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

Netzwerk-Scanning macht einen großen Teil des Internet-Verkehrs im IPv4 Adressraumes aus. So ist 98 Prozent des gesamten TCP Verkehrs weltweit auf `SYN`-Scans zurückzuführen [1]. Bekannte Tools wie *ZMap* werden stetig weiterentwickelt [2] und sind seit der Entwicklung von performanten Open-Source Scannern wie *ZMap* oder *Masscan* [3][4] dazu fähig, den gesamten IPv4 Adressraum in weniger als 45min 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 [3][4][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 [TODO]. Allerdings ist C anfällig für menschengemachte Fehler [9] wie ..TODO.. [TODO], von welchen die meisten sicherheitsrelevanten aus der Fraktion der Speicherverwaltung stammen [10]. Andere Sprachen wie Go oder Python lösen einige dieser Probleme durch die Nutzung einer automatischen Speicherverwaltung [TODO]. Diese Sprachen sind allerdings im Vergleich zu Sprachen wie C weniger performant [10].

Rust hingegen schneidet in Vergleichen bezüglich der Performance auf ähnlichem Niveau wie C ab, bringt gleichzeitig aber das höchste Sicherheitsniveau der genannten Sprachen mit [10][11]. Außerdem unterstützt Rust Konzepte von Sprachen hoher Abstraktionsebene, wie beispielsweise die der funktionalen Programmierung oder Objektorientierung [11], 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

[1] Auf hoher Abstraktionsebene

ist ungeklärt, ob der potenzielle Performanceunterschied 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, weshalb die Architektur teilweise asynchron gestaltet und leistungsfähige Linux-Schnittstellen wie `AF_PACKET` und `XDP` 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 spracheigene Sicherheitsgarantien eine tragfähige Alternative für den produktiven Einsatz darstellen?

1.3 Abgrenzung des Themas

Bei der in dieser Arbeit entwickelten Implementierung handelt es sich um einen horizontalen Scanner 2. Anders als beispielsweise beim regulären `SYN`-Scan des Tools *Nmap* [12], welcher in der Regel vertikal erfolgt.

Zusätzliche Mechanismen zur Verschleierung des Scans oder weiterführende Maßnahmen zur Treffererhöhung werden in dieser Implementierung rudimentär oder gar nicht behandelt, da der Fokus auf der Nutzung von Rust, sowie der Entwicklung eines Performanz-orientieren Netzwerkscanners liegt. Da der normale Ablauf des SYN-Scans bereits grundlegende Mechanismen in den Bereichen mitbringt [13], sind diese Gebiete für die Beantwortung der Forschungsfrage nicht notwendig. Die weiterführende Untersuchung der Ergebnisse bildet einen eigenen Forschungszweig und ist somit auch ausgeschlossen.

Außerdem beschränkt sich diese Arbeit auf den IPv4 Adressraum, da dies genügt, um der Forschungsfrage nachzugehen.

Kapitel 2: Theoretische Grundlagen

In diesem Kapitel werden die nötigen Grundlagen zum Verständnis des Portscanning 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 wird genauer vorgestellt.

2.1 Grundlagen der Netzwerkkommunikation

Bei der Kommunikation in TCP/IP [1] Netzwerken dient das IP-Protokoll und die IP-Adressen zur Identifikation der Maschine im Netzwerk, während die genaue Adressierung der spezifischen Anwendungen durch sogenannte Ports bzw. der sogenannten Portnummer bestimmt wird [12] (TODO SEITE?). Die Portnummer ist ein 16-Bit-Wert und kann somit zwischen jeweils einschließlich 0 und 65535 liegen [Computernetzwerke S107]. Einige Portnummern sind fest vergeben oder für bestimmte Anwendungen registriert [14], was es ermöglicht, gezielt nach bestimmten Anwendungen zu scannen. Der gesamte Kommunikations-Endpunkt wird *Socket* genannt. TODO cite [Computernetzwerke S108].

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 [12].
- **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 [12].

[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 Transportprotokoll des Internets [15] TODO S71. Es gewährleistet eine zuverlässige, verbindungsorientierte Datenübertragung zwischen den Prozessen der Hosts. Die ursprüngliche Spezifikation erfolgte im RFC 793 [16], welches durch RFC 9293 [17] konsolidiert wurde. Für die Entwicklung eines SYN-Portscanners sind insbesondere der Aufbau des TCP-Headers und der Mechanismus des Verbindungsaufbaues entscheidend.

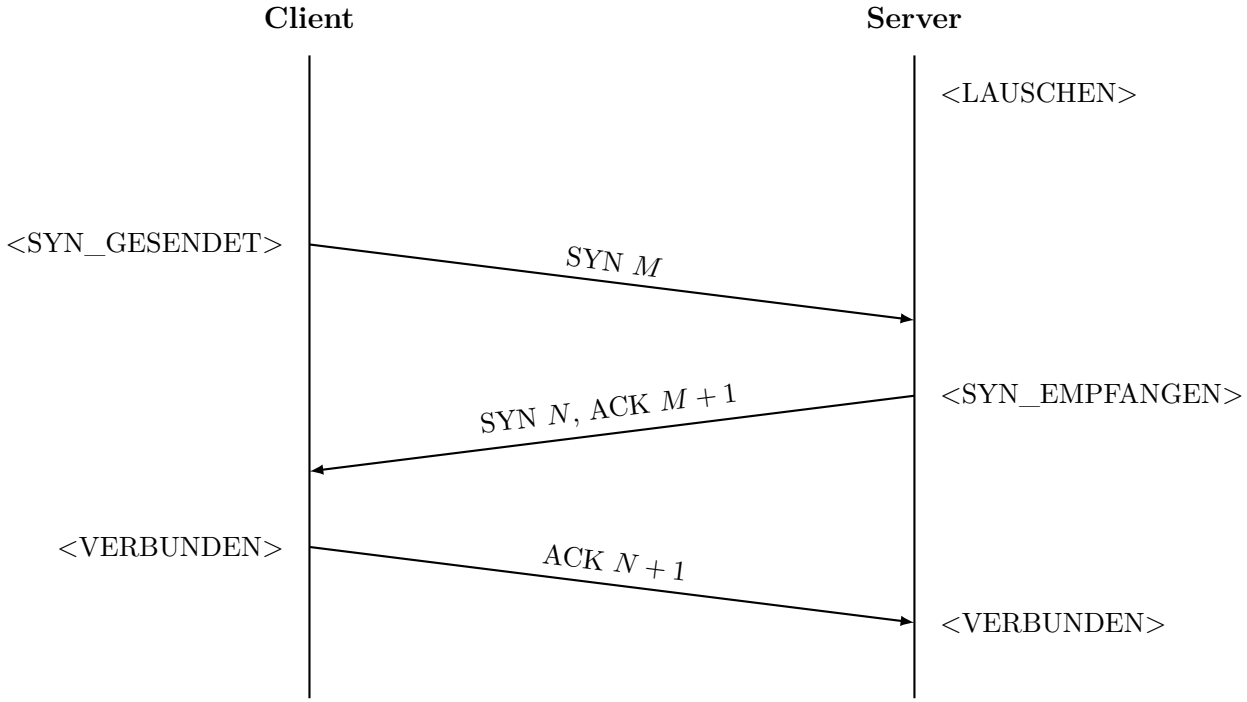

Abbildung 2.1: *Three-Way-Handshake* zum Aufbau einer TCP-Verbindung [15].

Da das TCP Protokoll Daten als Datenstrom (*Stream*) statt einzeln (*Message*) versendet, wird vorher eine Verbindung in einem sogenannten *Three-Way-Handshake* aufgebaut [15] S71/72. Bei diesem werden TCP Pakete mit jeweils unterschiedlichen Werten in den *Control Bits (Flags)* des TCP-Headers nach folgendem Muster ausgetauscht:

2.2 Portscanning

Portscanning, als Art des Netzwerk-Scannings, ist eines der fundamentalen Verfahren in der Netzwerksicherheit zum Auffinden von potenziellen Schwachstellen [12]. 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. [TODO NIST S4-3]

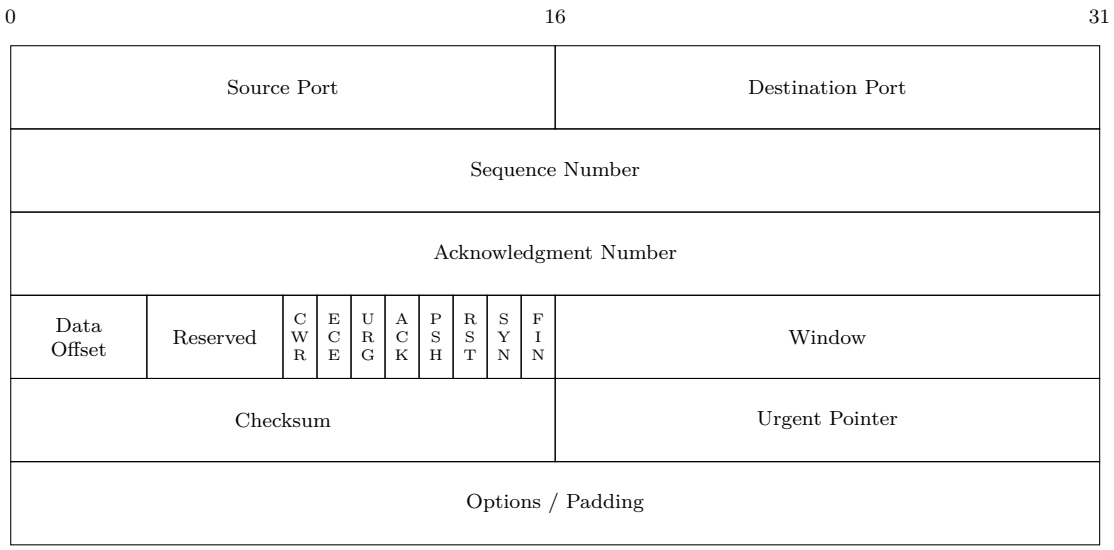

Abbildung 2.2: Aufbau des TCP-Headers nach RFC 9293 [17].

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

- **Vertikales Scannen:** Hierbei wird ein einzelner Ziel-Host (TODO Host erklären) 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 Penetrationtesting 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 [3].

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 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 [12]. 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 [12].

Um antwortende Hosts effizient zu identifizieren, wird das Prinzip des `SYN`-Cookies adaptiert [3], welches ursprünglich als Abwehrmechanismus gegen *Denial-of-Service*-Angriffe spezifiziert wurde [18, S. 8]. Dafür werden verbindungsspezifische Informationen unter Verwendung eines Hash-Algorithmus (z. B. keyed SipHash) 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
```

Da die Validierung einer Antwort somit rein mathematisch erfolgt und keine Speicherung in - oder Zugriff auf eine lokale Zustandstabelle benötigt, wird eine sowohl zeitliche als auch logische Entkopplung von Sende- und Empfangsprozessen erwirkt, was wiederum eine asynchrone Architektur ermöglicht.

Entscheidend für den Scanner sind also folgende Header Felder:

Tabelle 2.1: Relevante TCP 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.

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, 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 [19]. Dies ist für die Entwicklung eines Hochleistungs-Scanners von großem Vorteil.

Ein zentrales Konzept zum Verständnis der Performance-Grenzen ist die Unterscheidung zwischen *User Space* und *Kernel Space* [19] S23:

- **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*. 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 [TODO]. Dies belastet die CPU und wird für den Durchsatz zum Flaschenhals.

2.3.2 Raw-Sockets und Address Families

Als Endpunkt für die Kommunikation werden *Sockets* genutzt [20]. Die traditionelle Netzwerkprogrammierung unter Linux abstrahiert die Komplexität der Netzwerkprotokolle wie TCP. So übernimmt der Kernel dabei vollständig den *Three-Way-Handshake* und das Zustandsverwaltung [TODO]. 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 [21]. Der Entwickler muss die Protokoll-Header selbst konstruieren. Dies ist für *half-open* Port-Scanner essenziell, um individuelle Pakete zu generieren, ohne dass der Kernel automatisch in den Verbindungsaufbau eingreift.

Die Adress-Familien definieren dabei die Interpretation der Adressen und die Ebene des Zugriffs [22]. Hier ein paar Beispiele: <- TODO Besser formulieren

- **`AF_INET` (Standard):** `AF_INET` Operiert auf Layer 3 (IP Ebene), deshalb fügt der Kernel den IP-Header hinzu. Auch das Routing wird (TODO vollständig?) durch den Kernel vorgenommen.
- **`AF_PACKET` (Performant):** Diese Familie ermöglicht direkten Zugriff auf Layer 2 (Ethernet-Ebene). Anwendungen können rohe Ethernet-Frames lesen und schreiben. Dies bietet die vollständige Kontrolle über die Paketerstellung.
- **`AF_XDP` (Sehr Performant):** Hierbei handelt es sich um eine für Hochleistungspaketverarbeitung optimierte Address-Familie. Sie ermöglicht das Senden und Empfangen von Paketen unter fast vollständiger Umgehung des Kernel-Stacks.

2.3.3 Erweiterte Berkley Packet Filter (`eBPF`)

Ursprünglich als *Berkeley Packet Filter* (`BPF`) für Werkzeuge wie *tcpdump* entwickelt, um Pakete effizient zu filtern [23], wurde die Technologie zum erweitert, sodass grundlegend neue Möglichkeiten erschlossen wurden.

eBPF ist eine im Linux-Kernel integrierte virtuelle Maschine (VM), die es erlaubt, benutzerdefinierten Bytecode sicher und effizient im *Kernel*-Kontext auszuführen, ohne Kernel-Module schreiben oder den Kernel neu kompilieren zu müssen [24] S207. 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 TODO cite. 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][26].

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 Kontextwechsel (*Context Switches*) zwischen *Kernel-* und *User-Mode*, sowie die Allokation komplexer Metadatenstrukturen (sk_buff) [2] [25][27]. 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. [27] und Vieira et al. [26] 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.

[2] *Socket Buffer*

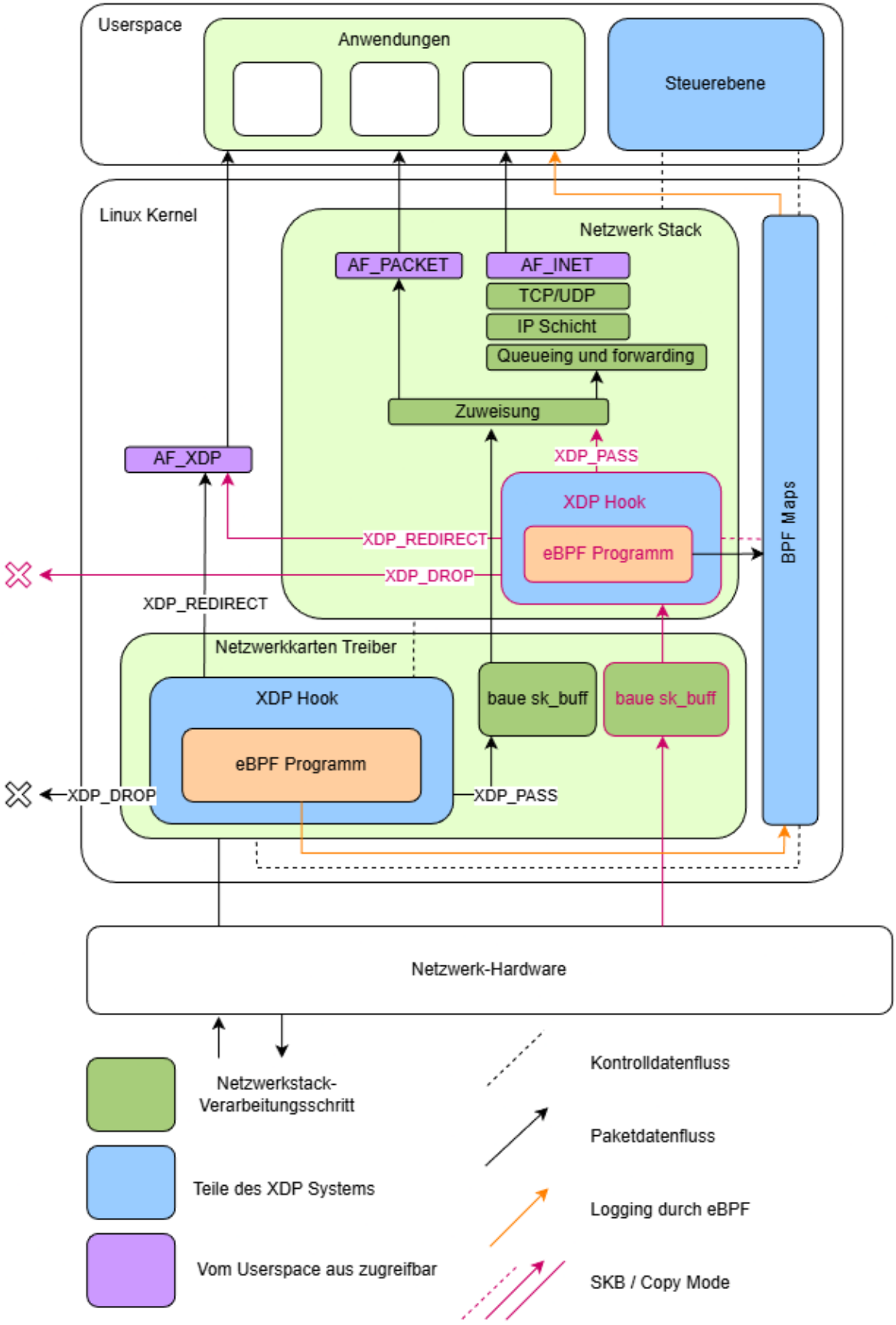

Abbildung 2.3: Der Empfangspfad durch den Kernel bei der Nutzung von `XDP` und `eBPF` (vereinfacht). Orientiert an Høiland et al. [25].

- ***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 an einer späteren Stelle im Netzwerkstack aus. Zwar gehen hier die massiven Performance-Vorteile der Speicherersparnis verloren, jedoch wird sichergestellt, dass `XDP`-Anwendungen auf jeder Hardware funktionsfähig bleiben. Trotzdem profitiert es von effizienteren, sperrfreien *Ring-Buffer*-Strukturen (<- TODO bessere Argumente)

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 [28]. 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 [29].

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 [10], zu vermeiden. Rust hingegen nutzt ein einzigartiges Modell, welches durch drei zentrale Konzepte bestimmt wird:

- ***Ownership*:** Jede Variable hat einen *Owner* (Besitzer). Wird der Besitzer gelöscht, wird auch die Variable gelöscht. Die Variable kann nur einen Besitzer haben [30]. 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 [28].
- ***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 [28].

Die Einhaltung dieser Regeln wird zur Kompilierzeit vom *Borrow-Checker* verifiziert. Außerdem hat Rust noch weitere Konzepte zur Steigerung der Sicherheit, Budgen et al. [28] 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 müssen *Pointer-Arithmetik* in sogenannte `unsafe`-Blöcke ausgelagert werden. Diese dienen als einer Art Umgehungsmöglichkeit der anderen Konzepte und lösen einen Programm-beendenden `panic` aus, welcher 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“ [30] sicher, da die `unsafe`-Blöcke korrekt gekapselt sind [30].

Durch das Zusammenspiel dieser Konzepte können Speicherfehler verschiedenster Art bereits zur Kompilierzeit vermieden werden, was Rust zur sichersten unter den derzeit gängigen Sprachen macht [10]. Allerdings müssen deshalb auch einige Regeln bei der Programmierung beachtet und eingehalten werden, weshalb der Sprache eine steile Lernkurve zugeschrieben wird [31].

2.4.2 Asynchrone Programmierung und Performance von Rust

Die im letzten Abschnitt genannten Konzepte schließen auch *data races*, welche beim Zugriff mehrerer *Threads* [3] auf den gleichen Speicher entstehen können und eine häufige Fehlerquelle in der asynchronen Programmierung darstellen [28] bereits zur Kompilierzeit aus [11]. Dies macht Rust zu einer guten Wahl für sowohl nebenläufige als auch parallele Programmierung.

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[4], für andere Prozesse freigegeben werden. Dadurch kann die Effizienz und die Responsivität des Systems erhöht werden [32], [33].

Die Parallelität hingegen, bezieht sich auf die tatsächliche physikalische Ausführung mehrerer Aufgaben zum gleichen Zeitpunkt [32], was eine entsprechende *Multi-Core*-Hardware

[3] Untergeordnete Arbeitseinheiten eines Prozesses
[4] Input-/Output-Operationen

voraussetzt [34]. Der Vorteil der Parallelität liegt in der Leistungssteigerung und der Maximierung des Datendurchsatzes bei rechenintensiven Problemen [32], [33].

Rusts Konzepte zur Performancesteigerung

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* [31], 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“ [35].

Dazugehörige Konzepte sind beispielsweise die Eliminierung von Laufzeit-Overhead durch die Vermeidung eines zur Laufzeit arbeitenden *Garbage Collectors* [10] [31], die *Monomorphisierung* um die Typen oder Größen generischer Strukturen wie z.B. `Vec` [5] oder `Option` nicht mehr während der Laufzeit bestimmen zu müssen [31], oder die Bereitstellung eigener Iteratoren, welche die Leistung manuell geschriebener Schleifen oft übertrifft [31].

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 [11] [28] [31].

[5] 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 resultierende Problematik betrachtet. Es folgt ein kurzer Überblick über weitere relevante Forschungsarbeiten zu ebenfalls relevanten Forschungssträngen sowie die Auswahl der Scanner, welche später als Vergleichsobjekte dienen vorgestellt.

3.1 Historische Entwicklung des horizontalen Netzwerk-Scannings

Ursprüngliche Netzwerkscanner wie *Nmap* [12] 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 [3].

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 [3]. Spätere Arbeiten, wie *Zippier ZMap*, optimierten diesen Ansatz weiter, um auch bis zu 10-Gbps Leitungen auszulasten [36] und somit die anhaltende Relevanz des *ZMap*-Projektes zu unterstreichen.

3.1.1 Der Standard Scanner: ZMap

In der wissenschaftlichen Literatur gilt *ZMap* [3] 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* [36], 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 performancetechnisch limitiert sind [2].

- **Zustandslose Architektur:** *ZMap* nutzt das Prinzip der `SYN`-Cookies 2, 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. [3].

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 [37]. Der Nachteil ist jedoch die hohe Komplexität, die exklusive Belegung von CPU-Kernen und die schwierige Integration in bestehende Systemumgebungen.

- **Eigener TCP-Stack im *User-Space* (*Masscan*):** Der Scanner *Masscan* [4] 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* [1] zu steuern. Dadurch kann *Masscan* deutliche Performancegewinne gegenüber *ZMap* erreichen [38].

[1] TODO

- **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 [8]. Dies erfordert jedoch spezialisierte Hardware. In dieser Untersuchung wurde mit Raten von 40Gbps getestet, wobei jedoch Raten von einem Terabit oder mehr theoretisch laut Li et al. 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 wie *Netmap* [39] welches bereits 2012 Konzepte wie *Shared Memory* [2] und *Zero-Copy* einführte aber den Netzwerkstack eher ersetzte oder *DPDK* (siehe 3.2), die Arbeit zu `XDP` (*eXpress Data Path*) einen Paradigmenwechsel darstellt. Høiland-Jørgensen et al. zeigen, dass 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 [38] [40], fokussieren sich aber eher auf die Trefferrate, welche in der Arbeit nicht priorisiert wird (1.3). Außerdem ist die Gestaltung der Testumgebung ist bei beiden 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 [41]. Gonzalez et al. entwickelten einen UDP Treiber für Linux und verglichen diesen mit einem ähnlichen C-Treiber [42]. Moon et al. erstellten einen *NAT* (Network Address Translator) und testeten den Durchsatz [43]. 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 [44].

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 [10], [28], [45].

[2] Zwischen Userspace und Kernelspace 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

- **Vergleichsobjekt aus der Forschung:** Als Vergleichsobjekt aus der Forschung dient ZMap. Wie bereits in 3.1.1 gezeigt, repräsentiert er das Standard-Vergleichsobjekt aus der Forschung und ist gleichzeitig einer der zahlreichen Varianten, welche in C geschrieben ist.
- **Rust Alternative:** Um auch ein Vergleich mit bereits bestehende Alternativen in Rust aufzuzeigen ... TODO
- **Alternative aus der Wirtschaft:** Da die Forschungsfrage unter anderem darauf abzielt die Eignung des implementierten Rust Scanners für den produktiven Einsatz zu untersuchen, soll dieser auch mit einem Scanner, welcher aktuell in der Wirtschaft genutzt wird, aber eigens in Go entwickelt wurde. Dies stärkt außerdem die Vergleichsbreite, da somit auch eine Sprache mit automatischer Speicherverwaltung in den Vergleich integriert ist.

3.5 Problematik bisheriger Ansätze

3.5.1 Problematik 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 zentrales Problem ist die fehlende intrinsische Speichersicherheit von C [TODO]. 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 [9]. Darüber hinaus geht die Leistungsfähigkeit von C oft zu Lasten der Wartbarkeit und Entwicklungseffizienz [11]. 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 [11]. 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].

3.5.2 Problematik alternativer Ansätze

Ein weiteres wesentliches Problem 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 [TODO]. Durumeric et al. merken an, dass die Wartung solcher spezialisierten Treiber 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.

3.5.3 Lösungsansatz

Die Nutzung von Rust stellt eine vielversprechende Lösung 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 von `XDP` und `eBPF` das Problem der fehlenden Kernelintegration für die Hochleistungspaketerarbeitung.

Die Kombination dieser Technologien und Werkzeuge stellt in dem Kontext des horizontalen *high-speed* Netzwerk-Scannings 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 korrekt berechnet werden, um vom Zielsystem als legitime Verbindungsanfragen akzeptiert zu werden.
- **/F-02/ Senden von Paketen:** Das System muss in der Lage sein, TCP Pakete an andere Zielsysteme zu senden.
- **/F-03/ Empfang von Paketen:** Das System muss in der Lage sein, eingehende Netzwerkpakete abzufangen und zu untersuchen.
- **/F-04/ Zustandsloses Scanning:** Zwischen den Sende- und Empfangskomponenten darf keine Kommunikation über die Zielsysteme bestehen, sodass kein Zustand über bereits kontaktierte Zielsysteme gespeichert wird.
- **/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 auf Zielsystem:** Nach der Identifikation eines offenen Ports, sollte der Scanner ein `RST`-Paket senden, um die halb-offene Verbindung auf dem Zielsystem sauber zu schließen und Ressourcenfreigabe zu ermöglichen.
- **/F-07/ Endausgabe und Vermeidung von Duplikaten:** Die Endausgabe muss die ausgewerteten Scanergebnisse inklusive IP-Adresse und Ziel-Port der offenen Zielsysteme enthalten und muss von Duplikaten bereinigt sein.
- **/F-08/ Durchsatzlimitierung (*Rate Limiting*):** Das Programm muss in der Lage sein, eine angegebene Durchsatzrate (in Byte pro Sekunde) nicht zu überschreiten, sodass eine konsistente Performanz-Messung möglich ist.
- **/F-9/ Anpassung an bestehende Infrastruktur:** 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 dem Forschungsziel, der Performance-Maximierung und der Verwendung von Rust ab.

- **/NF-01/ Maximierung des Durchsatzes:** Das System soll in der Lage sein, die verfügbare Bandbreite einer Standard-Gigabit-Schnittstelle bestmöglich auszunutzen. Zielgröße ist ein Sende-Durchsatz im mindestens fünfstelligen Paket-pro-Sekunde-Bereich, um mit etablierten Tools vergleichbar zu sein.
- **/NF-02/ Asynchrone Architektur:** Die Anwendung muss Gebrauch von Last-verteilenden Maßnahmen in Form von nebenläufiger Programmierung machen, um die Auslastung zu verteilen und Performance somit zu steigern.
- **/NF-03/ Nutzung moderner Kernel-Mechanismen:** Zur Steigerung der Performance und Forschungsrelevanz sollen fortschrittliche Linux-Mechanismen zur Paketverarbeitung, spezifisch `AF_XDP` oder `eBPF`, evaluiert und implementiert werden.
- **/NF-04/ Speichersicherheit:** Die Gesamtarchitektur soll die Sicherheitsgarantien von Rust wahren. `unsafe`-Blöcke können, falls nötig, genutzt werden (z.B. für systemnahe Netzwerkoperationen) sollten aber möglichst vermieden oder durch die Nutzung von externen Bibliotheken ersetzt werden, da diese intern `unsafe`-Blöcke häufig sicher kapseln [30].
- **/NF-05/ Minimale Ressourcennutzung:** Die Architektur und gewählten Technologien sollten die Ressourcennutzung (CPU-Zeit, RAM Verbrauch) neben der Durchsatzgeschwindigkeit nach /NF-01/ priorisieren und somit minimieren. Keine der beiden Anforderungen darf aufgrund der anderen stark vernachlässigt werden.

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 Performanz-steigernde Maßnahmen addressiert.

4.3 Untersuchungsdesign

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

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

4.3.1 Metriken

Der Begriff „Performanz-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 [46].

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:

Tabelle 4.1: Metriken zur Performanz-orientierten Evaluation der SYN-Scanner

Metrik	**Einheit**	**Beschreibung**
Paketrate	Pakete pro Sekunde (pps)	Durchschnittliche Anzahl der erfolgreich an den Netzwerkadapter übergebenen Pakete pro Sekunde.
CPU Auslastung	Prozent (%)	Die prozentuale Auslastung der CPU-Kerne, aufgeteilt in *User-Space* und *Kernel-Space*.
RAM Verbrauch	Megabyte (MB)	Der *Resident Set Size* (RSS), also der Anteil des Arbeitsspeichers, der physisch durch den Prozess belegt wird.

4.3.2 Testumgebung

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.

Der Aufbau besteht aus zwei physischen Knoten:

1. **Der Scanner-Knoten:** Auf diesem System wird der zu evaluierende Rust-Prototyp ausgeführt. Gleichzeitig erfolgt hier die Erfassung der in Tabelle 4.1 definierten Metriken.

2. **Der Ziel-Knoten:** Dieses System simuliert ein Zielnetzwerk. Um sicherzustellen, dass das Zielsystem bei hohen Paketraten nicht zum Flaschenhals wird, wird ein dafür entwickeltes `eBPF/XDP`-Programm verwendet.

Der Ziel-Knoten verfügt über einen konfigurierbaren Parameter für die Antwortwahrscheinlichkeit, um verschiedene Netzwerkszenarien zu simulieren. Eingehender und ausgehender Verkehr wird über eBPF-Maps atomar gezählt, um die vom Scanner berichtete Senderate auf der Empfängerseite, sowie die zurückgeschickten Antworten zu validieren.

4.3.3 Testablauf

Der experimentelle Ablauf gliedert sich in zwei definierte Szenarien, um sowohl die funktionale Stabilität als auch die maximalen Leistungsgrenzen des Scanners zu evaluieren. Für beide Szenarien werden die zuvor definierten Ressourcen-Metriken (CPU, RAM) sowie die Paketraten aufgezeichnet.

1. **Anforderungsvalidierung** Im ersten Szenario wird der Scanner mit einer festen, limitierten Senderate von 100.000 Paketen pro Sekunde (pps) betrieben. Dieser Test dient als Referenzmessung (*Baseline*), um die Erfüllung der funktionalen Anforderungen zu verifizieren. Bei 100.000 pps handelt es sich um eine signifikante Last, die jedoch weit unterhalb der theoretischen Sättigungsgrenze einer Gigabit-Leitung liegt. Ziel ist der Nachweis, dass der Scanner bei dieser definierten Last über einen längeren Zeitraum stabil arbeitet und ein deterministisches Ressourcenverhalten zeigt. Dies stellt sicher, dass gemessene Ineffizienzen im zweiten Szenario tatsächlich auf die Hardware- oder Software-Limitierung und nicht auf Implementierungsfehler zurückzuführen sind. Um die Erfüllung der funktionalen Anforderungen zu prüfen, wird der Ziel-Knoten außerdem auf eine gewisse Anzahl an Paketen mit validen `SYN/ACK` Paketen Antworten.

2. **Ermittlung der Performanzgrenze** Im zweiten Szenario wird die künstliche Drosselung des Scanners aufgehoben. Der Scanner versucht, Pakete so schnell zu generieren und zu versenden, wie es die CPU oder das Netzwerkinterface zulassen. Dieses Szenario dient der Ermittlung des maximalen Durchsatzes. Durch den Vergleich der CPU-Auslastung bei maximaler Last lässt sich feststellen, ob die Limitierung durch die Rechenleistung der Adressgenerierung oder durch den Overhead der Paketübertragung entsteht und wie viele Ressourcen dabei genutzt werden.

3. **Test mit realen Eingabeparametern** Im dritten Test werden Parameter gewählt, die in einem realen Szenario von Vorteil wären. Dabei soll eine feste Senderate gewählt werden, um die Vergleichbarkeit zu erhöhen. Um die Antwortrate in einem realen Szenario zu erhöhen, indem der Scan schlechter von dem normalen Netzwerkrauschen zu unterscheiden ist, werden in der Regel mehrere verschiedene *Source*-IP-Adressen zum Senden gewählt. Auch das Scannen mehrerer statt eines einzelnen *Destination*-Ports hilft dabei [2].

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 Übersicht genutzter Crates

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

Tabelle 5.1: Genutzte Crates

Crate	**Version**	**Nutzung**
tokio	1.47.1	Nutzung für asynchrone Komponenten, Kommunikation über *Channels*, Parsen des Standard-Inputs 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 Lock-Handling selbstständig durch

5.2 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
                finish_broadcaster.rs
                parser_std_in.rs
                scan_job.rs
            /shared
                helper.rs
                types_and_config.rs
        main.rs
    cargo.toml
/xdp-common
    /src
        lib.rs
/xdp-ebpf
    /src
        main.rs
...
```

In jedem Ordner ist eine `mod.rs` Datei zu finden, welche hier zugunsten der Lesbarkeit entfernt wurde. Diese Dateien dienen dazu, ein Verzeichnis als 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 Antowortpakete sind zum einen `xdp-ebpf`, `xdp-common` und `capturing_packets` zuständig. Letzteres beinhaltet die Logik zum Empfangen der Daten im *Userspace*, 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 *Userspace* weiterleitet. Die Komponente `job_controlling` ist für die Funktionsfähigkeit des Programmes auch essenziell, hat aber hauptsächlich die Aufgabe, die anderen Komponenten korrekt zu vernetzen.

5.2.1 Logische Komponenten des Scanners (`scanner`)

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

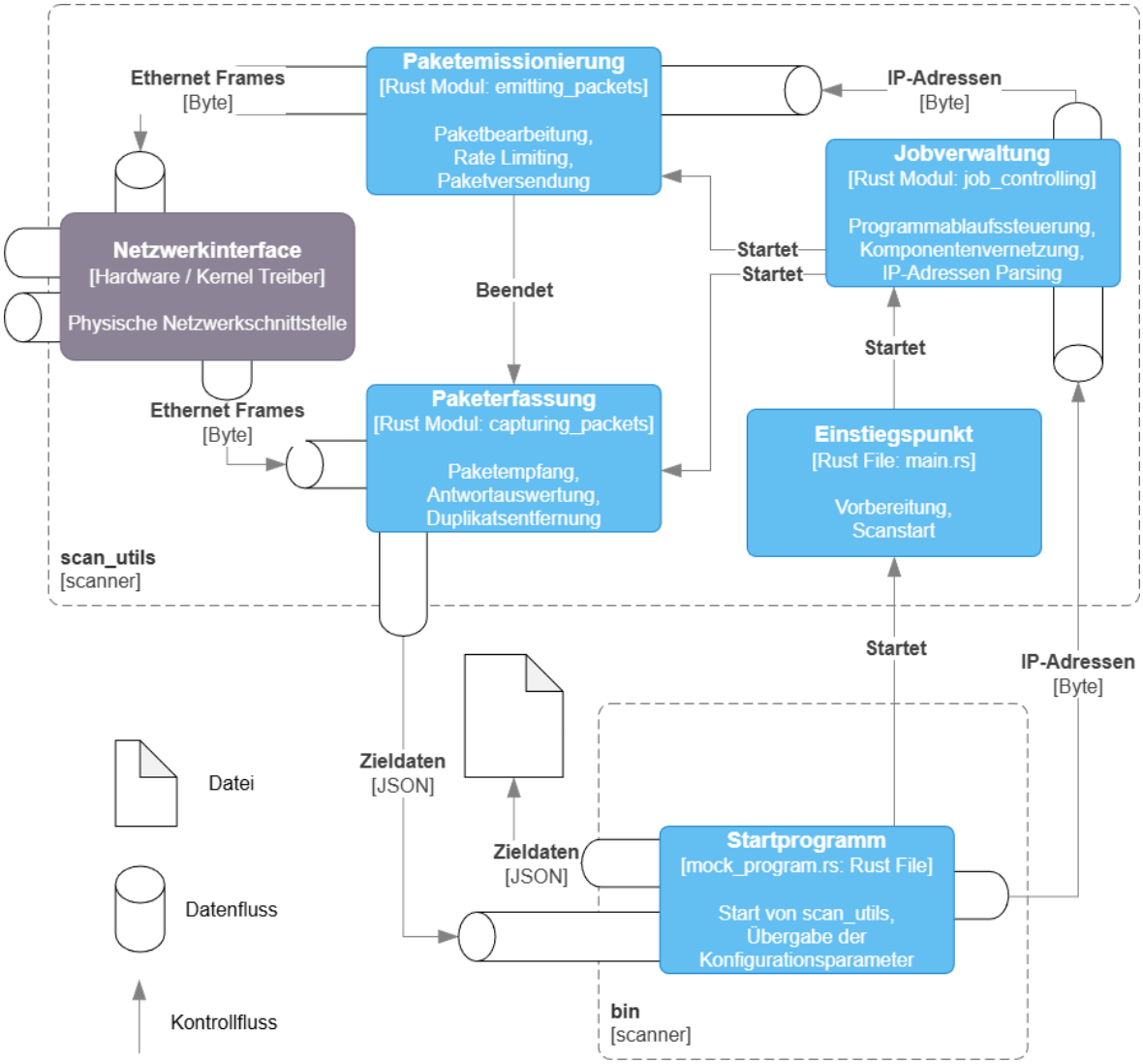

Abbildung 5.1: Diagramm logischer Komponenten des `scanner` Verzeichnisses (vereinfacht)

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 Userspace und Linux Kernel nicht explizit behandelt. 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 oder DRV) 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-Modus fast alle, Schritte des regulären Netzwerkstacks eingespart. Die Ergebnisse der Untersuchung im eBPF-Programm werden bei validen Paketen in eine BPF 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 Userspace 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.

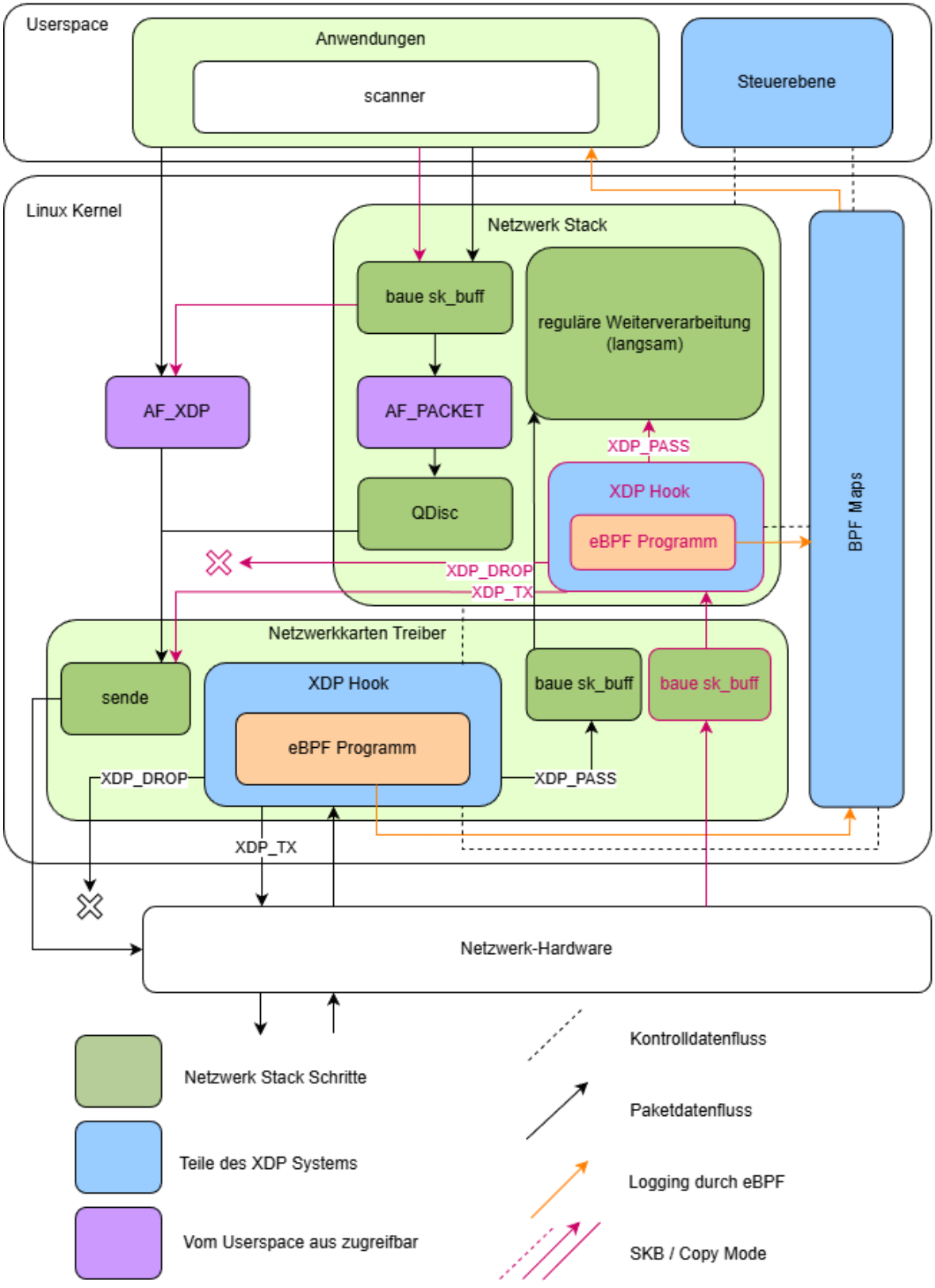

Abbildung 5.2: Weg der Pakete durch den Linux Kernel (vereinfacht)

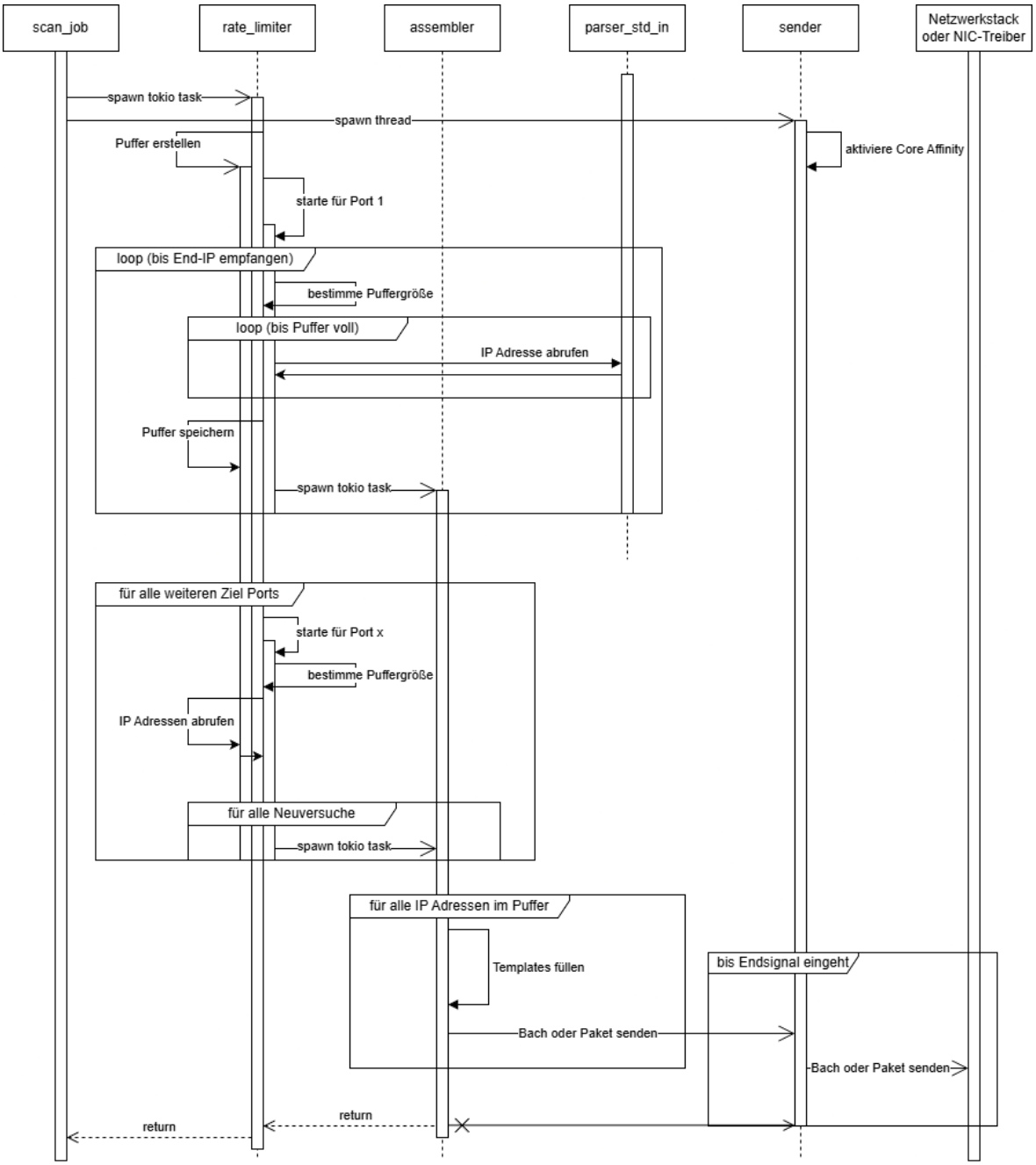

Abbildung 5.3: Ablaufes und Funktionsweise der `emitting_packets`-Komponente (vereinfacht)

5.3 Implementierung und Funktionsweise der Komponenten

5.3.1 Paketemissionierung (emitting_packets)

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 ruft 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 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 Betriebsystem-*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 [47]. Diese Nebenläufigkeit wird hier genutzt, um entsprechend der aktuellen Senderate *Assembler* zu erzeugen, die nicht den gesamten Betriebssystem-*Thread* blockiert, 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 (TODO maybe hierauf genauer eingehen).

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, der *Sequence Number* und berechnet die Checksummen des *IP-* und *TCP-Headers* 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 [48]. 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 Feldzuweisung der *Header* immer gleich sind [16], [17]. Somit 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 Kanäle 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.

5.4 Ergebnisverarbeitung (capturing_packets)

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 muss. 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

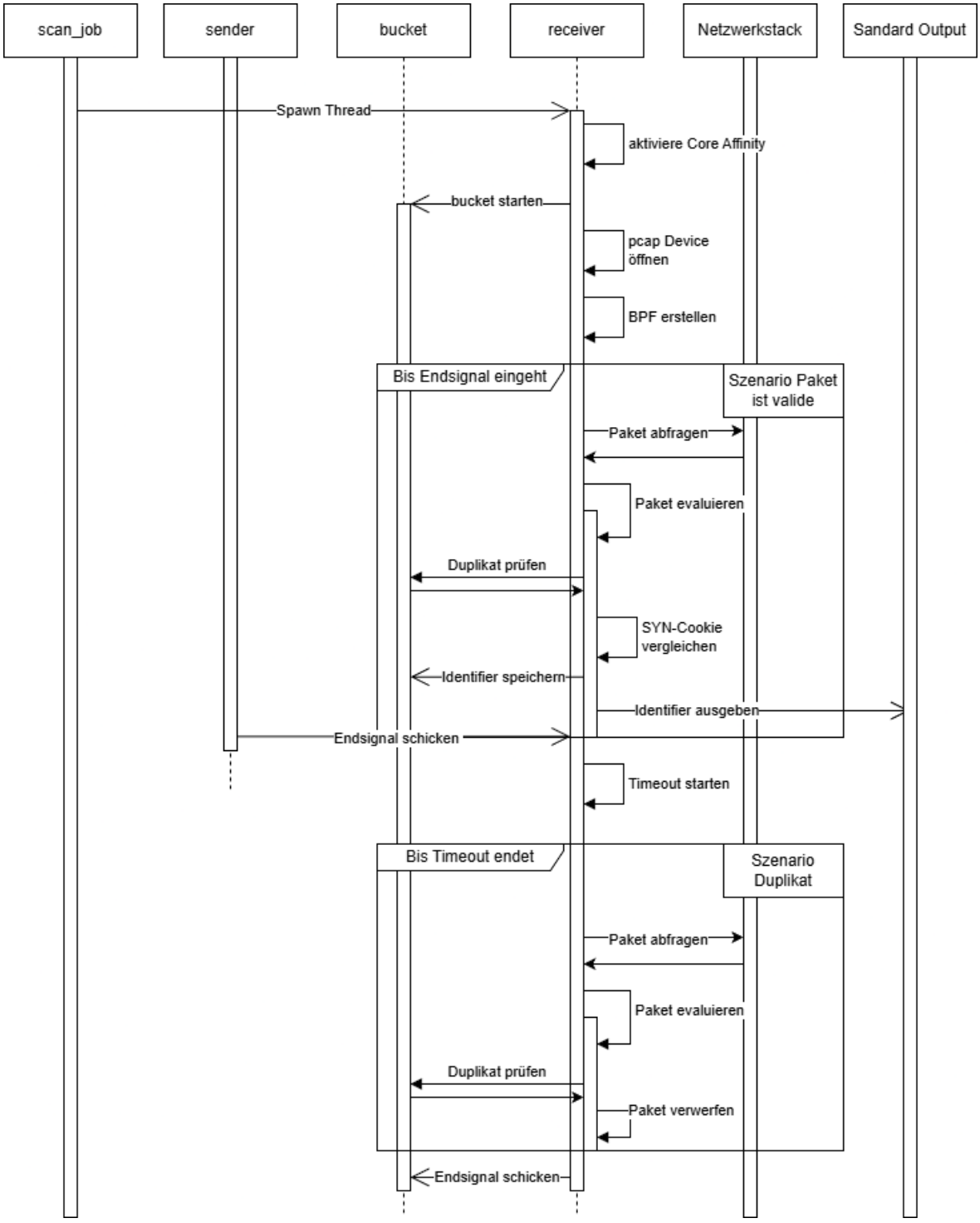

Abbildung 5.4: Exemplarisches Diagramm zur Funktionsweise der `capturing_packets`-Komponente (vereinfacht)

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 Performanz 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 2.3), selbst, wenn durch den BPF irrelevante Pakete hardwarenah herausgefiltert werden.

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 der tokio Crate 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 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 Reset-Antwort zurückschickt [TODO Quelle?].

5.5 Programmstart (`main.rs`, `bin/mock.program`) und Jobverwaltung (job_controlling)

Den 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 Komponente.

5.5.1 Funktionsweise

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 der Einfachheit halber 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 einen entscheidenden Beitrag zur Erfüllung der Anforderung /F-01/ leisten, werden mithilfe des pnet Crates erstellt, da dies 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 Sequenznumber gesetzt. Es wird für jede Quell-IP ein Template angelegt, da diese lediglich der Streuung der Paketquellen zur Verschleierung des Scans und somit der Erhöhung der Trefferrate dienen.

Die Ziel-IP-Adressen werden in Batches übertragen weil .....

Einstiegspunkt (`main.rs`)

In der Startfunktion werden zum einen die Konfigurationsargumente des Mock

Finish-Broadcaster (`finish_broadcaster`)

Standard-Input Parser (`parser_std_in`)

Scanjob (`scanjob.rs`)

5.6 eBPF

Sollte es sich um kein Duplikat handeln wird bei der anschließenden Prüfung des SYN-Cookies die Hash-Berechnung der vier in **??** genannten Werte mit dem gleichen geheimen Schlüssel nach **??** erfolgen. Somit wird Anforderung /F-05/ erfüllt.

5.6.1 Funktionsweise

5.7 Qualitätssicherung

5.8 Testumgebung

5.9 Contiki-Verzeichnisstruktur

Die Contiki-Verzeichnisstruktur besteht aus verschiedenen Unterverzeichnissen, die verschiedene Bedeutungen haben.

Codeauszug 5.2: Contiki-Verzeichnisstruktur mit ausgewählten Unterverzeichnissen

```
1  /apps
2      /ftp
3      /ping6
4      /telnet
5      /telnetd
6      /twitter
7      /webserver
8      /webserver-nano
9      ...
10 /core
11     /lib
12     /net
13     /sys
14     ...
15 /cpu
16     /arm
17     /avr
```

```
18      ...
19  /doc
20  /examples
21      /webserver-ipv6-raven
22      ...
23  /platform
24      /avr-raven
25      /avr-zigbit
26      ...
27  /tools
```

Das Verzeichnis /core beinhaltet das Kernstück von Contiki. Hier wird zum Beispiel das System im Unterverzeichnis /sys definiert. Das beinhaltet das Prozesshandling und Protothreads sowie verschiedene Timer. Ebenfalls befindet sich hier im Unterverzeichnis /net der Netzwerk-Stack aufbauend auf uIP. Zum Netzwerk-Stack gehört auch IEEE 802.15.4 und 6LoWPAN. Im Unterverzeichnis /lib stehen verschiedene libraries zur Verfügung.

In den beiden Verzeichnissen /cpu und /platform ist die unterschiedliche Hardware beschrieben, die von Contiki unterstützt wird. Ein Programm, das auf einem Zigbit-Modul vom Hersteller Atmel geladen werden soll, verwendet die Verzeichnisse /platform/avr-zigbit und /cpu/avr.

Das Verzeichnis /examples beinhaltet Beispielprojekte. Hier gibt es ein Projekt webserver-ipv6-raven. Anhand dieses Beispiels wird deutlich, was notwendig ist, um eine Webserver-Applikation für das Ravenboard zu kompilieren.

Im Hauptverzeichnis gibt es eine Datei Makefile.include. Diese Datei ist Teil des Contiki-Makefile-Systems. Sie wird innerhalb eines Contiki-Projektes aufgerufen und bindet, abhängig von der Konfiguration des Projektes, die richtigen Dateien des Contiki-Systems ein.

5.9.1 Übersetzung eines Contiki-Programms

Ein Programm wird in Contiki mithilfe des Befehls „make“ übersetzt. Am Beispiel des Projekts webserver-ipv6-raven soll exemplarisch gezeigt werden, wie das Makefile System funktioniert. Durch den Befehl „make“ wird die Datei Makefile im gleichen Verzeichnis abgearbeitet.

Codeauszug 5.3: Auszug aus examples/webserver-ipv6-raven/Makefile

```
1  ifndef TARGET
2    TARGET=avr-raven
3    MCU=atmega1284p
4  endif
5  all:
6    ${MAKE} -f Makefile.webserver TARGET=$(TARGET) NOAVRSIZE=1 webserver6.elf
```

Hier wird das TARGET, also die Plattform, und die MCU, die Microcontroller Unit, gesetzt und dann die Datei Makefile.webserver aufgerufen. Dabei wird der Parameter NOAVRSIZE gesetzt, um beim Übersetzen eine zusätzliche Ausgabe zur Speicherbelegung (avr-size) zu unterdrücken. Das Programm wird als Datei webserver6.elf erstellt.

Codeauszug 5.4: Auszug aus examples/webserver-ipv6-raven/Makefile.webserver

```
1 all: webserver6
2 APPS=raven-webserver raven-lcd-interface
3 UIP_CONF_IPV6=1
4 CONTIKI = ../..
5 include $(CONTIKI)/Makefile.include
```

In der Datei Makefile.webserver werden die Applikationen raven-webserver und raven-lcd-interface über die Variable APPS eingebunden. Die Compiler Variable `UIP_CONF_IPV6` wird gesetzt, um IPv6 zu aktivieren, weiterhin wird das Haupt-Makefile `Makefile.include` eingebunden.

Codeauszug 5.5: Auszug aus examples/webserver-ipv6-raven/webserver6.c

```
1 #include "webserver-nogui.h"
2 /*---------------------------------------------------------------------*/
3 AUTOSTART_PROCESSES(&webserver_nogui_process);
4 /*---------------------------------------------------------------------*/
```

In der Datei webserver6.c wird ausgewählt, welche Contiki-Prozesse automatisch gestartet werden sollen. In unserem Beispiel ist das der Prozess „webserver_nogui_process“. Dieser Prozess ist Teil der Applikation raven-webserver.

5.9.2 Programmieren eines Mikrocontrollers

Durch Übersetzung des Contiki-Beispielprojektes webserver-ipv6-raven wird eine Datei „webserver6.elf“ im ELF Format erzeugt. Hiervon wird eine Kopie „webserver6-avr-raven.elf“ erstellt. Diese Datei enthält Informationen darüber, was in den Flash und in den EEPROM des AVR Mikrocontrollers geladen werden muss. Ebenfalls beinhaltet es Informationen über das Setzen der Fuse Bits. Fuse Bits sind Einstellungen des Mikrocontrollers, die nicht von der Software geändert werden können. Sie schalten gewisse Funktionen, zum Beispiel woher die Taktfrequenz bezogen wird, ein oder aus.

Codeauszug 5.6: Auszug aus examples/webserver-ipv6-raven/Makefile

```
1 TARGET=avr-raven
2 MCU=atmega1284p
3 OUTFILE=webserver6-$(TARGET)
4 avr-objcopy -O ihex -R .eeprom -R .fuse -R .signature \
5           $(OUTFILE).elf $(OUTFILE).hex
6 avr-size -C --mcu=$(MCU) $(OUTFILE).elf
```

Durch zusätzliche Befehle im Makefile wird eine Datei „webserver6-avr-raven.hex“ erzeugt. Diese Datei enthält den Inhalt, der in den Flash geschrieben werden soll, im Intel-HEX-Format. Mit dem „avr-size“-Befehl wird die Anzahl der Bytes angezeigt, die im Flash, im RAM und im EEPROM benötigt werden. Dies kann mit dem zur Verfügung stehenden Speicher verglichen werden.

Codeauszug 5.7: Ausgabe vom Befehl „avr-size“

```
1  AVR Memory Usage
2  ----------------
3  Device: atmega1284p
4
5  Program: 70874 bytes (54.1% Full)
6  (.text + .data + .bootloader)
7
8  Data: 13013 bytes (79.4% Full)
9  (.data + .bss + .noinit)
10
11 EEPROM: 63 bytes (1.5% Full)
12 (.eeprom)
```

Durch Hinzufügen eines zusätzlichen Befehls ins Makefile kann eine Datei „webserver6-avr-raven_eeprom.hex“ erzeugt werden. Diese Datei enthält dann den Inhalt im Intel-HEX-Format, der in den EEPROM geschrieben werden soll. Mit den beiden Dateien im Intel-HEX-Format ist es möglich, den Mikrocontroller mit einem Programmiergerät zu beschreiben, das kein ELF Format lesen kann.

Codeauszug 5.8: Befehl um eeprom.hex zu erzeugen

```
1  avr-objcopy -O ihex -j .eeprom \
2             -set-section-flags=.eeprom="alloc,load" --change-section-lma \
3             .eeprom=0 $(OUTFILE).elf $(OUTFILE)_eeprom.hex
```

Mittels des Programms Atmel Studio 6 werden wahlweise die erzeugten Dateien im Intel-HEX-Format oder die erzeugte Datei im ELF Format in den Flash und in den EEPROM des Mikrocontrollers programmiert. Zu Beginn wurde das Beispielprojekt webserver-ipv6-raven auf den Mikrocontroller AT-Mega1284P eines Atmel Ravenboard programmiert. Dabei wurde die Programmierschnittstelle ISP und das Programmiergerät Atmel STK500 verwendet.

Später bei der entwickelten Hardware wurde die Programmierschnittstelle JTAG und das Programmiergerät Atmel JTAGICE3 verwendet.

Kapitel 6: Tests und Evaluation

In diesem Kapitel wird die erstellte Lösung getestet und die Entwicklung validiert. Dazu ist die Testumgebung zu beschreiben.

Ziel ist der glaubhafte Nachweis der Funktionsfähigkeit, bzw. die quantitative Bewertung der erstellten Lösung.

Ein wichtiges Kriterium ist die Nachvollziehbarkeit der Tests. Das heißt, die Tests müssen so beschrieben sein, dass der Leser der Arbeit die Ergebnisse eigenständig wiederholen und validieren kann.

6.1 Herleitung von Test-Cases

Herleitung der Testmethodik.

6.1.1 Funktionale Tests

...

6.1.2 Leistungstests/Performancetests

...

6.2 Bewertung der Ergebnisse

...

Kapitel 7: Fazit und Ausblick

In diesem Kapitel wird die erstellte Lösung begutachtet. Es werden Anwendungsmöglichkeiten, allgemein von Smart Objects und speziell für die Implementierung, behandelt. Mögliche Erweiterungen werden vorgeschlagen und ein Fazit der Arbeit wird gezogen.

7.1 Analyse der Implementierung

Die erstellte Lösung wird kritisch begutachtet. Dafür werden zuerst Kostenbetrachtungen bezogen auf die Hardware und auf die Software durchgeführt. Die Ergebnisse fließen in die abschließende Bewertung mit ein.

7.1.1 Kostenbetrachtung

Um in der Bewertung der Implementierung die Kosten einschätzen zu können, sollen hier zum einem die Hardwarekosten vorgestellt werden. Dazu sollen die einzelnen Materialkosten der verwendeten Bauteile zusammengefasst werden. Um die gesamten Hardwarekosten ermitteln zu können, müssen auch Entwicklungskosten und Fertigungskosten berücksichtigt werden. Fertigungskosten beinhalten dabei die eigentlichen Arbeitsstunden, die für die Fertigung aufgebracht werden müssen, als auch die Investitionskosten für den Fertigungsarbeitsplatz. Weil das sehr abhängig von der Art der Fertigung ist, sei es eine Einzelstückfertigung oder eine Serienfertigung, werden diese Kosten hier nicht näher betrachtet. Die Hardwarekosten werden rein durch die Materialkosten repräsentiert.

Zum anderen sollen die Softwarekosten abgeschätzt werden. Dies wird durch die Ermittlung der Entwicklungskosten für die Contiki-Applikation iec104 erreicht. Anhand eines angenommenen Stundensatzes für einen Softwareentwickler und den aufgewendeten Stunden werden die Kosten für die Entwicklung berechnet.

Materialkosten

In Tabelle 7.1 sind die Bauteile aufgelistet, die zum Erstellen der Steckdose verwendet wurden. Sie benennt das Bauteil und zeigt die Anzahl an, die verbaut wurden. Die Bauteile wurden – bis auf die Tchibo Steckdose – vom Handelsunternehmen Mouser Electronics bezogen. In der dritten Spalte ist die entsprechende Artikelnummer aufgelistet. Die vierte

Tabelle 7.1: Materialkosten Steckdose

#	**Bauteil**	**Artikelnummer**	**Einzel-preis in €**	**Massen-preis in €**
Zigbit-Platine				
1	ATZB-24-A2	556-ATZB-24-A2	25,43	14,42
1	Widerstand 100kOhm	71-CMF60100K00FKEB	0,206	0,099
1	Widerstand 1kOhm	71-CMF551K0000FHEK	0,116	0,005
1	Kondensator 3,3uF	667-EEU-HD1H3R3	0,165	0,104
1	Spannungsregler LP2950CZ-3.0	926-2950CZ-3.0/NOPB	0,676	0,27
1	Steckverbinder FFC 6 Pin	538-52271-0679	1,30	0,583
2	Steckverbinder FFC 18 Pin	538-52271-1879	1,71	0,94
Steckdosen-Platine				
1	Kondensator X2 275V	80-R46KN368050M2M	0,66	0,263
1	Widerstand 560Ohm, 5Watt	594-AC05W560R0J	0,38	0,198
2	Widerstand 1MOhm	594-MRS251M1%TR	0,074	0,038
2	Gleichrichterdiode	512-1N4004	0,076	0,025
2	Zener-Diode 24V	512-1N4749ATR	0,203	0,036
1	Kondensator 470uF	667-EEU-FR1E471YB	0,248	0,152
1	NPN-Transistor BC547B	512-BC547B	0,186	0,05
1	Widerstand 220kOhm	271-220K-RC	0,125	0,012
Steckdosen-Gehäuse				
1	Tchibo Digitale Zeitschaltuhr	4 043002 669758	5,99	5,99
Gesamtpreis			**38,61**	**24,27**

Spalte zeigt den Preis bei dem Bezug von nur einem Bauteil. Bei einer Massenfertigung der Steckdose werden andere Preise angeboten. Der Stückpreis bei einer Bestellung von 1000 Fertigungssätzen ist in der letzten Spalte aufgelistet. Die Preise wurden am 06.05.2013 abgefragt und sind inklusive Mehrwertsteuer angegeben.

Die Gesamtmaterialkosten der verwendeten Bauteile betragen 38,61€. Nicht berücksichtigt sind Verbrauchsmaterialien. Dazu zählen die Leiterplatte für die Zigbit-Platine, die aus einer vorhandenen Lochrasterplatine ausgesägt wurde, Leitungen für die Verkabelung und Lötzinn. Diese Kosten sollen hier vernachlässigt werden. Bei einer Kalkulation für eine Fertigung in einem Unternehmen müssen diese als Materialgemeinkostenzuschlag zu den Materialkosten hinzugefügt werden. Zusammen mit den Fertigungskosten, also die Arbeitskosten, die zum Fertigen der Steckdose notwendig sind, und den Verwaltungs- und Vertriebsgemeinkostenzuschlag ergeben sich die Selbstkosten je Stück.

Entwicklungskosten iec104

Um den Aufwand von einer Contiki-Applikation abzuschätzen, sollen beispielhaft die Entwicklungskosten der implementierten Applikation iec104 ermittelt werden. Zum Ermitteln

der Kosten, muss der Stundensatz des Entwicklers mit der Zeit in Stunden multipliziert werden, die dafür notwendig ist, eine solche Applikation zu programmieren. Dazu gehört eine Designphase, die eigentliche Programmierung und eventuelle Fehlerkorrekturen. Weil die Kosten proportional zur Zeit sind, soll nur diese betrachtet werden. Es wird geschätzt, dass für die Entwicklung der Applikation iec104 etwa drei bis vier Mannwochen benötigt wurden. Die Schätzung wird dadurch erschwert, dass nicht kontinuierlich an der Entwicklung gearbeitet wurde. Die Arbeit daran wurde öfter durch andere notwendige Arbeiten unterbrochen.

Deswegen soll noch eine quantitative Bewertung anhand der Anzahl geschriebener Zeilen Quellcode LOC (Lines Of Code) stattfinden. In der Praxis ist solch eine Bewertung, also der Rückschluss von den LOC auf den Aufwand, allerdings problematisch. Die Zeit, die für die Designphase verwendet wird, wird nicht berücksichtigt. Eine überlegte und optimierte Programmierung benötigt tendenziell weniger LOC. Die Qualität der Programmierung wird auch nicht bewertet. Verschiedene Programmiersprachen und individuelle Programmierstile wirken sich ebenso auf die LOC aus.

Trotzdem gibt es Faustformeln, die einen Zusammenhang zwischen der Anzahl geschriebener Zeilen Quellcode LOC und der eingesetzten Entwicklungszeit sehen. Nach **Ludewig:SoftwareEngineerin** wird in einem Softwareprojekt nach n Stunden Aufwand – wobei hier der Gesamtaufwand gemeint ist, nicht die reine Programmierungszeit – ein System mit zwei mal n LOC erzeugt. Ein studentisches Projekt, das nicht kommerziell vertrieben werden soll, kann in der gleichen Zeit in etwa die dreifache Anzahl an LOC liefern. Dort können demnach mit jeder Stunde Aufwand in etwa sechs LOC entwickelt werden.

Um nach dieser Faustformel herauszufinden, wie viele Stunden in die Entwicklung der Contiki-Applikiation iec104 eingeflossen sind, werden die Anzahl an Zeilen Quellcode ermittelt. Die Contiki-Applikation iec104 besteht aus 14 Dateien:

- 1 Makefile
- 7 Header-Dateien
- 6 C-Dateien

Insgesamt enthalten diese Dateien 1011 Zeilen. Wenn die 211 Leerzeilen und 112 Kommentarzeilen abgezogen werden, bleiben 688 Zeilen Quellcode. Nach der Faustformel oben ergibt sich eine Entwicklungszeit von etwa 115 Stunden. Bei der Annahme von einem Stundensatz von 100€ ergeben sich Entwicklungskosten von 11.500€. Bei einer Wochenarbeitszeit von 40 Stunden, ergeben sich ungefähr 3 Mannwochen, die notwendig sind, um die Contiki-Applikation iec104 zu entwickeln. Das deckt sich in etwa mit den Erfahrungen aus dieser Arbeit.

7.2 Anwendungsmöglichkeiten

Dieses Kapitel stellt zuerst allgemein verschiedene Anwendungsbereiche von Smart Objects vor. Darauffolgend wird erörtert, in welchen Anwendungsbereichen die erstellte Lösung eingesetzt werden könnte. Anhand der Erfahrung, die bei der Implementierung gemacht wurden, werden mögliche Erweiterungen behandelt.

7.2.1 Anwendungsbereiche von Smart Objects

Smart Objects können in einer Vielzahl von Anwendungsbereichen eingesetzt werden, da sie sehr flexibel in ihrer Beschaffenheit sind. Die Programmierung kann individuell angepasst werden. Durch die Verwendung der richtigen Sensoren und Aktoren sind sie für verschiedenste Einsatzbereiche verwendbar. Nachfolgend sollen folgende fünf mögliche Anwendungsbereiche kurz beschrieben werden:

- eHome-Bereich
- Gebäudeautomation
- Industrieautomatisierung
- Logistik
- Smart Grid

Der eHome-Bereich ist ein Bereich in dem sich mehrere proprietäre Lösungen verbreitet haben, die oft nicht untereinander kompatibel sind. Dies kann auch ein Mitgrund dafür sein, dass sich die Heimautomatisierung bisher nicht so entwickelt hat wie vor etwa 7-9 Jahren prognostiziert [**vasseur10interconnecting**]. Wichtig ist auch, gerade im eHome-Bereich, eine einfache Installation der Geräte. Nur wenn ein Endnutzer ohne spezielles Expertenwissen Geräte in Betrieb nehmen und verwenden kann, wird sich eine Lösung durchsetzen können. Mögliche Einsatzgebiete von Smart Objects im eHome-Bereich sind zum Beispiel Steuerungen von Licht, der Heizung, von Fenster, von Rollläden und von Türschlössern.

Der eHome-Bereich wird für private Wohnhäuser eingesetzt. Im Gegensatz dazu zielt die Gebäudeautomation eher auf den professionellen Einsatz in großen Gebäuden meist im Zusammenspiel mit einem visualisierten Gebäudemanagementsystem. **Vermesan:TheInternetOfThings** beschreiben unter dem Titel „Smart IPv6 Building“ verschiedene Forschungsprojekte, die die Verwendung von IPv6 in der Gebäudeautomation untersuchen. Unter anderem wird auch das Hobnet-Projekt erwähnt, das spezielle Anwendungsfälle von Smart Objects in der Gebäudeautomation untersucht. Hier kommt auch 6LoWPAN zum Einsatz. Allgemein sind die Ziele einer Gebäudeautomation Energieeinsparungen, Sicherheit und Komfortgewinn.

Der Bereich Smart Grid wird einer der größten Anwendungsbereiche für Smart Objects werden [**Hersent:TheInternetOfThings**]. Seit mehreren Jahren geht der Trend in der

Stromerzeugung immer mehr in Richtung erneuerbaren Energiequellen wie Windkraftanlagen oder Photovoltaikanlagen. Diese sind aber im Gegensatz zu klassischen fossilen Kraftwerken dezentral aufgestellt. Das führt zu einem erheblichen Umbruch in der Stromnetzführung. Die Energie wird nicht allein an wenigen zentralen Örtlichkeiten durch große Kraftwerke, sondern auch dezentral durch kleine teils privat teils kommerziell geführten Energiequellen erzeugt. Das erschwert die Kraftwerksregelung und die Leitung des Lastflusses. Um trotzdem eine gute Netzstabilität zu gewährleisten, muss das Stromnetz intelligenter und vielfältiger überwacht und gesteuert werden.

Eine Maßnahme dabei ist das Smart Metering. Stromverbrauchszähler liefern über eine Kommunikationsschnittstelle den aktuellen Energieverbrauch an das Energieversorgungsunternehmen. Solche Art intelligente Zähler existieren schon länger für große Energieverbraucher wie Produktionsbetriebe, seit einigen Jahren werden Smart Meter auch für Privathaushalte angeboten.

7.2.2 Anwendungsmöglichkeiten für die Implementierung

Das Protokoll IEC104 ist ein Fernwirkprotokoll aus dem Bereich der Energieautomatisierung. Der Bereich Smart Grid ist also ein klassischer Anwendungsfall für dieses Protokoll. An ein zentrales Leitsystem werden Informationen übertragen und Steuerbefehle entgegen genommen. Allerdings wird der Zugriff auf eine Steuerung von einem elektrischen Verbraucher im Privathaushalt für ein Energieversorgungsunternehmen nicht von Bedeutung sein. Ein möglicher Anwendungsfall liegt eher in einem Smart Meter. Hier könnten mittels IEC104 Verbrauchsstände übertragen werden. Zusätzlich wäre in Absprache mit dem Privathaushalt eine Steuerung möglich. Genaue Anwendungsfälle müssen noch erprobt werden. Denkbar wäre eine Notabschaltung einer lokalen Photovoltaikanlage, wenn mehr Energie ins Stromnetz eingespeist wird als dort verbraucht wird bzw. in andere Stromnetzregionen abgeführt werden kann. Ebenfalls denkbar ist eine gezielte Steuerung von größeren Energieverbrauchern, bei denen der Einsatz zeitlich flexibel ist (Demand Side Management). Beispiele dafür sind die Waschmaschine oder die Batterie eines Elektrofahrzeugs. Bei einer großer Netzauslastung können diese steuerbaren Verbraucher zurückgefahren werden. Das Energieversorgungsunternehmen muss dann weniger Reserven für Spitzenlast vorhalten. So hilft eine intelligente Netzführung dabei, Kosten in der Energieerzeugung zu reduzieren.

7.2.3 Erweiterungsvorschläge für die Implementierung

Die erstellte Lösung ist ein Prototyp, der beispielhaft implementiert worden ist. Während der Arbeit sind mehrere Ideen entstanden, wie die Implementierung verbessert oder erweitert werden kann. Zuerst werden Vorschläge für die IEC104-Slave-Applikation aufgelistet.

- Informationsmeldungen mit Zeitstempel
  Die Statusänderung der Steckdose erfolgt aktuell über den Datentyp M_SP_NA_1

(Type Ident 1): Einzelbitmeldung ohne Zeitstempel. Eine Verbesserung wäre die Verwendung des Datentyps M_SP_TB_1 (Type Ident 30): Einzelbitmeldung mit dem Zeitstempel CP56Time2a. Hier wird zusätzlich ein Zeitstempel mit Jahr, Monat, Tag, Stunde, Minute, Sekunde und Millisekunde übertragen. So ist der exakte Zeitpunkt der Statusänderung bekannt. Voraussetzung dafür ist aber, dass die Steuerung mit der aktuellen Zeit synchronisiert ist. Über IEC104 existiert dafür eine Möglichkeit mit dem Datentyp C_CS_NA_1 (Type Ident 103): Zeitsynchronisationsbefehl. Ein Zeitstempel wird vom IEC104-Master zum IEC104-Slave gesendet und dort übernommen. Weil die Übertragungszeit dabei aber nicht berücksichtigt wird, ist die Verwendung vom IEC-Standard nur bedingt empfohlen. In der Praxis wird oft das Protokoll NTP (Network Time Protocol) verwendet. Eine Implementierung von NTP im Betriebssystem Contiki existiert bereits **Contiki-syslog**.

- interne Statusinformationen des Contiki Betriebssystems
  Das Modul iec104-para kann mit zusätzlichen Informationsobjekten erweitert werden. Zusätzliche interne Statusinformationen wie die Anzahl der laufenden Prozesse, TCP/IP-Verbindungen oder die Betriebszeit könnten übertragen werden. Für Messwerte gibt es bereits den Datentyp M_ME_NA_1 (Type Ident 9): Messwert, normalisiert und ohne Zeitstempel. Weitere Datentypen können implementiert werden.

Eine weitere Verbesserungsmöglichkeit betrifft nicht die Implementierung selbst sondern dem Testaufbau. Als 6LoWPAN-Router wurde ein handelsüblicher PC mit einer Linux-Installation verwendet. Als 6LoWPAN-Schnittstelle wurde ein USB-Stick RZUSBSTICK von Atmel verwendet, der auf dem PC eine Ethernet-Schnittstelle simuliert. Unter anderem hat das den Nachteil, dass auf dem PC der 6LoWPAN-Netzwerkverkehr nicht beobachtet werden kann, weil nur der simulierte Ethernet-Verkehr sichtbar ist. Es wird aber aktuell daran gearbeitet, 6LoWPAN direkt im Linux-Kernel zu implementieren [**Ott2012**]. Eine eingeschränkte Variante wird schon seit der Kernel Version 3.2.46 unterstützt und laufend erweitert. Als Funkchips werden der MRF24J40 von Microchip und der AT86RF230 von Atmel, der auch im Zigbit-Modul verbaut ist, unterstützt. **Ott2012** stellt eine Hardware vor bestehend aus einem BeagleBone und einem MRF24J40MA Funkchip. Damit kann 6LoWPAN nativ unter Linux verwendet werden ohne zusätzliche Hardware und ohne ein zusätzliches Betriebssystem. Es ist allerdings nicht bekannt, ob es Interoperabilitätsprobleme zwischen der 6LoWPAN-Implementierung im Linux-Kernel und im Contiki-Betriebssystem gibt.

7.3 Zusammenfassung

Es konnte gezeigt werden, dass eine internetfähige Steuerung mithilfe eines 8-bit Mikrocontrollers implementiert werden kann. Das Betriebssystem Contiki stellt dazu den notwendigen TCP/IP-Stack und andere Werkzeuge und Hilfsmittel zur Verfügung. Eine Webserver-Anwendung und eine Vielzahl anderer Anwendungen sind Teil des Betriebssystems. Mit der

Contiki-Applikation iec104 wurde gezeigt, dass die Implementierung des Protokolls IEC104 in einer limitierten Umgebung durchaus möglich ist. Auch die Verwendung von IEC104 im Zusammenspiel mit IPv6 hat funktioniert. Bisher war keine Implementierung von IEC104 über IPv6 bekannt. Generell zeigt die Contiki-Applikation iec104, dass die Implementierung neuer internetfähiger Anwendungen möglich und nicht aufwendiger als bei anderen Betriebssystemen ist. Die geringen Ressourcen müssen allerdings bei der Programmierung berücksichtigt werden.

Bei der Implementierung ist der 8KByte große RAM-Speicher die markanteste Ressourcengrenze. Die Auslastung beträgt 88,9%. Das Hinzufügen von zusätzlichen Contiki-Applikationen oder das Erweitern der Applikation iec104 ist deswegen nicht ohne Weiteres möglich. Durch den Einsatz von anderer Hardware mit einem größeren RAM-Speicher kann dieses Problem umgangen werden. So steht bei dem AVR Ravenboard mit 16KByte doppelt so viel RAM-Speicher zur Verfügung. Der Hersteller Redwire bietet mit dem Econotag ein fertiges Modul mit USB-Schnittstelle an. Verwendet wird der Freescale MC13224V, der einen 32-bit ARM7 Mikrocontroller mit einer IEEE-802.15.4-Schnittstelle kombiniert. Er verfügt über 128KByte Flash- und 96KByte RAM-Speicher. Das Betriebssystem Contiki unterstützt diese Modul über die Platform redbee-econotag. Beide, das AVR Ravenboard und das Econotag, sind aber für den Einbau in die verwendete Steckdose zu groß.

Als Kommunikationstechnologie wurde 6LoWPAN verwendet. Es scheint ideal für den Einsatz bei ressourcenbeschränkten Geräten zu sein. Anfang dieses Jahres hat die ZigBee Alliance die Spezifikation ZigBee IP veröffentlicht [**zigbee-ip**]. Damit werden unter der Verwendung von 6LoWPAN vermaschte drahtlose IPv6-Netzwerke unterstützt. Dies ist ein Beispiel dafür, dass die Verwendung von 6LoWPAN in vielen Bereichen voranschreitet. Eine weitere Entwicklung, die die Verbreitung von 6LoWPAN unterstützt, ist die Spezifizierung des Protokolls RPL [**RFC6550**]. RPL ist ein Routing-Protokoll, das für verlustbehaftete Sensornetze entwickelt worden ist. Die speziellen Anforderungen konnten von existierenden Routing-Protokolle wie OSPF oder RIP nicht erfüllt werden. Die Implementierung von RPL im Betriebssystem Contiki wird ContikiRPL genannt [**tsiftes10rpl**].

Sehr wichtig für die weitere Verbreitung von 6LoWPAN ist die Interoperabilität von verschiedenen Implementierungen. **ko11beyond** haben zwei unabhängige Implementierungen, Contiki und TinyOS, zusammen getestet. Die Interoperabilität zwischen beiden war gegeben, allerdings hatten kleine Unterschiede im jeweiligen Protokollstack einen Einfluss auf die Gesamtsystemleistung. Neben der Interoperabilität ist die Leistungsfähigkeit bei dem Zusammenspiel verschiedener Implementierungen von Bedeutung.

Als weitere Schwierigkeit kommt hinzu, dass viele Implementierungen den Funkempfänger so oft wie möglich ausschalten. Im Betriebssystem Contiki steht diese Funktionalität unter dem Namen ContikiMAC zur Verfügung. So kann der Energieverbrauch um bis zu 80% reduziert werden [**dunkels11contikimac**]. Das gesteuerte Ausschalten des Funkempfängers hat allerdings einen markanten Einfluss auf das Kommunikationsverhalten im Netzwerk. Weil keine Spezifikationen oder Standards für diese Funktionalität existieren, verhalten

sich Implementierungen sehr verschieden. **dunkels11adhoc** beschreiben, dass das Zusammenspiel von Kommunikationstechnologie und Ausschaltverhalten des Funkempfängers ein wichtiges Forschungsgebiet für das Internet-of-Things ist.

Anhang A: Der Blindtext

Weit hinten, hinter den Wortbergen, fern der Laender Vokalien und Konsonantien leben die Blindtexte. Abgeschieden wohnen Sie in Buchstabhausen an der Kueste des Semantik, eines grossen Sprachozeans. Ein kleines Baechlein namens Duden fliesst durch ihren Ort und versorgt sie mit den noetigen Regelialien. Es ist ein paradiesmatisches Land, in dem einem gebratene Satzteile in den Mund fliegen. Nicht einmal von der allmaechtigen Interpunktion werden die Blindtexte beherrscht - ein geradezu unorthographisches Leben.

Eines Tages aber beschloss eine kleine Zeile Blindtext, ihr Name war Lorem Ipsum, hinaus zu gehen in die weite Grammatik. Der grosse Oxmox riet ihr davon ab, da es dort wimmele von boesen Kommata, wilden Fragezeichen und hinterhaeltigen Semikoli, doch das Blindtextchen liess sich nicht beirren. Es packte seine sieben Versalien, schob sich sein Initial in den Guertel und machte sich auf den Weg. Als es die ersten Huegel des Kursivgebirges erklommen hatte, warf es einen letzten Blick zurueck auf die Skyline seiner Heimatstadt Buchstabhausen, die Headline von Alphabetdorf und die Subline seiner eigenen Strasse, der Zeilengasse. Wehmuetig lief ihm eine rethorische Frage ueber die Wange, dann setzte es seinen Weg fort. Unterwegs traf es eine Copy. Die Copy warnte das Blindtextchen, da, wo sie herkaeme waere sie zigmal umgeschrieben worden und alles, was von ihrem Ursprung noch uebrig waere, sei das Wort und und das Blindtextchen solle umkehren und wieder in sein eigenes, sicheres Land zurueckkehren. Doch alles Gutzureden konnte es nicht ueberzeugen und so dauerte es nicht lange, bis ihm ein paar heimtueckische Werbetexter auflauerten, es mit Longe und Parole betrunken machten und es dann in ihre Agentur schleppten, wo sie es fuer ihre Projekte wieder und wieder missbrauchten. Und wenn es nicht umgeschrieben wurde, dann benutzen Sie es immernoch.

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

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

[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 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] 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

[10] 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.

[11] 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

[12] 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.

[13] 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

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

[15] 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

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

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

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

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

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

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

[23] 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

[24] 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

[25] T. Høiland-Jørgensen 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] 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

[27] 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/

[28] 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

[29] 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

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

[31] 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/

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

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

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

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

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

[37] 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

[38] 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

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

[40] 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

[41] 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/

[42] 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

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

[44] P. Emmerich 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/

[45] 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

[46] *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

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

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

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