist darauf zurückzuführen, dass `pcap` intern *Raw-Sockets* mit `AF_INET` nutzt, welches im Vergleich zu `AF_PACKET` oder `AF_XDP` deutlich mehr Schritte im Netzwerk-Stack durchlaufen muss (siehe 5.2), selbst, wenn durch den BPF irrelevante Pakete hardwarenah herausgefiltert werden.

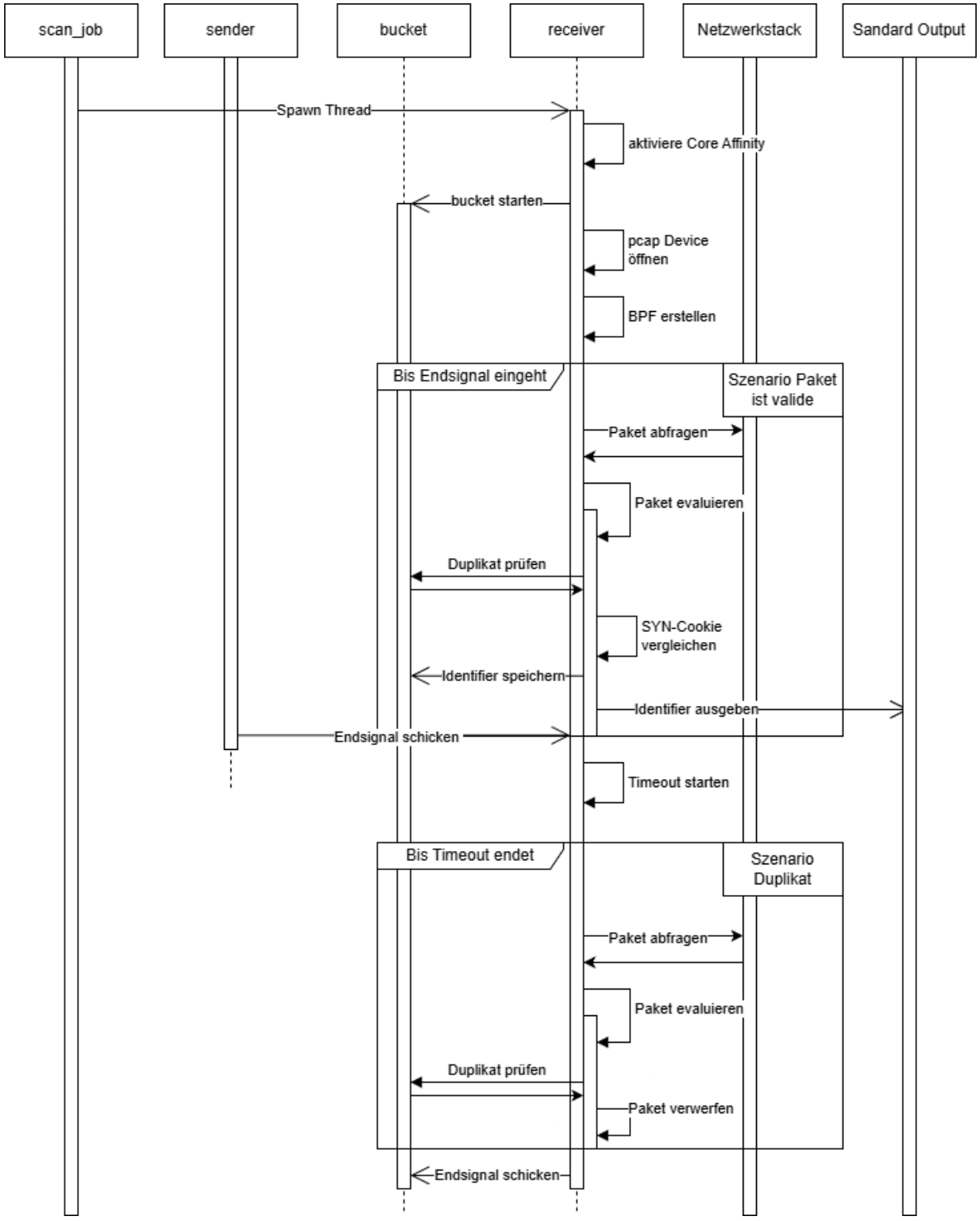

Abbildung 5.4: Exemplarisches Diagramm zur Funktionsweise der `capturing_packets`-Komponente (vereinfacht)

Im aktuellen Ansatz wird der *Receiver* stattdessen in einem `tokio` *Task* erstellt, um Asynchronität zu gewährleisten. Außerdem dient er nun ausschließlich dem Empfang, der durch das `eBPF`-Programm über den `RingBuf` geloggten Daten, der Verwaltung der Duplikaterkennung und der Ausgabe valider Daten. Im ersten Schritt werden Daten aus dem `RingBuf` abgerufen. Da der Zugriff auf den `RingBuf` über einen Unix-Dateideskriptor erfolgt, dessen Leseoperationen standardmäßig den *Thread* blockieren, muss dieser in das asynchrone Modell der Anwendung integriert werden. Dafür bietet `tokio` eine Lösung, welche es ermöglicht, durchgehend auf neue Pakete zu warten, ohne den ausführenden *Thread* zu blockieren. Anschließend wird die Ziel-IP-Port-Kombination der Duplikatprüfung, welche im nächsten Absatz genauer beschrieben wird, unterzogen. Sollte es sich um ein Duplikat handeln, werden die Daten verworfen, ansonsten werden sie in den *Standard Output* geschrieben. Somit wird Anforderung /F-07/ erfüllt.

Bucket (`bucket.rs`)

Zur Duplikaterkennung wird ein *Timed Bucket System* genutzt, in welchem mehrere *Buckets* (*HashMaps*) als Zwischenspeicher für die bisherigen Antworten dienen. Es kann nur in den derzeit aktiven *Bucket* geschrieben werden, doch aus allen wird gelesen. Nach einer festen Zeiteinheit wird der nächste *Bucket* aktiv und der am längsten inaktive geleert. Durch die Aufteilung in mehrere *Buckets* sollen starke Auslastungshöhepunkte durch das Leeren einer sehr aufgeblähten *HashMap* verhindert werden. Außerdem werden dadurch längere *Locking*-Zeiten bei asynchronen Schreib- und Lesezugriffen vermieden. Die Suche nach Duplikaten gestaltet sich dabei recht schnell, da *HashMaps* eine Suche der Zeitkomplexität $O(1)$ ermöglichen. Um das *Locking* zu verwalten wurde auf *DashMaps* aus dem `dashmap` *Crate* zurückgegriffen, welche für den asynchronen Einsatz optimiert wurden.

Die IP-Adresse und Ziel-Port des Zielsystems der validen Antwort wird entsprechend Anforderung /F-07/ nach der Duplikatsbereinigung in den *Standard Output* geschrieben. Dieser wird vom Mock-Programm (`mock_program.rs`) in eine Datei weitergeleitet.

Um die Umsetzung der Anforderung /F-06/ muss sich nicht explizit gekümmert werden, da der Linux-Netzwerk-Stack bei Erhalt einer Antwort nach einer gespeicherten Verbindung zu dieser Anfrage sucht, anschließend merkt, dass keine vorhanden ist, da das SYN-Paket über einen *Raw-Socket* verschickt wurde und automatisch eine `RST`-Antwort zurückschickt

5.2.3 Programmstart und Jobverwaltung (`job_controlling`)

Der Start des Programmes, die Konfiguration sowie das Starten und Verbinden der einzelnen Komponenten geht von den in dieser Sektion beschriebenen Komponenten aus. Des Weiteren übernehmen diese auch das *Parsing* und die Weiterleitung der Ziel-IP-Adressen zur `emitting_packets`-Komponente.

Startprogramm (`mock_program.rs`)

Um die Anforderung /F-09/ zu erfüllen, nimmt der Scanner die IP-Adressen der Ziele über den *Standard Input* entgegen. Für die Evaluation in dieser Arbeit wurde, um die Vergleichbarkeit herzustellen ein Programm erstellt, welches die Aufgabe des Startens des Scanners, die Erstellung der *Ethernet-Templates*, das Schreiben der Daten in den *Standard Input* und das Lesen aus dem *Standard Output* des Scanners übernimmt. Dort werden auch die Konfigurationsparameter eingetragen.

Die *Ethernet-Templates*, welche das Fundament zur Erfüllung der Anforderung /F-01/ darstellen, werden mithilfe des `pnet` *Crates* erstellt, da dieser eine entwicklerfreundliche Schnittstelle dafür bereitstellt. Dort werden alle Parameter für eine reguläres `TCP-SYN`-Paket bis auf die Ziel-IP, den Ziel-Port und die *Sequence Number* gesetzt. Es wird für jede Quell-IP ein *Template* angelegt, um die Streuung der Paketquellen zur Verschleierung des Scans und somit die Trefferrate zu erhöhen.

Einstiegspunkt (`main.rs`)

Die `main.rs`-Datei dient als Einstiegspunkt und Startfunktion des SYN-Scanners. Dort wird mithilfe des `aya` *Crates* das `eBPF`-Programm geladen und die `eBPF Maps` initialisiert. Des Weiteren werden die Konfigurationsparameter erfasst und letztendlich ein *Scanjob* mit allen benötigten Informationen gestartet.

Eine ressourcensparende Maßnahme ist außerdem die Nutzung eines alternativen *Allocators* [2], welcher in der `main.rs`-Datei definiert wird. Anstelle des *System Allocators*, welcher ein breites Anwendungsfeld bedient, bietet der `jemallocator` einen Fokus auf Nebenläufigkeit und Skalierbarkeit auf Multiprozessorsystemen **jemalloc**. Das wird dadurch erreicht, dass viermal mehr der sogenannten Arenen als verfügbare Prozessoren erstellt und die *Threads* darunter aufgeteilt werden. *Threads* blockieren sich gegenseitig nur, wenn sie in der gleichen Arena sind, was durch die Vielzahl deutlich seltener vorkommt. Außerdem ist die Allokierung von Speicher durch die Nutzung fester *Size Classes* für sehr viele kleine Allokationen, wie sie in dem Scanner zum Beispiel bei der Vielzahl an *Batches* von IP-Adressen oder Pakete geschieht, effizienter, als die Vorgehensweise des *System Allocators*, welcher die genaue Speichergröße sucht **TODO**.

Standard Input Parser (`parser_std_in`)

Der *Parser* parst zum einen die Konfigurationsparameter aus dem *Standard Input* und zum anderen die Ziel-IP-Adressen. Um möglichst performant und ressourcensparend vorzugehen, wird im Allgemein ein Fokus auf die Vermeidung neuer Allokationen und die Daten, falls möglich, mit *Zero-Copy*-Operationen zu verarbeiten. Deshalb geschieht das

[2] TODO

Entgegennehmen, Deserialisieren und Weiterleiten der Adressen in *Batches*. Dies minimiert unter anderem die Anzahl der *Context Switches* durch die Reduzierung von *System Calls*.

Der folgende Codeauszug 5.2 zeigt das *Parsing* der Daten welche im Binärformat übertragen wurden. Um das Verarbeiten unvollständiger IP-Adressen zu vermeiden, wird sich nach jeder Iteration ein *Offset* gemerkt, sodass keine Daten verloren gehen, falls ein *Batch* nicht vollständig gefüllt oder sauber am Ende einer Adresse endet.

Codeauszug 5.2: Binärformat-*Parsing* im *Standard-Input Parser*

```
const BATCH_SIZE: usize = 8192; // 8KB chunks -> 2048 IPv4-Adressen
let mut buffer = [0u8; BATCH_SIZE];
let mut offset = 0;
let mut ip_batch: Vec<[u8; 4]> = Vec::with_capacity(BATCH_SIZE / 4);

loop {
    let read_len = reader.read(&mut buffer[offset..]).await?;

    if read_len == 0 { /* EOF handling */ }

    let valid_data_len = offset + read_len;
    let mut cursor = 0;

    // Verarbeite vollstaendige IP-Pakete
    while cursor + 4 <= valid_data_len {
        let bytes: [u8; 4] = buffer[cursor..cursor + 4].try_into().unwrap();
        cursor += 4;

        if bytes == [0, 0, 0, 0] { /* Terminator -> return */ }

        ip_batch.push(bytes);
    }

    // Sende akkumulierten Batch
    if !ip_batch.is_empty() {
        let batch = std::mem::replace(&mut ip_batch, ↩
            Vec::with_capacity(BATCH_SIZE / 4));
        sender.send(batch).await?;
    }

    // Kopiere fragmentierte Bytes an Puffer-Anfang
    let remaining = valid_data_len - cursor;
    if remaining > 0 {
        buffer.copy_within(cursor..valid_data_len, 0);
    }
    offset = remaining;
}
```

Im bereitgestellten Codeauszug wurden explizit die in 5.2 beschriebenen Performanz-steigernden Maßnahmen umgesetzt.

Maßnahme	Beschreibung
`Vec::with_capacity()`	Der IP-Puffer wird in einen Puffer mit fester Größe gelesen, welcher somit nur einmal allokiert werden muss.
`std::mem::replace`	Ermöglicht das Tauschen des `ip_batch`-Puffers gegen einen neu erstellten, leeren Puffer, ohne dass die Inhalte kopiert werden müssen, da lediglich die *Ownership* gewechselt wird.
Asynchroner `tokio`-*Reader*	Die Nutzung des asynchronen `tokio`-*Readers* ermöglicht, dass auch bei vollem *Channel* der *Parser* nie andere Prozesse blockiert.

Tabelle 5.2: Performanz-steigernde Maßnahmen im *Standard-Input Parser*

Scanjob (`scanjob.rs`)

Ein *Scanjob* beinhaltet alle *User Space*-Komponenten die für den SYN-Scan benötigt werden. Er startet diese und vernetzt sie mithilfe von `tokio` *Channels*. Beim Starten der Komponenten werden alle benötigten Informationen übergeben.

Da das Anheften eines neuen `XDP`-Programmes einen Neustart des Netzwerkkartentreibers erfordert, wird vor der Erstellung der Sende-*Sockets* eine kurze Zeit gewartet. Auch nach dem Starten des *Senders* und des *Receivers* wird kurz gewartet, damit alles auf Abruf ist, sobald der *Rate Limiter* mit der Produktion der `SYN`-Pakete beginnt. Dies dient der allgemeinen Stabilität des Programmes. Wie bereits in den meisten anderen Komponenten wird auch hier `tokio` zur konkurrenten Ausführung der verschiedenen Programmteile genutzt. Dies ermöglicht das unabhängige Handeln der einzelnen Bestandteile, was einerseits der Notwendigkeit des gleichzeitigen Sendens und Empfangens entspringt und andererseits der *Performance*-Steigerung durch Einsparung von Wartezeiten dient.

5.3 eBPF

Der `eBPF` dient dem Scanner als Empfangspunkt für eingehende Pakete. Je nachdem in welchem `XDP`-Modus das Programm ausgeführt wird, agiert dieses direkt im Treiber der Netzwerkkarte oder an der ersten Stelle nach Erstellung eines Puffers im Netzwerkstack. Dadurch können die Pakete bereits am frühstmöglichen Punkt evaluiert werden, dessen relevante Daten extrahiert und direkt ohne Umweg über den geteilten Speicher der `eBPF Maps` in das *User Space*-Programm übertragen werden (siehe 5.2). Dies führt zu einer massiven Ressourceneinsparung und das wiederum zu einer Zeiteinsparung, da etliche Zwischenschritte, welche normalerweise durchlaufen werden müssten, um das Paket durch den Netzwerkstack zum *User Space* zu leiten und das Wiederversenden ohne Kopieraufwand

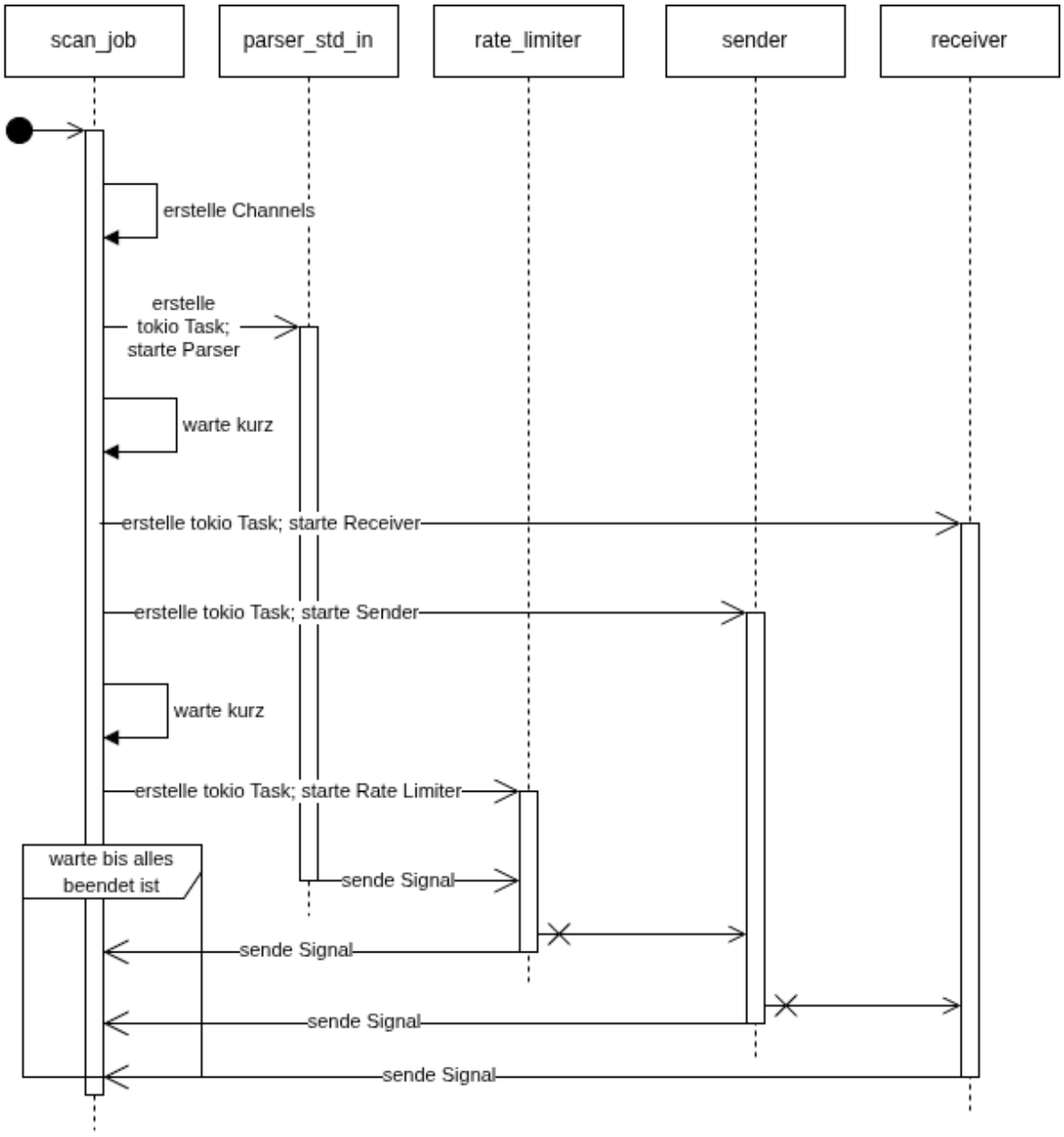

Abbildung 5.5: Funktionsweise der `scan_job.rs` Datei (vereinfacht)

oder *Context Switches* passiert. Durch die `XDP_TX`-Funktion können `RST`-Pakete direkt im `eBPF`-Programm erstellt und wieder verschickt werden, sodass der Netzwerkstack sowie *User Space* komplett vermieden werden.

5.3.1 `XDP`-Programm

Um ein `eBPF`-Programm nutzen zu können wird es in die `XDP`-*Hook* des Kernels geladen und zuvor korrekt in `ELF`-Dateien übersetzt, damit die `XDP`-*Hook* das Programm akzeptiert. Das Übersetzen des Rust-Codes übernimmt der `aya` *Crate* und das Anhängen des Programmes passiert in `main.rs`, indem die durch `aya` übersetzten Dateien an das genutzte Netzwerkinterface gebunden wird. Auch das Laden der `eBPF Maps` wird durch `aya` übernommen und in der `main.rs` über die Rust-Schnittstelle dargestellt. Auch wenn immer noch ein gewisses Maß an Komplexität besteht, erleichtert der *Crate* den Implementierungsaufwand dadurch enorm. Die zum Informationsaustausch zwischen `eBPF`-Programm und *User Space*-Programm genutzten `eBPF Maps` werden in 5.3 aufgeführt.

Name	Typ	Nutzung
`STATS`	`PerCpuArray<u64>`	Effiziente, lockfreie Protokollierung von Statistiken pro CPU [55]
`WHITELIST_IPV4`	`HashMap<[u8; 4], u8>`	*HashMap* zum Abgleich der für den Scan genutzten Quell-IP-Adressen
`EVENTS`	`RingBuf`	Effizienter, geteilter Puffer-Ring [56] zur Übermittlung der extrahierten Zielinformationen valider Pakete an den *Receiver*
`SIPHASH_KEY`	`Array<u64>`	Schlüssel zur korrekten Auswertung des *SYN-Cookies*

Tabelle 5.3: Genutzte `eBPF Maps`

5.3.2 Funktionsweise

Wie in Abbildung 5.6 zu sehen, wird aus allen empfangenen Paketen zuerst der *Ethernet-*, *IP-* und *TCP-Header* extrahiert und überprüft. Sollte dabei festgestellt werden, dass ein Paket kein valides `IPv4-SYN-ACK`-Paket darstellt, wird es per `XDP_PASS` an den normalen Netzwerkstack des Kernels weitergeleitet. Auch in der anschließenden Prüfung, ob die, da es sich in dem Fall um eine Antwort handelt, Ziel-IP-Adresse in der `WHITELIST_IPV4` vertreten ist oder die Prüfung des *SYN-Cookies* führt bei Fehlschlag zu einem `XDP_PASS`. Das bietet den Vorteil, dass der reguläre Netzwerkverkehr trotz Nutzung des SYN-Scanners unbeeinträchtigt ist. Um den *SYN-Cookie* zu vergleichen wird die Hash-Berechnung mithilfe des *Sip-Hashing*-Algorithmus und des per `SIPHASH_KEY` übergegeben Schlüssels durchgeführt. Die vier in 5.2.1 genannten Werte werden wie in 2.2.1 beschrieben verrechnet und ausgewertet, um die Anforderung /F-05/ zu erfüllen.

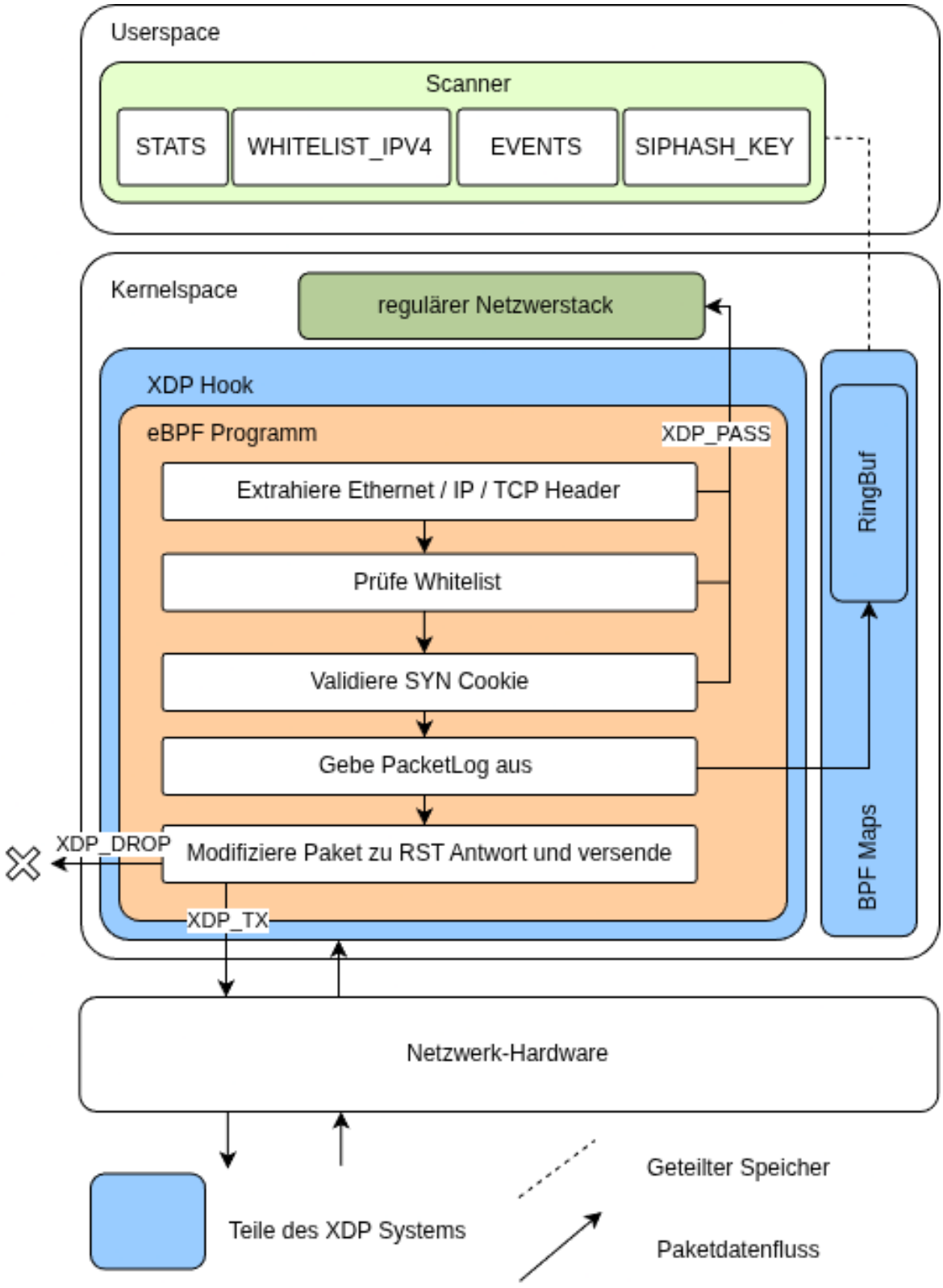

Abbildung 5.6: Funktionsweise des eBPF-Programmes (vereinfacht)

In dem eBPF-Programm können aus mehreren Gründen keine Funktionen der Standardbibliothek von Rust genutzt werden. Zum Beispiel würden viele Funktionen schlichtweg nicht funktionieren, da sie per *System Call* auf Betriebssystemdiensten beruhen die hier nicht verfügbar sind, weil direkt im Kernel gearbeitet wird. Dynamische Speicherstrukturen wie `Vec<T>`, `String`, oder *HashMap* können nicht genutzt werden, da kein Zugriff auf den allgemeinen *System Allocator* besteht. Im Allgemeinen ist der eBPF-*Verifier* sehr restriktiv, weshalb im eBPF-Programm nur wenige Bibliotheken nutzbar sind. In allen bereits beschriebenen Schritten des eBPFs wurde deshalb mit dem `network_types` *Crate* gearbeitet. Dieser nutzt keine Technologien der Standardbibliothek und erlaubt es, die rohen Speicherbereiche abstrahiert darzustellen und Zeiger auf die Strukturen der *Header* mit leichter verständlichen *Enums* darzustellen. So können die gewünschten Paketstrukturen gelesen und verändert werden, ohne, dass neuer Speicher allokiert werden muss.

Die Vermeidung neuer Speicherallokation wird unter anderem durch die Nutzung der `ptr_at`-Funktion umgesetzt. Die `ptr_at`-Funktion dient der Navigation durch den per `ctx` übergebenen Speicherbereich, indem ein *Offset* übergeben und *Pointer* vom Punkt des *Offsets*, des Endes des Speicherbereichs und die Länge des Bereichs zurückgegeben werden.

Codeauszug 5.3: `ptr_at`-Funktion zum Navigieren durch Speicherbereiche

```
1  #[inline(always)]
2  unsafe fn ptr_at<T>(ctx: &XdpContext, offset: usize) -> Result<*const T, ↩
       ()> {
3      let start = ctx.data();
4      let end = ctx.data_end();
5      let len = mem::size_of::<T>();
6      if start + offset + len > end {
7          /* Error handling */
8      }
9      Ok((start + offset) as *const T)
10 }
```

So wird beispielsweise im folgenden Beispiel der Speicherbereich des *IP-Header* extrahiert, indem der *Offset* eines *Ethernet-Header* (16 Byte) übergeben wird.

Codeauszug 5.4: Extraktion des Speicherbereichs des *IP-Header*

```
1      // IPv4 Header
2      let ip: *mut Ipv4Hdr = match unsafe { ptr_at_mut(ctx, EthHdr::LEN) } {
3          Ok(p) => p,
4          Err(_) => {
5              /* Error handling */
6          }
7      };
```

Dabei ist zu beachten, dass diese Aufrufe von einem `unsafe`-Block umschlossen sind. Da

dieser Speicher vom Linux-Kernel und nicht von der Rust-Runtime verwaltet wird, kann Rust nicht sicherstellen, dass die Rust-typischen Sicherheitsgarantien gewährleistet sind. Diese Entscheidung wurde bewusst getroffen, da diese Arbeit performance-orientierte Implementierungen behandelt, auch wenn dadurch Sicherheitsgarantien von Rust umgangen werden. Durch die besonderen Umstände im Kontext eines `eBPF`-Programmes mit strengem *Verifier* sind die Sicherheitskonzepte kaum umsetzbar, wenn *Zero-Copy* angestrebt wird. Dies hängt damit zusammen, dass Rust normalerweise mit einem `panic!` reagiert, sollte beispielsweise auf einen falschen Index eines *Slices* (z.B. `&[u8]`) zugegriffen werden. Dies ist im Kernel-Kontext nicht erlaubt. Stattdessen bleibt die Möglichkeit eine Kopie des Speicherbereichs anzufertigen, so wie es beispielsweise der `etherparse` *Crate* tut, um die Struktur dann in einem sicheren Kontext zu verwalten. Dies mindert aber die *Performance* deutlich und ist somit für performanz-orientiere Anwendungen, wie die hier implementierte, nicht die präferierte Lösung.

Wenn sich ein Paket als valide Antwort herausstellt, werden die Zieldaten über den `RingBuf` an den *User Space* weitergeleitet. Die Daten werden in der in `xdp_common` definierten Struktur namens `PacketLog` übertragen, welche die gescannte Adresse und den gescannten Port beinhaltet. Das `xdp_common` Verzeichnis dient lediglich der Definition dieser Struktur und dazugehörigen Getter-Funktionen.

Das anschließende Erstellen und Versenden des `RST`-Paketes dient einerseits dazu, die Verbindung beim Zielsystem korrekt zu schließen und somit einen normalen TCP-Aushandlungsprozess zu simulieren, andererseits hindert es das Zielsystem am Senden weiterer `SYN-ACK`-Antworten sowie dem Aufrechterhalten eines Verbindungsstatuses. Um die *Performance* zu erhöhen wird auch hier ein *Zero-Copy*-Ansatz gewählt, indem das ursprüngliche Paket durch das Tauschen entsprechender Werte und das Neuberechnen einiger Werte umgewandelt wird. Die Veränderungen der Felder sind in Abbildung 5.3.2 beschrieben.

Das fertige Paket wird anschließend per `XDP_TX` direkt in den Puffer der Netzwerkkarte geschrieben und durch die Modifikationen als valides `RST`-Paket an den *Sender* der ursprünglichen `SYN-ACK`-Antwort zurückgeschickt. Dies dient der Erfüllung der Anforderung /F-06/.

5.4 Qualitätssicherung

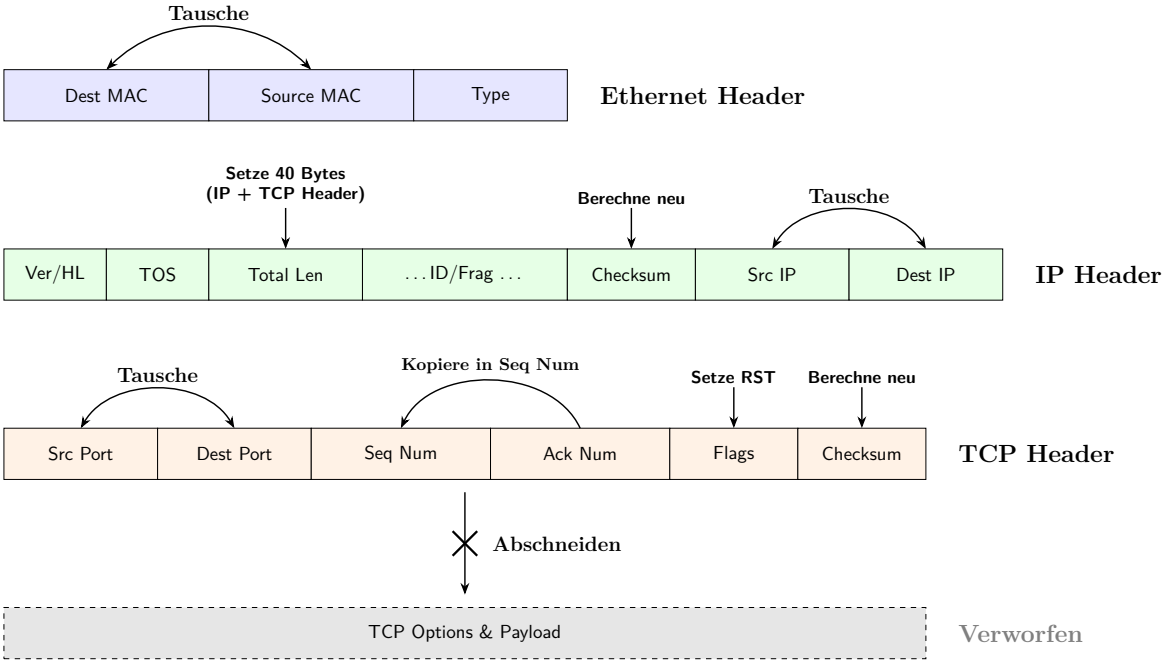

Abbildung 5.7: *In-Memory*-Modifikation des `SYN-ACK`-Paketes zum `RST`-Paket

Kapitel 6: Testumgebung und Durchführung

6.1 Versuchsaufbau

6.1.1 Hardware-Spezifikation

Für die Umsetzung des Testes wurden die in 6.1 spezifizierten Systeme verwendet. Es wurden keine Veränderungen vorgenommen, die das Betriebssystem von seinem nativen Zustand abbringen, um Anforderung /NF-06/ zu genügen. Die einzige Konfigurationsmaßnahme außerhalb des Programmes ist eine softwareseitige Anpassung der Sende-, sowie Empfangsringe der Netzwerkkarte auf das Maximum der bereitgestellten Hardware. Dies ist in diesem Szenario zu empfehlen, da es sich explizit um ein Hochleistungsszenario handelt. Das Anpassen der genutzten Puffer dient dem Abfangen von Lastspitzen, indem mehr Pakete vom Netzwerkkartentreiber gepuffert werden können. Für die genutzten Netzwerkkarten bedeutet dies konkret eine Erhöhung der sogenannten Deskriptoren von 256 auf 4096 (siehe **Anhang A.1**).

Komponente	**Scanner-Knoten**	**Ziel-Knoten**
	Hardware	
CPU	Intel Core i5-11400F (6 Kerne, 2.6 GHz)	Intel Core i5-7500 (4 Kerne, 3.4 GHz)
RAM	48 GB DDR4 (2667 MHz)	8 GB DDR4 (2400 MHz)
Netzwerkkarte	Intel I210-T1 (Gbit)	Intel I350-T2 (Gbit)
Verbindung	Direktverbindung via CAT 6 S/FTP Kabel (Gbit)	
	Software	
Betriebssystem	Ubuntu 24.04.3 LTS	Ubuntu 24.04.3 LTS
Kernel	6.14.0-37-generic	6.14.0-37-generic
Treiber	igb (6.14.0-37)	igb (6.14.0-37)

Tabelle 6.1: Hardware- und Software-Spezifikationen der Testumgebung im Vergleich

Um die Performanz des Scanners isoliert von externen Störfaktoren zu evaluieren, wird ein dedizierter Laboraufbau gewählt. Dafür werden zwei Geräte per Ethernet-Kabel direkt miteinander verbunden:

1. **Der Scanner-Knoten:** Das System, auf dem der Prototyp ausgeführt wird und welches die Systemmetriken erfasst.

2. **Der Ziel-Knoten:** Ein System, welches ein Zielnetzwerk simuliert.

Für die Schaffung einer einheitlichen und reproduzierbare Messgrundlage, welche menschliche Fehler möglichst vermeidet, wurden für die Szenarien (siehe 4) Python-Programme erstellt, mit welchen der Ablauf, die Datenerfassung und Datenaufbereitung weitesgehend automatisiert wird. Dieser Ansatz gewährleistet konsistente Rahmenbedingungen in Form von einheitlichen Pausenzeiten und der einheitlichen Erfassung der Systemressourcen.

Für das Szenario 1 (siehe 4) werden die Scanausgaben, sowie die in 6.2 beschriebenen Tools genutzt, um zusätzlich zu den Ausgaben des *Dummy-Receiver*-Programmes exakte Validierung anstellen zu können.

Tool	**Nutzung**
Netcat	TODO
tcpdump	TODO
xdpdump	TODO
Wireshark	TODO

Tabelle 6.2: Genutzte Tools zur Validierung der funktionalen Anforderungen (siehe 4)

In den Szenarien 2 und 3 erfolgt die Messung der Metriken (siehe 4) mit einer Abtastrate von 10 Hz (Intervall $t = 0,1\,\text{s}$). Da die Scanner Subprozesse und mehrere *Threads* starten, wird die gesamte Systemlast gemessen. Das *Benchmarking*-Programm verzichtet auf *High-Level*-Tools, um den *Overhead* der Messung selbst zu minimieren und liest die erforderlichen Kernel-Statistiken direkt aus `procfs`. Dies ist ein virtuelles Dateisystem, welches der Anzeige und Änderung von Systemparametern dient [57] und eine direkte Schnittstelle zu den internen Datenstrukturen des Linux-Kernels bietet.

Um mögliche Fluktuationen in der Grundlast zu vermeiden, wird jeglicher Zugang zum Internet geschlossen und kein weiterer Prozess abseits des Benchmarking-Programmes manuell gestartet. Jedes Szenario wird in fünf unabhängigen Iterationen durchgeführt, sodass statistische Ausreißer weniger ins Gewicht fallen.

6.1.2 Aufbau des Ziel-Knotens

Der Zielknoten besteht ähnlich wie die Empfangslogik des Scanners aus einem mithilfe des `aya` *Crates* erstellten `eBPF`-Programmes, welches die eingehenden Pakete mittels Zeiger-Operationen parst, anschließend validiert und bei Erfolg ein Antwortpaket via `XDP_TX` versendet. Allerdings wird hier geprüft, ob es sich um ein `IPv4-SYN`-Paket handelt und anschließend eine darauf zugeschnittene `SYN-ACK`-Antwort statt eines `RST`-Paketes versendet. Die Modifikation passiert auch hier am selben Paket ohne dieses zuvor zu kopieren.

Zur Steuerung der Antwortwahrscheinlichkeit ist ein beim Start des Programmes veränderbarer Parameter integriert, welcher über die Kommandozeile übergeben wird. Wenn ein valides Paket erhalten wurde, wird eine Zufallszahl generiert und in Verbindung mit der eingegebenen Prozentzahl genutzt, um zu entscheiden, ob eine Antwort gesendet wird oder nicht.

Auch die Statistiken werden mit der gleichen Datenstruktur wie beim Scanner - dem `PerCpuArray` - ermittelt, um eine lockfreie, effiziente Erhebung zu gewährleisten. Es werden folgende Statistiken ermittelt, um die Funktionsweise des Scanners und des Laboraufbaus zu validieren:

1. **Empfangene Pakete gesamt**
2. **Valide `SYN`-Pakete**
3. **Gesendete Antworten**

6.2 Versuchsablauf

Die `benchmark_suite.py ??` startet die Scanner-Prozesse nacheinander und misst mit einem parallelen *Monitoring-Thread* die Systemressourcen in folgenden Phasen:

1. **Ruhezustands-Messung:** Vor den eigentlichen Tests wird über einen Zeitraum von fünf Sekunden der Systemzustand ohne Last gemessen. Dieser Durchschnittswert dient als Referenzpunkt, um das Grundrauschen des Betriebssystems später herausrechnen zu können.
2. **Aufzeichnung:** Die Aufzeichnung der Metriken beginnt eine Sekunde vor dem Prozessstart, um das Anlaufverhalten und Initialisierungsspitzen der Scanner vollständig zu erfassen.
3. **Aktive Phase:** Während der Scanner läuft, überwacht ein *Watchdog*-Algorithmus[1] den ausgehenden Datenverkehr. Der Scanner gilt als aktiv, sobald die Senderate 100 *PPS* überschreitet und als inaktiv, sobald die Grenze wieder unterschritten wird.
4. **Externes Beenden:** Sollte die Rate nach dem Start für mehr als sieben Sekunden unter einen Schwellenwert von 100 *PPS* fallen, wird der Prozess terminiert. Dies ist notwendig, da Masscan sich häufig nicht von alleine beendet.

In Szenario 1 **TODO** wird zur Aswertung statt der Metriken der `benchmark_suite.py` ein eigenes Programm (`validate_scanner.py` **TODO**) genutzt. Für die Messungen zur Validierung der Pakete wird gar kein automatisierter Ablauf genutzt, da aufgrund der Nutzung von Wireshark 6.2 eine manuelle Auswertung erfolgt.

[1] TODO

6.2.1 Datenaufbereitung und Bereinigung

Um die Scanergebnisse zu plausibilisieren wird zusätzlich via Python-Programm **??** die Anzahl der ausgegebenen Ergebnisse gezählt. Zur Validierung werden im ersten Szenario die Werte zusätzlich mit den empfangenen Paketen des Ziel-Knotens und den theoretischen Werten verglichen. Da keine automatische Synchronisierung der Knoten stattfindet, wird dieses Szenario also manuell durchgeführt.

Die Rohdaten werden nach der Messung mithilfe des Programmes **??** statistisch bereinigt und visualisiert. Eine einfache Mittelwertbildung über die gesamte Laufzeit alleine ist nicht zielführend, da Start- und Stopp-Phasen die Ergebnisse verzerren würden. Stattdessen werden die Ergebnisse mit folgenden Vorgehensweisen aufbereitet:

- **Isolation der Hochlastphase:** Für die Berechnung des durchschnittlichen Durchsatzes und der Effizienz (*PPS*/CPU Auslastung) werden nur jene Zeitfenster berücksichtigt, in denen der Scanner aktiv sendet.
- **Netto-Ressourcenberechnung:** Von den gemessenen CPU- und RAM-Werten wird der in Phase (siehe 4) ermittelte *Baseline*-Wert subtrahiert. Dies stellt sicher, dass die dargestellten Ergebnisse ausschließlich den Ressourcenbedarf des Scanners abbilden und unabhängig von Hintergrundprozessen des Betriebssystems sind.

Aus den bereinigten Daten werden anschließend Diagramme und Tabellen via Python `matplotlib` generiert, welche die Daten in anschaulicher Weise darstellen.

6.3 Inkompatibilitäten und Limitierungen

6.3.1 *Zero-Copy*-Modus

Da finanziell bedingt nur die Möglichkeit besteht, die in 6.1 angegebene Hardware zu nutzen, muss für die Evaluation eine Einschränkung gemacht werden. Der äußerst effiziente *Zero-Copy*-Modus ist bei der Nutzung von Netzwerkkarten mit dem `igb`-Treiber eingeschränkt. Aufgrund der niedrigen Anzahl an Sende- und Empfangsringen auf der Netzwerkkarte können dem `XDP`-Programm keine dedizierten *Queues* [2] zugeteilt werden. Dies hat zur Folge, dass sich die Sende-Ringe mit dem Betriebssystem geteilt werden müssen und es somit zu sogenannter *Lock Contention* [3] kommen kann [58]. Dies ist sehr Ressourcenaufwändig und führte in den Tests dazu, dass der `AF_XDP`-*Socket* und das `eBPF`-Programm um den Zugriff auf einen Sendering konkurrierten, sobald ein `SYN-ACK`-Paket über die gleiche *Queue* einging, über die auch versendet wurde. Dies ist unvermeidbar, da ein `RST`-Paket per `XDP_TX` zwangsweise über die selbe *Queue* verschickt wird, über die es eingeht **TODO** und die

[2] Warteschlangen

[3] Der Zugriff auf den Ring muss bei jeder Operation ausgehandelt werden, sollten mehrere Parteien ihn gleichzeitig nutzen wollen

Pakete durch den ...TODO... gleichmäßig auf alle *Queues* verteilt werden **TODO**. Aufgrund der sehr hohen Senderate und der vergleichbar kleinen Puffer der Netzwerkkartenringe [4] laufen dieser durch die *lock contention* voll und blockieren das Versenden der `RST`-Antworten in dieser *Queue*.

Diese Vermutungen wurden dadurch gestützt, dass bei den Tests unabhängig der gesendeten Menge immer 25% der `RST`-Pakete beim Ziel-Knoten fehlten. Da das Scanner-System 4 *Queues* besitzt und eine *Queue* für das Senden genutzt wurde entstehen somit ein Viertel-, beziehungsweise 25% der Pakete, welche aufgrund der *Lock Contention* verloren gingen. Tests mit Beschränkung der *Queue*-Anzahl mithilfe des folgenden Befehls: `sudo ethtool -L enp6s0 combined x` auf die Anzahl x bestätigten dies auch. Bei einer *Queue* wurden rund 100% und bei 3 *Queues* rund 33% der `RST`-Pakete nicht gesendet.

Aufgrund dessen wird der *Zero-Copy*-Modus ausschließlich ohne das Senden von `RST`-Paketen getestet.

6.3.2 Paketrate

Die Durchsatzrate ist durch die genutzten Netzwerkkarten sowie dem LAN-Kabel auf das Limit einer Gigabit-Verbindung beschränkt. Die Tests zur maximalen Durchsatzrate können deshalb nur eingeschränkt durchgeführt werden. Das theoretische Limit einer Gigabit-Leitung liegt nach IEEE 802.3 [59], bei einer Paketgröße von 64 Byte plus 20 Byte *Overhead*, bei 1,488 Millionen *PPS*. Deshalb wird der Fokus dort verstärkt auf die gemessene Ressourcenauslastung gelegt.

6.4 Umsetzung der Testszenarien

Im Folgenden wird die technische Umsetzung der in Abschnitt 4 definierten Szenarien beschrieben. Die Szenarien 2 und 3 (siehe 4) werden mit allen drei zu evaluierenden Scanner-Varianten (Rust-XDP-Copy, Rust-XDP-ZeroCopy, Rust-`AF_PACKET`) sowie den Vergleichstools (ZMap, Masscan) durchgeführt. Szenario 1 (siehe 4) wird nur mit den Rust-Scannern durchgeführt.

In allen Szenarien wurden 64 verschiedene Source-IP-Adressen genutzt, da dies ein essenzieller Faktor zur Erhöhung der Antwortwahrscheinlichkeit in realen Hochgeschwindigkeits-Scan-Szenarien darstellt [2]. Außerdem antwortet der Ziel-Knoten auf 20% der Pakete mit validen SYN-ACK Antworten. Dies ist im Vergleich zur realistischen Antwortrate wenn man den kompletten IPv4-Raum scannt ein sehr hoch angesetzter Wert [2], soll aber die Funktionsfähigkeit für Spezialfälle sicherstellen.

[4] Für die Intel I210 und I350 sind es maximal 4096 Deskriptoren (siehe A.1)

Szenario 1 - Anforderungsvalidierung

Der erste Test dient als *Proof of Concept* des Scanners. Hierbei wird ein kleiner IP-Adressraum (`/20`) gescannt. Der Ziel-Knoten antwortet auf alle IP-Adressen welche mit einer geraden Zahl enden, um die anschließende Auswertung zu vereinfachen. Dabei wird das Senden von `RST`-Paketen aktiviert.

Im zweiten manuellen Test, werden nur jeweils 4 Pakete verschickt. Dies genügt, um zu erkennen, ob die Pakete korrekt sind und fördert die Übersichtlichkeit. Die Ergebnisse werden mithilfe der in 6.2 beschriebenen Netzwerktools ausgewertet.

Szenario 2 - Performanzgrenze

Um die Performance und Effizienz der Scanner zu validieren, werden alle bremsenden Faktoren, die nicht essenziell für das reine Versenden und Empfangen von Paketen sind, deaktiviert.

- **Senderate:** Die Senderate wird so gewählt, dass sie das theoretische Limit der Gigabit-Leitung übersteigt.
- **Features:** Es wird auf rechenintensive Funktionen wie die Deduplizierung von Antworten, sowie das Senden von `RST`-Antworten verzichtet.
- **IP-Raum:** Als Ziel dient ein `/6`-Netzwerk[5], um eine hinreichend lange Laufzeit für die Erfassung stabiler Messwerte zu gewährleisten.

Szenario 3 - Reales Szenario

Das dritte Szenario simuliert einen praxisnahen Scan-Vorgang. Hierbei werden Parameter gewählt, die helfen, Sicherheitsmechanismen zu umgehen oder die Zuordnung von Antworten zu erleichtern.

- **Senderate:** Die Senderate wird auf 500.000 *PPS* fixiert, da übermäßig hohe Raten die Wahrscheinlichkeit erhöhen, dass Scan-Muster von *Firewalls* oder *IPS* erkannt und blockiert werden.
- **Ports:** Der Scan erfolgt parallel auf den Ports 80 und 443, um das Scan-Verhalten zu diversifizieren und weiter zu verschleiern.
- **Quellports:** Es wird ein Bereich von 128 *Source-Ports* $(60000 - 60127)$ verwendet. Dies dient der besseren Lastverteilung auf der Empfängerseite und Verschleierung des Scans.

[5] 67.108.864 Millionen IP-Adressen

- **IP-Raum:** Der Zielbereich wird auf ein `/10`-Netzwerk[6] beschränkt, um die Gesamtdauer des Tests in einem praktikablen Rahmen zu halten.

[6] 4.194.304 Millionen IP-Adressen

Kapitel 7: Evaluation und Ausblick

In diesem Kapitel werden die in der Testumgebung ermittelten Messergebnisse vorgestellt, analysiert und diskutiert. Ziel ist es, die Leistungsfähigkeit des implementierten Rust-Scanners im Vergleich zu etablierten Tools zu bewerten und die Erfüllung der definierten Anforderungen zu überprüfen. Abschließend wird ein Ausblick auf mögliche Weiterentwicklungen gegeben.

7.1 Darstellung und Reproduzierbarkeit der Messergebnisse

Die Messergebnisse wurden mittels der in 6.2.1 beschriebenen Skripte aufbereitet und stehen im Anhang zur Verfügung. Im Folgenden wird nur auf die ausschlaggebenden Ergebnisse eingegangen. Für jede Messung liegen allerdings umfangreiche Daten, sowie Diagramme und Tabellen im bereitgestellten GitHub Repository **??** zur Verfügung. Der Ablauf, inklusive Erhebung und konkrete Parameter, sowie die Ausgaben der Tests lässt sich dort unter logs_benchmark_suite.txt anhand der Ein- und Ausgaben im Terminal nachvollziehen. Mittels dessen und der README.md Datei können die Benchmarks exakt reproduziert und nachvollzogen werden.

7.1.1 Ergebnisse S-01: Anforderungsvalidierung

7.1.2 Ergebnisse S-02: Performanzgrenzen

Gemäß 6.2.1 wird in *Aktiv* (Zeitraum während Paketfluss besteht) und *Gesamt* (Gesamte Laufzeit des Programmes) unterschieden. *Netto* beschreibt dabei, dass die Werte von der Grundlast bereinigt wurden. Aus den Ergebnissen des Tests zum Szenario /S-02/ erschließen sich nach [TODO Ergebnisse] und [TODO Validierung] die in 7.1.2 dargestellten Werte. Aufgrund von Einschränkungen durch die eigene *Blacklist* hat ZMap weniger IP-Adressen gescannt, weshalb es weniger Ergebnisse hervorbrachte. Außerdem ist zu beachten, dass sowohl Masscan als auch ZMap keine Option zur Vermeidung des Sendens von RST-Antworten besitzen. Da Masscan die Pakete in dessen eigens gefertigtem Userspace-TCP-Stack erstellt und versendet, fließen diese auch in die PPS Metrik ein. Die durch ZMap

Scanner	**PPS**	**Netto (Aktiv)**		**Netto (Gesamt)**		**Ergebnisse**
	(aktiv) [Mio]	CPU [%]	RAM [MB]	CPU [%]	RAM [MB]	[Mio]
SYN-Rust (XDP, Zero-Copy)	1,48	5,7	190,8	4,7	160,7	13,42
SYN-Rust (XDP, Copy)	1,23	9,2	237,7	7,7	203,7	13,42
SYN-Rust (XDP, Generic)	1,23	10,4	263,8	8,7	231,8	13,42
Masscan	1,05	13,5	42,6	12,3	42,1	13,42
SYN-Rust (AF_PACKET)	1,06	14,2	265,6	12,1	240,9	13,42
ZMap	1,35	21,1	13,0	17,8	12,4	10,07

Tabelle 7.1: Vergleich der Performance-Metriken

bedingten RST-Antworten werden automatisch vom Kernel gesendet und nicht in den genutzten Kernel-Logs erfasst.

Zur Berechnung der in 7.1 gezeigten Werte zur Effizienz der Scanner während der aktiven Phase, wurde der Durchsatz pro CPU-Prozent in jedem Durchlauf berechnet und daraus anschließend der Durchschnittswert gebildet.

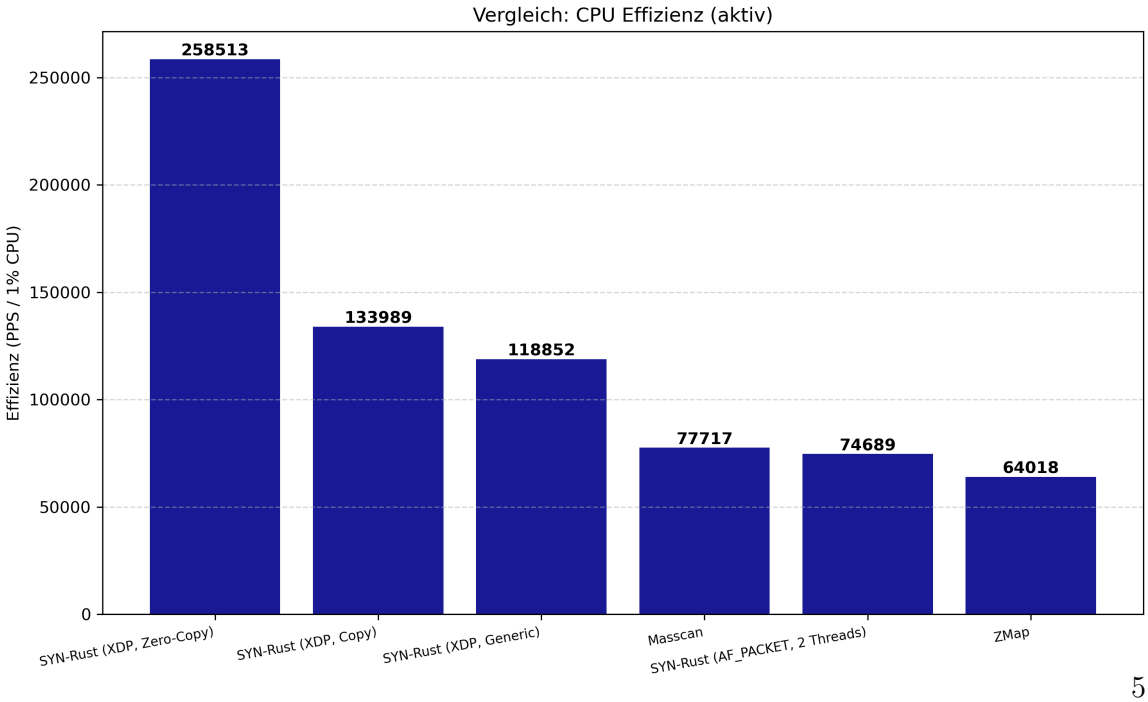

Abbildung 7.1: Effizienz der SYN-Scanner im Benchmark (aktiv)

7.1.3 Ergebnisse S-03: Reales Szenario

Die Ergebnisse mancher Variationen des SYN-Rust in der Tabelle 7.2 weichen bezüglich der PPS Metrik von der Durchsatzlimitierung (500.000 PPS) ab. Dies ist in diesem Szenario /S-03/ explizit erwünscht. Die Daten der Tabelleneinträge sind in [TODO Ergebnisse] und [TODO Validierung] zu finden. Für ZMap gilt bezüglich der RST-Antworten das gleiche wie auch in 7.1.2.

Scanner	**Effizienz**	**PPS**	**Netto (Aktiv)**		**Netto (Gesamt)**		**Erg.**
	[PPS/%]	(aktiv) [Mio]	CPU [%]	RAM [MB]	CPU [%]	RAM [MB]	[Mio]
SYN-Rust (XDP, Zero-Copy, kein RST)	184353	0,51	2,8	82,2	1,8	62,0	1,64
SYN-Rust (XDP, Copy)	106118	0,60	5,7	77,0	3,6	57,7	1,64
SYN-Rust (XDP, Generic)	103087	0,60	5,9	96,9	3,7	74,7	1,64
SYN-Rust (AF_PACKET)	99871	0,61	6,1	112,2	3,8	88,3	1,64
Masscan (ohne Deduplizierung, RST automatisch)	75534	0,60	6,6	34,8	4,8	31,4	1,68
ZMap (RST automatisch)	13272	0,59	37,2	62,3	26,5	58,3	1,68

Tabelle 7.2: Vergleich der Performance-Metriken (Unter Berücksichtigung der RST-Pakete)

7.2 Diskussion der Ergebnisse

7.2.1 Analyse des Durchsatzes und der Latenz

7.2.2 Ressourceneffizienz (CPU und RAM)

7.2.3 Vergleich mit dem Stand der Technik

7.3 Abgleich mit den Anforderungen

7.4 Ausblick

7.5 Fazit

Anhang A: Ergänzende Systeminformationen

A.1 Netzwerkkarten-Konfiguration (Ethtool)

Der folgende Auszug zeigt die Standard-Konfiguration der Netzwerkschnittstelle `enp6s0` vor der Optimierung des Ring-Buffers.

```
Ring parameters for enp6s0:
Pre-set maximums:
RX:             4096
RX Mini:        n/a
RX Jumbo:       n/a
TX:             4096
TX push buff len:       n/a
Current hardware settings:
RX:             256
RX Mini:        n/a
RX Jumbo:       n/a
TX:             256
RX Buf Len:     n/a
CQE Size:       n/a
TX Push:        off
RX Push:        off
TX push buff len:       n/a
TCP data split:         n/a
```

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

[1] H. Griffioen, G. Koursiounis, G. Smaragdakis und C. Doerr, „Have you syn me? characterizing ten years of internet scanning," in *Proceedings of the 2024 ACM on Internet Measurement Conference*, 2024, S. 149–164.

[2] Z. Durumeric, D. Adrian, P. Stephens, E. Wustrow und J. A. Halderman, „Ten Years of ZMap," en, in *Proceedings of the 2024 ACM on Internet Measurement Conference*, Madrid Spain: ACM, Nov. 2024, S. 139–148, ISBN: 979-8-4007-0592-2. DOI: 10.1145/3646547.3689012. Adresse: https://dl.acm.org/doi/10.1145/3646547.3689012.

[3] R. D. Graham, *robertdavidgraham/masscan*, C, Jan. 2026. Adresse: https://github.com/robertdavidgraham/masscan.

[4] Z. Durumeric, E. Wustrow und J. A. Halderman, „ZMap: Fast Internet-wide Scanning and Its Security Applications," en,

[5] S. Rudnev, A. Zolkin, N. Artemyev und A. Tychkov, „THE ECONOMIC IMPORTANCE OF CYBERSECURITY FOR ENTERPRISES IN THE CONTEXT OF DIGITAL TRANSFORMATION," *EKONOMIKA I UPRAVLENIE: PROBLEMY, RESHENIYA*, Jg. 11/2, S. 46–55, Jan. 2024. DOI: 10.36871/ek.up.p.r.2024.11.02.006.

[6] O. I. Falowo, I. Okpala, E. Kojo, S. Azumah und C. Li, „Exploration of Various Machine Learning Techniques for Identifying and Mitigating DDoS Attacks," in *2023 20th Annual International Conference on Privacy, Security and Trust (PST)*, Aug. 2023, S. 1–7. DOI: 10.1109/PST58708.2023.10320151. Adresse: https://ieeexplore.ieee.org/document/10320151/.

[7] X. Li, *idealeer/xmap*, C, Jan. 2026. Adresse: https://github.com/idealeer/xmap.

[8] G. Li, M. Zhang, C. Guo u. a., „IMap: Fast and Scalable In-Network Scanning with Programmable Switches," en, 2022, S. 667–681, ISBN: 978-1-939133-27-4. Adresse: https://www.usenix.org/conference/nsdi22/presentation/li-guanyu.

[9] S. Peta, „C Programming Language - Still Ruling the World," en, *International Journal of Science and Research (IJSR)*, Jg. 11, Nr. 4, S. 548–552, Apr. 2022, ISSN: 23197064. DOI: 10.21275/SR22403142926.

[10] A. Al-Boghdady, K. Wassif, M. El-Ramly, A. Al-Boghdady, K. Wassif und M. El-Ramly, „The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT's Low-End Devices," en, *Sensors*, Jg. 21, Nr. 7, März 2021, Company: Multidisciplinary Digital Publishing Institute Distributor: Multidisciplinary Digital Publishing Institute Institution: Multidisciplinary Digital Publishing Institute Label:

Multidisciplinary Digital Publishing Institute publisher: publisher, ISSN: 1424-8220. DOI: 10.3390/s21072329. Adresse: https://www.mdpi.com/1424-8220/21/7/2329.

[11] W. Bugden und A. Alahmar, „The safety and performance of prominent programming languages," *International Journal of Software Engineering and Knowledge Engineering*, Jg. 32, Nr. 05, S. 713–744, 2022.

[12] P. C. van Oorschot, „Memory Errors and Memory Safety: C as a Case Study," *IEEE Security and Privacy*, Jg. 21, Nr. 2, S. 70–76, März 2023, ISSN: 1558-4046. DOI: 10.1109/MSEC.2023.3236542.

[13] M. Costanzo, E. Rucci, M. Naiouf und A. D. Giusti, „Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body," Nr. arXiv:2107.11912, Okt. 2021, arXiv:2107.11912 [cs]. DOI: 10.48550/arXiv.2107.11912. Adresse: http://arxiv.org/abs/2107.11912.

[14] U. T. H. O. Malaysia und F. H. Roslan, „A Comparative Performance of Port Scanning Techniques," en, *Journal of Soft Computing and Data Mining*, Jg. 4, Nr. 2, Okt. 2023, ISSN: 2716621X. DOI: 10.30880/jscdm.2023.04.02.004. Adresse: https://publisher.uthm.edu.my/ojs/index.php/jscdm/article/view/13623/5962.

[15] G. Lyon, *Nmap network scanning: official Nmap project guide to network discovery and security scanning*, eng, Zero-day release: May 2008. Sunnyvale, CA: Insecure.Com LLC, 2010, ISBN: 978-0-9799587-1-7.

[16] Adresse: https://nmap.org/book/port-scanning.html#port-scanning-port-intro.

[17] en. Adresse: https://www.hanser-elibrary.com/doi/epdf/10.3139/9783446484856.

[18] IANA, *Service Name and Transport Protocol Port Number Registry*. Adresse: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

[19] M. Kerrisk, *The Linux programming interface: a Linux und UNIX system programming handbook*, eng, Ninth printing. San Francisco, CA: No Starch Press, 2018, ISBN: 978-1-59327-220-3.

[20] S. Wendzel, *IT-Sicherheit für TCP/IP- und IoT-Netzwerke: Grundlagen, Konzepte, Protokolle, Härtung* (Springer eBook Collection), ger, 2., aktualisierte und erweiterte Auflage. Wiesbaden: Springer Vieweg, 2021, ISBN: 978-3-658-33422-2. DOI: 10.1007/978-3-658-33423-9.

[21] J. Postel, *Transmission Control Protocol*, en. 1981, RFC0793. DOI: 10.17487/rfc0793. Adresse: https://www.rfc-editor.org/info/rfc0793.

[22] W. Eddy, *Transmission Control Protocol (TCP)*. Aug. 2022. DOI: 10.17487/RFC9293. Adresse: https://datatracker.ietf.org/doc/rfc9293.

[23] K. A. Scarfone, M. P. Souppaya, A. Cody und A. D. Orebaugh, *Technical guide to information security testing and assessment*. en, 0. Aufl. Gaithersburg, MD, 2008, NIST SP 800–115. DOI: 10.6028/NIST.SP.800-115. Adresse: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-115.pdf.

[24] W. Eddy, *TCP SYN Flooding Attacks and Common Mitigations*. Aug. 2007. DOI: 10.17487/RFC4987. Adresse: https://datatracker.ietf.org/doc/rfc4987.

[25] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann u. a., „The eXpress data path: fast programmable packet processing in the operating system kernel," en, in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, Heraklion Greece: ACM, Dez. 2018, S. 54–66, ISBN: 978-1-4503-6080-7. DOI: 10.1145/3281411.3281443. Adresse: https://dl.acm.org/doi/10.1145/3281411.3281443.

[26] *socket(2) - Linux manual page*, man7.org, Accessed: 2026-01-11. Adresse: https://man7.org/linux/man-pages/man2/socket.2.html.

[27] *raw(7) - Linux manual page*, man7.org, Accessed: 2026-01-11. Adresse: https://man7.org/linux/man-pages/man7/raw.7.html.

[28] *address_families(7) - Linux manual page*, man7.org, Accessed: 2026-01-11. Adresse: https://man7.org/linux/man-pages/man7/address_families.7.html.

[29] Adresse: https://man7.org/linux/man-pages/man7/packet.7.html.

[30] S. McCanne und V. Jacobson, „The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA: USENIX Association, Jan. 1993. Adresse: https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet.

[31] N. R. Pinnapareddy, „eBPF for high-performance networking and security in cloud-native environments," *International Journal of Science and Research Archive*, Jg. 15, Nr. 2, S. 207–225, Mai 2025, ISSN: 25828185. DOI: 10.30574/ijsra.2025.15.2.1264.

[32] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior und L. F. M. Vieira, „Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," en, *ACM Computing Surveys*, Jg. 53, Nr. 1, S. 1–36, Jan. 2021, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3371038.

[33] X. Zhang, X. Shu, L. Chen und R. Xie, „High-Performance Network Firewall Based on XDP," in *2024 20th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, Guangzhou, China: IEEE, 2024, S. 1–6, ISBN: 979-8-3503-5632-8. DOI: 10.1109/ICNC-FSKD64080.2024.10702282. Adresse: https://ieeexplore.ieee.org/document/10702282/.

[34] W. Bugden und A. Alahmar, „Rust: The Programming Language for Safety and Performance," Nr. arXiv:2206.05503, 2022, arXiv:2206.05503 [cs]. DOI: 10.48550/arXiv.2206.05503. Adresse: http://arxiv.org/abs/2206.05503.

[35] R. Jung, J.-H. Jourdan, R. Krebbers und D. Dreyer, „Safe systems programming in Rust," en, *Communications of the ACM*, Jg. 64, Nr. 4, S. 144–152, Apr. 2021, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3418295.

[36] en. DOI: 10.1145/3158154. Adresse: https://dl.acm.org/doi/epdf/10.1145/3158154.

[37] C. Cui und H. Xu, „Unleashing the Efficiency of Rust: An Empirical Study of Performance Bugs in Rust Projects,“ in *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*, São Paulo, Brazil: IEEE, Okt. 2025, S. 371–381, ISBN: 979-8-3503-9302-6. DOI: 10.1109/ISSRE66568.2025.00045. Adresse: https://ieeexplore.ieee.org/document/11229568/.

[38] A. Silberschatz, P. B. Galvin und G. Gagne, *Operating system concepts*, eng, 10th edition. Hoboken, NJ: Wiley, 2018, ISBN: 978-1-119-32091-3.

[39] R. H. Arpaci-Dusseau und A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00. Arpaci-Dusseau Books, Aug. 2018.

[40] en. Adresse: https://go.dev/blog/waza-talk.

[41] B. Stroustrup, *The design and evolution of C++*, eng. Reading (Mass.): Addison-Wesley, 1994, ISBN: 978-0-201-54330-8.

[42] D. Adrian, Z. Durumeric, G. Singh und J. A. Halderman, „Zippier ZMap: Internet-Wide Scanning at 10 Gbps,“ en,

[43] R. Abu Bakar und B. Kijsirikul, „Enhancing Network Visibility and Security with Advanced Port Scanning Techniques,“ en, *Sensors*, Jg. 23, Nr. 17, S. 7541, Aug. 2023, ISSN: 1424-8220. DOI: 10.3390/s23177541.

[44] J. M. Pittman, „A Comparative Analysis of Port Scanning Tool Efficacy,“ Nr. arXiv:2303.11282, März 2023, arXiv:2303.11282 [cs]. DOI: 10.48550/arXiv.2303.11282. Adresse: http://arxiv.org/abs/2303.11282.

[45] L. Rizzo, „netmap: a novel framework for fast packet I/O,“ en,

[46] R. Taupaani und R. Harwahyu, „ZTSCAN: ENHANCING ZERO TRUST RESOURCE DISCOVERY WITH MASSCAN AND NMAP INTEGRATION,“ en, *JITK (Jurnal Ilmu Pengetahuan dan Teknologi Komputer)*, Jg. 10, Nr. 4, S. 868–877, Mai 2025, ISSN: 2527-4864. DOI: 10.33480/jitk.v10i4.6628.

[47] R. Sagramoni, G. Lettieri und G. Procissi, „On the Impact of Memory Safety on Fast Network I/O,“ in *2024 IEEE 25th International Conference on High Performance Switching and Routing (HPSR)*, 2024, S. 161–166. DOI: 10.1109/HPSR62440.2024.10635971. Adresse: https://ieeexplore.ieee.org/document/10635971/.

[48] A. Gonzalez, D. Mvondo und Y.-D. Bromberg, „Takeaways of Implementing a Native Rust UDP Tunneling Network Driver in the Linux Kernel,“ *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*, 2023. DOI: 10.1145/3623759.3624547.

[49] S. Moon, „Toward building memory-safe network functions with modest performance overhead,“ 2017.

[50] P. Emmerich, S. Ellmann, F. Bonk u. a., „The Case for Writing Network Drivers in High-Level Programming Languages,“ en, in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Cambridge, UK: IEEE, 2019, S. 1–13, ISBN: 978-1-7281-4387-3. DOI: 10.1109/ANCS.2019.8901892. Adresse: https://ieeexplore.ieee.org/document/8901892/.

[51] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić und L. Ryzhyk, „System Programming in Rust: Beyond Safety,“ *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017. DOI: 10.1145/3102980.3103006.

[52] *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*. Adresse: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-2:v1:en.

[53] Adresse: https://docs.rs/tokio/latest/tokio/task/.

[54] Adresse: https://docs.kernel.org/security/siphash.html.

[55] en. Adresse: https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_PERCPU_ARRAY/.

[56] en. Adresse: https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_RINGBUF/.

[57] Adresse: https://www.kernel.org/doc/Documentation/filesystems/proc.txt.

[58] Adresse: https://github.com/torvalds/linux/commit/9cbc948b5a20c9c054d9631099c0426c16da

[59] DOI: 10.1109/IEEESTD.2022.9844436. Adresse: https://ieeexplore.ieee.org/document/9844436/.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 06.02.2025

Lennard Alexander Dubhorn