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 Science (B. Sc.)
im Studiengang
Wirtschaftsinformatik

Tag der Abgabe: 14.02.2025

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

Inhaltsverzeichnis

Kurzfassung

Das horizontale Scannen von Netzwerken oder Adressräumen ist eine fundamentale Methode der proaktiven Sicherheitsforschung. Etablierte Hochleistungsscanner wie ZMap oder Masscan basieren überwiegend auf C, was zwar maximale *Performance* ermöglicht, jedoch aufgrund fehlender Speichersicherheit Risiken für Sicherheitslücken birgt. Diese Bachelorarbeit untersucht, inwieweit ein in Rust implementierter Scanner hinsichtlich Durchsatz und Ressourceneffizienz mit diesen Tools konkurrieren kann und dabei durch die sprach-eigenen Garantien ein intrinsisch höheres Sicherheitsniveau bietet.

Hierfür wurde ein asynchroner `SYN`-Scanner („SYN-Rust“) entwickelt, der moderne Linux-Kernel-Schnittstellen wie `AF_XDP` und `eBPF` nutzt, um den Netzwerkstack partiell zu umgehen. Ergänzend kommt im *User-Space* eine logisch entkoppelte Architektur unter Verwendung der `tokio`-Laufzeitumgebung zum Einsatz, die eine effiziente Nebenläufigkeit gewährleistet. In einer kontrollierten Gigabit-Testumgebung wurde der Prototyp gegen ZMap und Masscan evaluiert.

Die Ergebnisse zeigen, dass die Rust-Implementierung im *Zero-Copy*-Modus die Bandbreitengrenze der Gigabit-Schnittstelle vollständig ausschöpft. Besonders hervorzuheben ist die Ressourceneffizienz: SYN-Rust verarbeitete pro CPU-Auslastungsprozent etwa dreimal mehr Pakete als Masscan und viermal mehr als ZMap. Die Arbeit belegt somit, dass Rust in Kombination mit modernen Kernel-Mechanismen eine leistungsfähige und sichere Alternative zu C für die Entwicklung systemnaher Netzwerkanwendungen im Bereich des Netzwerkscannings darstellt.

Abstract

Horizontal network scanning is a fundamental method in proactive security research. Established high-performance scanners such as ZMap or Masscan are predominantly based on C, which enables maximum performance but poses risks of security vulnerabilities due to a lack of memory safety. This thesis investigates the extent to which a scanner implemented in Rust can compete with these tools in terms of throughput and resource efficiency, while offering an intrinsically higher level of security through the language's inherent guarantees.

To this end, an asynchronous `SYN` scanner („SYN-Rust“) was developed that leverages modern Linux kernel interfaces such as `AF_XDP` and `eBPF` to partially bypass the network stack. Complementing this, the user-space employs a logically decoupled architecture utilizing the `tokio` runtime to ensure efficient concurrency. The prototype was evaluated against ZMap and Masscan in a controlled Gigabit test environment.

The results indicate that the Rust implementation in zero-copy mode fully saturates the bandwidth limit of the Gigabit interface. Particularly noteworthy is the resource efficiency:

SYN-Rust processed approximately three times more packets per percentage point of CPU utilization than Masscan and four times more than ZMap. The thesis thus demonstrates that Rust, combined with modern kernel mechanisms, represents a powerful and secure alternative to C for developing low-level network applications in the field of network scanning.

Kapitel 1: Einleitung

1.1 Motivation und Einführung in das Themengebiet

Das Scannen von Netzwerken oder gar dem gesamten Internet macht einen nicht zu vernachlässigenden Teil des Datenverkehrs im IPv4-Adressraum aus. So waren im Jahr 2024 weltweit 98 Prozent des unaufgeforderten TCP-Verkehrs auf `SYN`-Scans zurückzuführen[1]. Etablierte *Open-Source*-Projekte wie ZMap[2] oder Masscan[3] sind in der Lage, den gesamten IPv4-Adressraum innerhalb weniger Minuten zu scannen[4]. Durch das proaktive Scannen eigener Netzwerke können Schwachstellen identifiziert werden, bevor diese von Angreifern ausgenutzt werden. Darüber hinaus liefern breit angelegte Scans empirische Daten über globale Trends und Veränderungen in der Sicherheitslandschaft und stellen somit eine Datengrundlage für die Sicherheitsforschung dar. Angesichts der Tatsache, dass Cyberangriffe, wie beispielsweise *Denial-of-Service*-Attacken[5], sowohl die Reputation als auch die finanzielle Stabilität von Unternehmen massiv gefährden[6], ist die Verfügbarkeit und Weiterentwicklung leistungsfähiger Analysewerkzeuge von kritischer Bedeutung.

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

Rust hingegen schneidet in Vergleichen bezüglich der *Performance* auf ähnlichem Niveau wie C ab, bringt gleichzeitig aber das höchste Sicherheitsniveau der genannten Sprachen mit, indem es Speicherfehler weitestgehend verhindert[11], [13]. Außerdem unterstützt Rust Konzepte von Sprachen hoher Abstraktionsebene, wie beispielsweise die der funktionalen Programmierung oder Objektorientierung[13], wobei in der genannten Untersuchung zudem der Codeumfang geringer ausfiel als bei der untersuchten C-Variante.

Bisher fehlt eine fundierte Untersuchung darüber, ob Rust als moderne Sprache, welche Sicherheitsgarantien, *High-Level*[1] Konzepte und *Performance* vereint, in Kombination mit

[1] Auf hoher Abstraktionsebene

aktuellen Linux-Schnittstellen wie `AF_XDP` oder `eBPF` in der Lage ist, eine konkurrenzfähige Alternative zu gängigen Hochleistungsscannern, welche überwiegend in C geschrieben sind, darzustellen. Es ist ungeklärt, ob der potenzielle *Performance*-Unterschied gering genug ist, um durch die gewonnene Sicherheit kompensiert zu werden, weshalb diese Arbeit an diesem Punkt ansetzt.

1.2 Zielsetzung und Forschungsfrage

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

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

1.3 Abgrenzung des Themas

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

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

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

Kapitel 2: Theoretische Grundlagen

In diesem Kapitel werden die nötigen Grundlagen zum Verständnis des Portscannings 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, um das Verständnis für das nachfolgende Konzept der Implementierung zu schaffen.

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

2.1 Grundlagen der Netzwerkkommunikation

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

2.1.1 Ports

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

- **Offen:** Eine Anwendung lauscht auf dem Port und akzeptiert eingehende gültige TCP oder UDP Anfragen [15].
- **Geschlossen / Gefiltert:** Der mit dem Port verbundene Dienst ist ansprechbar, akzeptiert jedoch keine eingehenden Verbindungen. Oft wird eine ICMP (Fehler) Antwort zurückgegeben oder es erfolgt keine Antwort, da beispielsweise kein Service für diesen Port existiert [15].

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

2.1.2 Transmission Control Protocol (TCP)

Das *Transmission Control Protocol* operiert in der Transportschicht des TCP/IP-Modells und ist eines der meistgenutzten Transportprotokolle des Internets [20, S. 71]. Es gewährleistet eine zuverlässige, verbindungsorientierte Datenübertragung zwischen den Prozessen der *Hosts*. Die ursprüngliche Spezifikation erfolgte im RFC 793 [21], welches durch RFC 9293 [22] konsolidiert wurde. Für die Entwicklung eines `SYN`-Scanners sind insbesondere der Aufbau des TCP-Headers (Abb. 2.1) und der Mechanismus zum Verbindungsaufbau (Abb. 2.2) entscheidend.

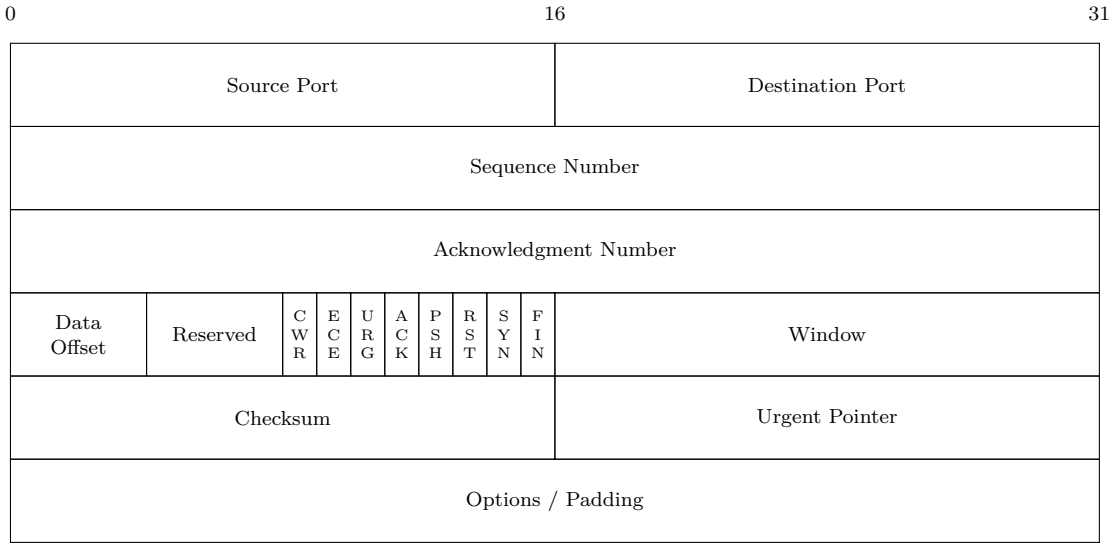

Abbildung 2.1: Aufbau des TCP-Headers nach RFC 9293 [22].

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

2.2 Portscanning

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

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

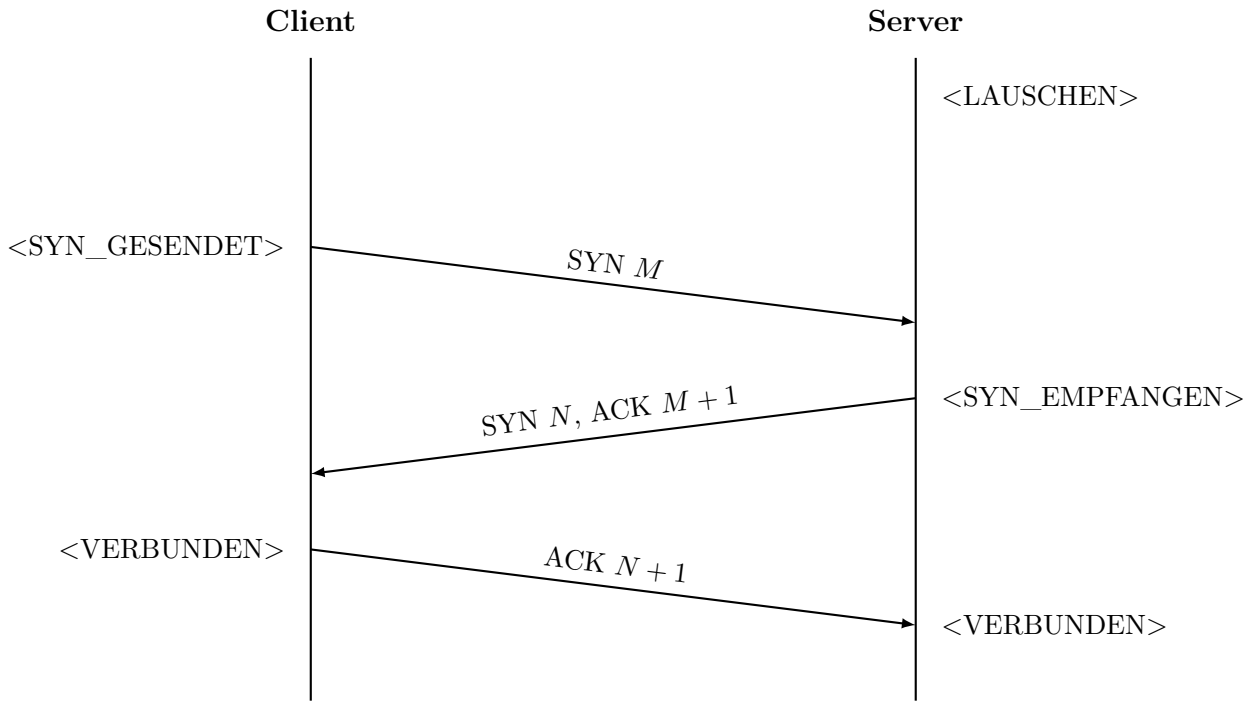

Abbildung 2.2: *Three-Way-Handshake* zum Aufbau einer TCP-Verbindung [20].

- **Vertikales Scannen:** Hierbei wird ein einzelner Ziel-*Host*[2] auf eine Vielzahl von Ports (oft alle 65535) gescannt, um ein möglichst vollständiges Profil möglicher Schwachstellen des Zielsystems zu erlangen, weshalb es sich besonders für das *Penetration Testing* eignet.
- **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 sammeln [4].

2.2.1 SYN-Scanning

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

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

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

ziert wurde [24, S. 8]. Dafür werden verbindungsspezifische Informationen unter Verwendung eines *Hash*-Algorithmus (z. B. *Keyed SipHash* [25]) kodiert und als *Sequence Number* in den TCP-Header des ausgehenden `SYN`-Pakets eingetragen. Antwortet ein Ziel-*Host* mit einem `SYN-ACK`-Paket, so enthält dessen *Acknowledgment Number* gemäß TCP-Spezifikation den inkrementierten Wert der ursprünglichen *Sequence Number*. Die Validierung lässt sich abstrahiert wie folgt beschreiben:

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

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

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

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

Tabelle 2.1: Relevante TCP-Header Felder

2.3 Schnittstellen zur Paketverarbeitung unter Linux

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

2.3.1 Linux

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

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

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

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

Die Kommunikation zwischen diesen Ebenen erfolgt über *System Calls* [19, S. 44]. Jeder Wechsel (*Context Switch*) zwischen *User-* und *Kernel-Space* sowie das Kopieren von Daten zwischen diesen Speicherbereichen erzeugt *Overhead*. Beim Versenden und Empfangen sehr vieler Pakete summiert sich dieser *Overhead*, da jedes Paket im Normalfall sowohl *Kernel-Space* als auch *User-Space* durchschreitet. Dies belastet die CPU und wird für den Durchsatz zum Flaschenhals [26].

2.3.2 *Raw-Sockets* und Adressfamilien

Als Endpunkt für die Kommunikation werden *Sockets* genutzt [27]. Die traditionelle Netzwerkprogrammierung unter Linux abstrahiert die Komplexität der Netzwerkprotokolle wie TCP. So übernimmt der Kernel dabei vollständig den *Three-Way-Handshake* und die Zustandsverwaltung [19, S. 1158]. Für einen `SYN`-Scanner ist dies ungeeignet, da der Scanner lediglich das initiale `SYN`-Paket senden und die Antwort registrieren will, ohne eine vollwertige Verbindung aufzubauen, welche Ressourcen im Kernel binden würde.

Raw-Sockets erlauben der Anwendung, Netzwerkpakete unter Umgehung bestimmter *Layer* des Kernel-Stacks zu senden und zu empfangen [28]. Der Entwickler muss die Protokoll-Header selbst konstruieren. Dies ist für *Half-Open* Portscanner essenziell, um individuelle Pakete zu generieren, ohne dass der Kernel automatisch in den Verbindungsaufbau eingreift.

Die Adressfamilien definieren dabei die Interpretation der Adressen und die Ebene des Zugriffs [29]. Der Linux-Kernel stellt diverse Adressfamilien bereit. Zum Verständnis, im Rahmen dieses Projektes, sind folgende Varianten von zentraler Bedeutung:

- **`AF_INET` (Netzwerk-Ebene):** Diese Familie operiert auf Layer 3 der IP-Ebene [28]. Bei Nutzung von *Raw-Sockets* fügt der Kernel standardmäßig den IP-Header hinzu und übernimmt das vollständige Routing zur korrekten Netzwerkschnittstelle [19, S. 1202].

- **`AF_PACKET` (Sicherungsschicht):** Diese Familie ermöglicht direkten Zugriff auf Layer 2 (Ethernet-Ebene). Anwendungen erzeugen vollständige *Ethernet-Frames* und haben somit die volle Kontrolle. Das Versenden oder Empfangen von Paketen erfordert jedoch weiterhin die Allokation von Kernel-internen Datenstrukturen [30].

- **`AF_XDP` (Hochperformant):** Hierbei handelt es sich um eine speziell für Hochleistungsanwendungen optimierte Adressfamilie. Sie ermöglicht das Senden und Empfangen von Paketen unter Umgehung des regulären Kernel-Netzwerkstacks. Dabei ist zwischen dem universell verfügbaren *Copy-Mode*, in welchem Daten zwischen Kernel und User-Space kopiert werden, und dem Treiber-abhängigen *Zero-Copy-Mode*, in welchem Daten direkt in den Speicher der Anwendung geschrieben werden, zu unterscheiden [26].

2.3.3 Erweiterte Berkeley Packet Filter (`eBPF`)

Ursprünglich als *Berkeley Packet Filter* (`BPF`) für Werkzeuge wie `tcpdump` entwickelt, um Pakete effizient zu filtern [31], wurde die Technologie erweitert, sodass grundlegend neue Möglichkeiten außerhalb des reinen Filterns von Paketen erschlossen wurden.

`eBPF` ist eine im Linux-Kernel integrierte virtuelle Maschine (VM), die es erlaubt, benutzerdefinierten *Bytecode* sicher und effizient im *Kernel*-Kontext (siehe Abb. 2.3) auszuführen, ohne Kernel-Module schreiben oder den Kernel neu kompilieren zu müssen [32]. `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 [33]. So wird undefiniertes Verhalten durch Fehler wie beispielsweise Endlosschleifen oder falsche Speicherzugriffe strikt vermieden.

Da eBPF-Programme ereignisbasiert ausgeführt werden und keinen eigenen persistenten Speicher besitzen, werden sogenannte `bpf`-Maps verwendet, um Zustände zu bewahren und Daten auszutauschen [26]. Dies sind generische Schlüssel-Wert-Speicher, die sowohl von verschiedenen eBPF-Programmen als auch vom *User-Space* gelesen und beschrieben werden können [26]. Dabei gibt es verschiedene Datenstrukturen. Eine davon ist der `RingBuf` (`BPF_MAP_TYPE_RINGBUF`). Hierbei handelt es sich um einen für den Datenaustausch vom Kernel zum *User-Space* optimierten Ringpuffer, der im Vergleich zu älteren Methoden wie *Perf Buffer* durch geteilte Speicherregionen effizienter arbeitet und die Reihenfolge der Ereignisse garantiert [34].

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. [26]

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) [26][33].

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 speicheraufwendige *Context Switches* zwischen *Kernel-* und *User-Space*, sowie die Allokation komplexer Metadatenstrukturen (eines sk_buff [3]) [26][35]. XDP greift vor dieser Allokation ein (siehe Abb. 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 [26].

Die *Performance* und Verfügbarkeit von XDP hängen vom verwendeten Betriebsmodus ab. Nach Zhang et al. [35] und Vieira et al. [33] lassen sich drei Modi unterscheiden:

- ***Native Mode* (*Driver Mode*):** Dies ist der Standardmodus für Hochleistungsanwendungen. Das XDP-Programm wird direkt im Netzwerkkartentreiber ausgeführt. Die Verarbeitung erfolgt nach dem *DMA*-Transfer (*Direct Memory Access*) in den *Ring-Buffer*, aber vor der sk_buff-Allokation. Dies erfordert explizite Unterstützung durch den Treiber der Netzwerkkarte.
- ***Offloaded Mode* (*Hardware Mode*):** Hierbei wird das eBPF-Programm vom Kernel auf die Netzwerkkarte ausgelagert und direkt auf der Hardware ausgeführt. Dies bietet die höchste *Performance*, da die Host-CPU vollständig von der Paketverarbeitung entlastet wird, setzt aber die Nutzung einer sogenannten *SmartNIC* voraus.
- ***Generic Mode* (*SKB Mode*):** Dieser Modus dient der Kompatibilität. Wenn ein Treiber XDP nicht nativ unterstützt, führt der Kernel das XDP-Programm im Netzwerkstack des Kernels aus. Zwar gehen hier die massiven *Performance*-Vorteile der Speicherersparnis verloren, jedoch wird sichergestellt, dass XDP-Anwendungen auf jeder Hardware funktionsfähig bleiben.

2.4 Die Programmiersprache: Rust

Rust ist eine multiparadigmatische Systemprogrammiersprache, die ursprünglich von Mozilla Research entwickelt wurde. Das Hauptaugenmerk der Sprache liegt auf der Sicherheit, wobei auch *Performance* und Nebenläufigkeit zunehmend an Bedeutung gewinnen [36]. Rust vereint dabei als erste Sprache Speichersicherheits-Konzepte höherer Abstraktionsebenen mit der direkten Ressourcenkontrolle systemnaher Sprachen [37].

[3] *Socket Buffer*

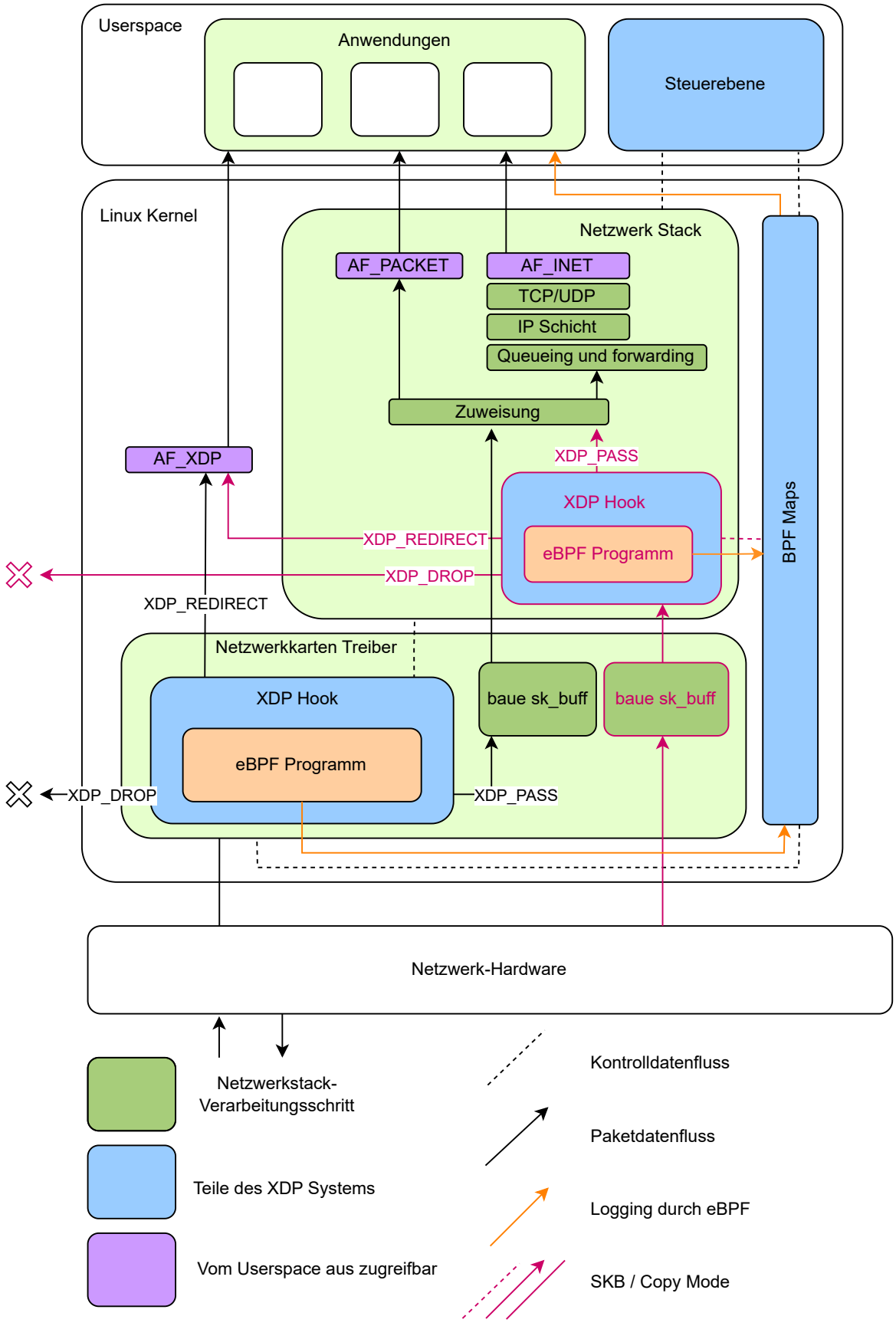

Abbildung 2.3: Der Empfangspfad durch den Kernel bei der Nutzung von `XDP` und `eBPF` (vereinfacht). Orientiert an Høiland et al. [26].

2.4.1 Konzepte und Besonderheiten

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

- ***Ownership*:** Jede Variable hat einen *Owner* (Besitzer). Wird der Besitzer gelöscht, wird auch die Variable gelöscht. Die Variable kann nur einen Besitzer haben [38]. 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 [36].

- ***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 [36].

Die Einhaltung dieser Regeln wird zur Kompilierzeit vom *Borrow-Checker* verifiziert. Außerdem hat Rust noch weitere Konzepte zur Steigerung der Sicherheit. Budgen et al. [36] führen weitere Konzepte auf, 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 zurückgeben, die entweder den gewünschten Wert x als `Some(x)` oder `None` enthält. Außerdem muss *Pointer*-Arithmetik in sogenannte `unsafe`-Blöcke ausgelagert werden. Diese dienen als Möglichkeit zur Umgehung der anderen Konzepte. Sie lösen im Fehlerfall eine Programm-beendenden `panic` aus, welche verhindert, dass das Programm in einem undefinierten Zustand weiter läuft.

Trotz der Möglichkeit und Notwendigkeit, `unsafe`-Blöcke zu nutzen (beispielsweise für hardwarenahe Operationen oder der Arbeit mit C-Bibliotheken), was dem Sicherheitskonzept der Sprache widerspricht, sind laut Jung et al. „zahlreiche wichtige Rust Bibliotheken“ [38] sicher, da sie die `unsafe`-Blöcke korrekt kapseln [38].

Durch das Zusammenspiel dieser Konzepte können Speicherfehler verschiedenster Art bereits zur Kompilierzeit vermieden werden, was Rust zur sichersten unter den derzeit gängigen Sprachen macht [11]. Allerdings müssen deshalb auch einige Regeln bei der Programmierung beachtet und eingehalten werden, weshalb der Sprache eine steile Lernkurve zugeschrieben wird [39].

2.4.2 Asynchrone Programmierung und *Performance* von Rust

Die im letzten Abschnitt genannten Konzepte schließen auch *Data Races*, welche beim Zugriff mehrerer *Threads* [4] auf den gleichen Speicher entstehen können, bereits zur Kompilierzeit aus [13]. Dies macht *Data Races* zu einer häufigen Fehlerquelle in der asynchronen Programmierung [36]. Die Beseitigung dieser Fehlerquellen 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[5] für andere Prozesse freigegeben werden. Dadurch kann die Effizienz und die Responsivität des Systems erhöht werden [40] [41].

Die Parallelität hingegen bezieht sich auf die tatsächliche physikalische Ausführung mehrerer Aufgaben zum gleichen Zeitpunkt [40], was eine entsprechende *Multi-Core*-Hardware voraussetzt [42]. Der Vorteil der Parallelität liegt in der Leistungssteigerung und der Maximierung des Datendurchsatzes bei rechenintensiven Problemen [40][41].

Rusts Konzepte zur *Performance*-Steigerung

Neben den möglichen *Performance*-Vorteilen durch die nebenläufige Programmierung, welche aber letztendlich dem Programmierer überlassen ist, bietet die Sprache ihre größten internen *Performance*-Vorteile durch die *Zero-Cost Abstraction*. Das Konzept der *Zero-Cost Abstraction* [39], 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“ [43].

Dazugehörige Konzepte sind beispielsweise die Eliminierung von Laufzeit-*Overhead* durch die Vermeidung eines zur Laufzeit arbeitenden *Garbage Collectors* [11] [39], die *Monomorphisierung* um die Typen oder Größen generischer Strukturen wie z.B. `Vec` [6] oder `Option` nicht mehr während der Laufzeit bestimmen zu müssen [39], oder die Bereitstellung eigener Iteratoren, welche die Leistung manuell geschriebener Schleifen oft übertrifft [39].

Diese Konzepte und vor allem die Prüfung der in Abschnitt 2.4.1 vorgestellten Konzepte zur Kompilierzeit, führt dazu, dass Rust in Benchmarks gängige Sprachen wie Java, Python, oder Go übertrifft und sogar mit der Geschwindigkeit von C konkurriert [13] [36] [39].

[4] Untergeordnete Arbeitseinheiten eines Prozesses
[5] Ein-/Ausgabe-Operationen
[6] Vektor, ähnlich einer Liste

Kapitel 3: Stand der Technik

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

3.1 Historische Entwicklung des horizontalen Netzwerkscannings

Ursprüngliche Netzwerkscanner wie Nmap [15] wurden primär für die vertikale Analyse einzelner *Hosts* oder kleiner Netzwerke konzipiert. Sie arbeiten teils zustandsbehaftet, 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 **Durumeric_Wustrow_Halderman**.

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 soweit gesteigert werden, dass 97 % der theoretischen Geschwindigkeit von Gigabit-Ethernet erreicht wurden. Dies ermöglichte erstmals Scans des gesamten IPv4-Adressraums in unter 45 Minuten von einem einzelnen Rechner aus **Durumeric_Wustrow_Halderman**. Spätere Arbeiten, wie Zippier ZMap, optimierten diesen Ansatz weiter, um auch bis zu 10-Gbps-Leitungen auszulasten **Adrian_Durumeric_Singh_Hal** und somit die anhaltende Relevanz des ZMap-Projektes zu unterstreichen.

3.1.1 Der Standard Scanner: ZMap

In der wissenschaftlichen Literatur gilt ZMap **Durumeric_Wustrow_Halderman** 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 **Adrian_Durumeric_Singh_Halderman**, dass durch spezialisierte Treiber wie `PF_RING ZC` (*Zero-Copy*) noch höhere Geschwindigkeiten möglich sind, jedoch weisen die Autoren darauf hin, dass solche externen Treiber oft Wartungsprobleme und Inkompatibilitäten mit sich bringen. Daher setzt die aktuelle Version von ZMap primär auf universell verfügbare Linux-Schnittstellen, auch wenn diese *Performance*-technisch limitiert sind [2].

- **Zustandslose Architektur:** ZMap nutzt das Prinzip der `SYN`-Cookies, um keinen Zustand für ausgehende Verbindungen im Arbeitsspeicher halten zu müssen.

- **Adressgenerierung mittels zyklischer Gruppen:** ZMap nutzt zyklische multiplikative Gruppen modulo p (wobei p eine Primzahl $> 2^{32}$ ist). Dies ermöglicht eine pseudozufällige Permutation des gesamten IPv4-Adressraums, was nötig ist, um Zielnetzwerke nicht zu überlasten. **Durumeric_Wustrow_Halderman**.

3.2 Alternative Implementierungsansätze

Neben der reinen *Socket*-Programmierung und klassischen *Raw-Sockets* haben sich weitere, teils modernere Techniken 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 [44]. Der Nachteil ist jedoch die hohe Komplexität, die exklusive Belegung von CPU-Kernen und die schwierige Integration in bestehende Systemumgebungen [26].

- **Eigener TCP-Stack im *User-Space* (Masscan):** Der Scanner Masscan [3] umgeht den Flaschenhals des Betriebssystems mithilfe eines eigenen TCP-Stacks im *User-Space*. Dies erlaubt es dem Scanner, die Statusverwaltung und das Timing von Paketen komplett unabhängig vom Kernel-*Scheduler* zu steuern. Der *Scheduler* ist normalerweise dafür verantwortlich, die Rechenzeit der CPU auf die laufenden Prozesse

zu verteilen [19, S. 737]. Die dabei entstehenden *Context Switches* können das präzise Timing von Hochleistungsanwendungen stören. Dadurch kann Masscan deutliche *Performance*-Gewinne gegenüber ZMap erreichen [45].

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

3.3 Weitere relevante wissenschaftliche Arbeiten

Der Forschungsstand zu horizontalen Hochleistungs-Netzwerkscannern wurde bereits in den vorherigen Abschnitten Abschnitte 3.1 und 3.2 aufgegriffen. Ergänzend dazu ist noch anzubringen, dass nach bestehenden Ansätzen zur Kernel-Umgehung die Arbeit zu `XDP` (*eXpress Data Path*) einen Paradigmenwechsel darstellt. Zuvor genutzte Umgehungsstrategien waren unter anderem Netmap [46], welches bereits 2012 Konzepte wie *Shared Memory*[1] und *Zero-Copy* einführte, aber den Netzwerkstack eher ersetzte, statt ihn zu komplementieren, oder `DPDK` (siehe Abschnitt 3.2). Høiland-Jørgensen et al. zeigen, dass mit `XDP` durch eine programmierbare Paketverarbeitung im Kernel-Treiber eine mit `DPDK` vergleichbare *Performance* erreicht werden kann, ohne die Integration in das Betriebssystem aufzugeben [26].

Zwei Studien vergleichen `SYN`-Scanner [45] [47], fokussieren sich aber eher auf die Trefferrate, welche in der Arbeit nicht priorisiert wird (siehe Abschnitt 1.3). Außerdem ist die Gestaltung der Testumgebung bei beiden Arbeiten nicht auf Hochleistungs-Szenarien ausgelegt.

Die Eignung von Rust für hochperformante Netzwerkprogrammierung wird in mehreren wissenschaftlichen Arbeiten evaluiert. Sagramoni et al. schrieben eine Netzwerkbibliothek in Rust und verglichen sie mit der ursprünglichen C-Bibliothek [48]. Gonzalez et al. entwickelten einen UDP Treiber für Linux und verglichen diesen mit einem ähnlichen C-Treiber [49]. Moon et al. erstellten einen *NAT* (Network Address Translator) und testeten den Durchsatz [50]. 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 [51].

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

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 [36] [11] [52].

3.4 Vergleichsobjekte für die Evaluation

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

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

3.5 Nachteile bisheriger Ansätze

3.5.1 Nachteile C-basierter Ansätze

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

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

Darüber hinaus geht die Leistungsfähigkeit von C oft zu Lasten der Wartbarkeit und Entwicklungseffizienz [13]. Um maximale Durchsatzraten zu erzielen, sind in C häufig komplexe, manuelle Optimierungen notwendig. Costanzo et al. heben hervor, dass die Entwicklung von korrektem und effizientem C-Code im Vergleich zu Rust-Code einen signifikant höheren Programmieraufwand erfordert, insbesondere wenn komplexe Nebenläufigkeit umgesetzt

werden soll [13]. Selbst die Autoren von ZMap sagen in ihrer Retrospektive explizit, dass sie für eine heutige Implementierung ihres Scanners Rust wählen würden, um die Wartbarkeit und Sicherheit der Codebasis langfristig zu gewährleisten [2].

Ein weiterer wesentlicher Nachteil bisheriger Hochleistungsansätze (wie `DPDK` oder `PF_RING`) ist ihre fehlende Integration in den Linux-*Mainline-Kernel*. Sie erfordern oft proprietäre Treiber oder Kernel-Module, die das Sicherheitssystem des Kernels umgehen und bei Updates zu Inkompatibilitäten führen können [2]. Durumeric et al. merken an, dass die Einführung und Wartung von `PF_RING` für ZMap über die Jahre eine erhebliche Hürde darstellte [2]. `XDP` füllt diese Lücke, indem es *High-Performance*-Paketverarbeitung direkt im Kernel ermöglicht, ohne dessen Sicherheit und Kompatibilität zu kompromittieren.

3.5.2 Lösungsansatz

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

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

Kapitel 4: Anforderungsanalyse und Methodik

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

4.1 Anforderungsanalyse

4.1.1 Funktionale Anforderungen

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

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

- **/F-06/ Schließen der Verbindung:** Nach der Identifikation eines offenen Ports muss der Scanner ein `RST`-Paket senden, um die halboffene Verbindung auf dem Zielsystem korrekt zu beenden.
- **/F-07/ Endausgabe:** Es muss eine Endausgabe in einer Datei oder dem *Standard Output* geben, in welcher die ausgewerteten Scanergebnisse - bestehend aus IP-Adresse und Ziel-Port der offenen Zielsysteme - enthalten sind.
- **/F-08/ Durchsatzlimitierung:** Das Programm muss in der Lage sein, eine angegebene Durchsatzrate (in Byte pro Sekunde) bezüglich der gesendeten `SYN`-Pakete um nicht mehr als 3 % zu über- oder unterschreiten. Die Anzahl der insgesamt versendeten Pakete darf sich dabei nicht verändern.
- **/F-09/ Eingabeschnittstelle:** Das Programm muss die zu scannenden Ziel-IP-Adressen aus dem *Standard Input* des Programmes entnehmen, um sich in die Infrastruktur des Unternehmens, welches diese Arbeit begleitet, zu integrieren.

4.1.2 Nicht-funktionale Anforderungen

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

- **/NF-01/ Maximierung des Durchsatzes:** Das System soll in der Lage sein, die verfügbare Bandbreite einer Standard-Gigabit-Schnittstelle vollständig auszunutzen.
- **/NF-02/ Asynchrone Architektur:** Die Implementierung muss auf einem asynchronen Programmiermodell basieren, um durch nicht-blockierende *I/O*-Operationen eine hohe Nebenläufigkeit zu gewährleisten.
- **/NF-03/ Nutzung moderner Kernel-Mechanismen:** Zur Evaluation der Forschungsfrage müssen Linux-native Schnittstellen zur hochperformanten Paketverarbeitung wie `AF_XDP` oder `eBPF` verwendet werden.
- **/NF-04/ Speichersicherheit:** Die Implementierung soll die Sicherheitsgarantien von Rust wahren. `unsafe`-Blöcke können genutzt werden, wenn sie für die Erfüllung von funktionalen oder nicht-funktionalen Anforderungen von großer Bedeutung sind. Sie sollten aber möglichst vermieden oder durch die Nutzung anderer Sicherheitsmechanismen ergänzt werden.
- **/NF-05/ Minimale Ressourcennutzung:** Der CPU- und Arbeitsspeicherverbrauch soll im Verhältnis zum erzielten Durchsatz minimiert werden.
- **/NF-06/ Technologische Einschränkung:** Das Programm darf ausschließlich Techniken verwenden, die im Linux-Kernel-Ökosystem verfügbar sind, um Abhängigkeiten von Drittanbieter-Treibern zu vermeiden.

4.2 Untersuchungsdesign

In diesem Abschnitt wird das methodische Vorgehen zur Validierung der Anforderungen beschrieben. Die Verifikation der in Abschnitt 4.1 definierten Anforderungen erfolgt anhand von zwei Methoden:

- **Dynamische Tests:** Diese validieren das Laufzeitverhalten und die Performanz des Systems. Anforderungen wie das korrekte Senden und Empfangen von Paketen (Abschnitt 4.1.1) oder die Einhaltung eines Durchsatzlimits (Abschnitt 4.1.1) werden durch explizite Testfälle (*Proof of Concept*) und Evaluationsszenarien nachgewiesen.
- **Statische Inspektion:** Anforderungen, die sich auf die Architektur, die Wahl der Programmiersprache oder die Verwendung spezifischer Kernel-Schnittstellen beziehen, werden durch die Inspektion der Implementierung verifiziert. Der Nachweis für die asynchrone Architektur (Abschnitt 4.1.2), die Nutzung von `AF_XDP` und `eBPF` (Abschnitt 4.1.2), die Speichersicherheit durch Rust (Abschnitt 4.1.2), die technologische Einschränkung (Abschnitt 4.1.2) oder der generelle Aufbau (Abschnitt 4.1.1) gilt als erbracht, indem die entsprechenden Konzepte im Design verankert und im Quellcode umgesetzt wurden (siehe Kapitel 5).

Anschließend wird erklärt, wie *Performance* im Kontext eines `SYN`-Scanners zu definieren ist und zuletzt werden die, in dieser Arbeit zur Evaluation genutzten, Metriken und Evaluationsszenarien festgelegt.

4.2.1 Evaluationstests für den *Proof of Concept*

Um die Funktionsweise, beziehungsweise den Scanner nach Abschnitt 4.1 prüfen zu können, werden zwei Tests durchgeführt:

1. **/T-01/ Sende- und Empfangsvalidierung:** Der erste Test dient als Validierung der Anforderungen Abschnitt 4.1.1. Es werden Pakete verschickt und von einer anderen, antwortenden Instanz empfangen. Dabei wird untersucht, ob die korrekte Anzahl an `SYN`-Paketen verschickt, `SYN-ACK`-Paketen empfangen und `RST`-Antworten verschickt wird[1]. Zur Validierung werden die Werte der Ausgaben des Scanner-Knotens mit den empfangenen Paketen des Ziel-Knotens und den erwarteten Werten verglichen.
2. **/T-02/ Paketvalidierung:** Im zweiten Test wird die Korrektheit der erstellten Pakete validiert, um die Umsetzung der Anforderungen Abschnitt 4.1.1 zu überprüfen. Dafür werden Pakete an eine antwortende Instanz verschickt. Diese antwortet nur auf korrekte Pakete und stoppt das Senden von Antworten, wenn gültige `RST`-Pakete eingehen. In dem Test wird somit untersucht, ob diese Verhaltensweisen auftreten. Zusätzlich wird mithilfe von externen Tools die Validität der Pakete geprüft.

[1] Mit `SYN` / `SYN-ACK` / `RST` ist der gesetzte Wert der TCP-*Flags* (siehe Abschnitt 2.2.1) gemeint.

Die konkrete Umsetzung und genaue Spezifizierung der Tools wird in Kapitel 6 beschrieben.

4.2.2 Evaluationsszenarien

Um die in Abschnitt 4.1 definierten Anforderungen zu validieren, werden zwei zu untersuchende Szenarien definiert:

1. **/S-01/ Ermittlung der Performanzgrenzen:** In diesem Szenario wird jegliche künstliche Drosselung aufgehoben. Das Ziel ist es, die maximalen Durchsatzraten zu ermitteln. Hierbei wird geprüft, wie effizient die Ressourcen unter Volllast genutzt werden, um die nicht-funktionalen Anforderungen Abschnitt 4.1.2 zu untersuchen.

2. **/S-02/ Simulation unter realen Parametern und *Features*:** Um die Vergleichbarkeit zu praxisrelevanten Szenarien zu erhöhen, werden Parameter gewählt, die für echte Internetscans typisch sind. Dies testet die *Performance*-Effizienz unter möglichst realen Bedingungen und untersucht die funktionale Anforderung Abschnitt 4.1.1. Dabei wird auch ein Augenmerk auf die Nutzung von *Features* gelegt, die im Kontext eines realen Scans von Nutzen sind, beispielsweise zur Verschleierung des Scans.

4.2.3 Metriken

Der Begriff „*Performance*-Effizienz“ wird gemäß der Norm *ISO/IEC 25010* als die Fähigkeit eines Produkts, seine Funktionen innerhalb festgelegter Zeit- und Durchsatz-Parameter zu erfüllen und dabei die Ressourcen unter den gegebenen Bedingungen effizient zu nutzen, verstanden [53].

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

- **/M-01/ Paketrate:** Die Paketrate wird in Pakete pro Sekunde *PPS* dargestellt und beschreibt die durchschnittliche Anzahl der erfolgreich an den Netzwerkadapter übergebenen Pakete pro Sekunde. Da die Scanner nur sehr kleine Pakete verschicken, ist die Paketrate in *Performance*-orientierten Projekten dieser Art der limitierende Faktor **Durumeric_Wustrow_Halderman**, weshalb sie maßgeblich als Metrik für die *Performance* dient.

- **/M-02/ CPU-Auslastung:** Die CPU-Auslastung von hochperformanten Netzwerkanwendungen findet maßgeblich in drei Bereichen statt: im *User-Space* (Ausführung der

Anwendung), im *Kernel-Space* (Systemaufrufe) und in der Verarbeitung von *SoftIRQs* [54, S. 5, 134, 137]. *SoftIRQs*[2] stellen hierbei einen wesentlichen Faktor für die Netzwerklast dar. Eine performante Messung muss daher die prozentuale Auslastung der CPU-Kerne in Bezug auf alle drei Metriken erfassen.

- **/M-03/ RAM-Verbrauch:** Der RAM-Verbrauch als zweiter primärer Teil der Ressourcenmetriken wird in Megabyte (MB) angegeben und stellt den Anteil des physisch durch den Scanner belegten Arbeitsspeichers dar.

[2] *SoftIRQs* dienen dazu, rechenintensive Aufgaben, die durch Hardware-Unterbrechungen ausgelöst wurden (wie den Empfang von Netzwerkpaketen), zeitlich verzögert abzuarbeiten. Dies verhindert, dass die CPU durch neue Hardware-Signale zu lange blockiert wird [54, S. 134].

Kapitel 5: Konzeption und Implementierung

In diesem Kapitel wird zuerst das Konzept zur Erfüllung der Anforderungen vorgestellt. Anschließend wird die konkrete Umsetzung der Komponenten detailliert dargelegt und erklärt. Die folgende Beschreibung der Architektur und Implementierung dient gemäß Kapitel 4 zugleich als Nachweis für die Erfüllung der Anforderungen, die durch statische Inspektion verifiziert werden.

Der gesamte Quellcode ist auf GitHub bereitgestellt [55].

5.1 Konzeptioneller Lösungsansatz

Zur Verdeutlichung des Softwareentwurfs werden in diesem Abschnitt zunächst die logischen Komponenten der Anwendung und deren Zusammenspiel vorgestellt. Im Anschluss daran werden die spezifischen Maßnahmen erläutert, die zur Steigerung der *Performance* und Effizienz ergriffen wurden.

5.1.1 Logische Komponenten des Scanners

Abbildung 5.1 visualisiert die logischen Komponenten des Scanners sowie deren Interaktionen. Um die in Abschnitt 4.1.2 geforderte Asynchronität sowie die Entkopplung von Sende- und Empfangsprozessen zu realisieren, folgt die Architektur einem *Pipeline*-basierten Ansatz. Dabei werden die einzelnen Module lose gekoppelt und kommunizieren über sogenannte *Channels* miteinander. Diese dienen nicht nur dem Datenaustausch, sondern fungieren gleichzeitig als Puffer, um temporäre Lastspitzen auszugleichen. Durch diese Modularisierung wird sichergestellt, dass *I/O*-intensive Aufgaben wie das Lesen der Eingabedaten rechenintensive Prozesse wie die Paketkonstruktion nicht blockieren. In der Darstellung wird zudem zwischen Kontrollflüssen, die einmalig Informationen wie Startbefehle und Parameter übertragen, und kontinuierlichen Datenflüssen differenziert.

Das Startprogramm erstellt Paketrohlinge und dient der Eingabe sowie Übergabe der Konfigurationsdaten. Außerdem startet es das Scanner-Programm. Die Jobverwaltung hat hauptsächlich die Aufgabe, die anderen Komponenten korrekt zu vernetzen und zu starten. Die Paketemissionierung übernimmt die Paketbearbeitung, die Durchsatzlimitierung und

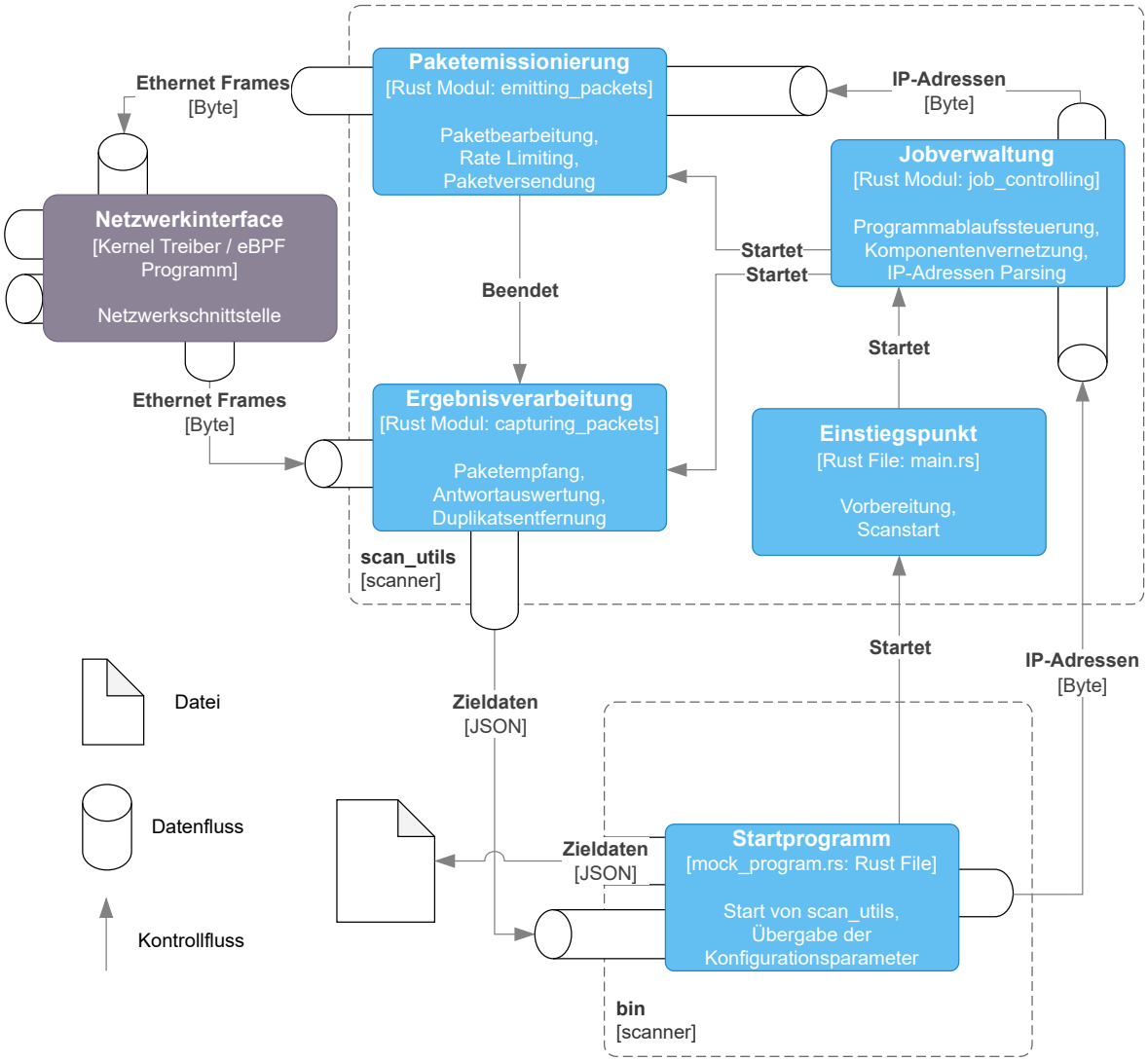

Abbildung 5.1: Diagramm logischer Komponenten des Scanners (vereinfacht)

das Versenden der Pakete. Für die Aufgabe des Empfangens und Auswerten der Antwortpakete sind zum einen ein `eBPF`-Programm zuständig, welches im Komponentendiagramm an der *Netzwerkinterface*-Komponente angesiedelt ist und zum anderen die Ergebnisverarbeitung. Letzteres beinhaltet die Logik für den Empfang der vom `XDP`-Programm übermittelten Daten im *User-Space* sowie für die anschließende Duplikatsentfernung. In dem Diagramm ist zu erkennen, dass zwischen der Paketemissionierungs-Komponente und der Ergebnisverarbeitungs-Komponente kein Datenfluss besteht, sondern lediglich das Signal zum Beenden des Scans ausgetauscht wird. Daran ist das zustandslose Design zu erkennen, welches die Anforderung Abschnitt 4.1.1 erfüllt.

In Abb. 5.1 wird der Weg der Pakete durch den Netzwerkkartentreiber und die Trennung der Zuständigkeiten von *User-Space* und Linux-Kernel nicht explizit behandelt. Um nun aber die Funktionsweise des `eBPF`-Programmes zu verbildlichen, wird in Abb. 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. Im Empfangsprozess ist zu sehen, dass das `eBPF`-Programm je nach Modus (*SKB Mode* oder *Driver Mode*) im Treiber der Netzwerkkarte oder direkt zu Beginn des Kernel-Netzwerkstacks ausgeführt wird. In beiden Fällen werden dort die eingehenden Pakete zuerst untersucht und je nach Ergebnis der Untersuchung direkt verworfen, an den Netzwerkstack weitergeleitet oder verändert und an den Treiber oder per `XDP_TX` direkt an die Netzwerkkarte zum Versenden zurückgegeben. So werden alle Schritte (oder im Falle des *SKB Mode* fast alle) des regulären Netzwerkstacks eingespart. Die Ergebnisse der Untersuchung im `eBPF`-Programm werden bei gültigen Paketen in eine `eBPF Map`, in diesem Fall einen `RingBuf` (siehe Abschnitt 5.3.1), geloggt. Dies hat den Vorteil, dass nur die relevanten Inhalte des Pakets (IP-Adresse, Port) statt des gesamten Pakets übermittelt werden müssen. Außerdem hat das *User-Space*-Programm direkten Zugriff auf den `RingBuf` und kann die Daten somit ohne Umwege abgreifen.

Auf diese Art und Weise kann der SYN-Scanner die Verarbeitungsschritte sowohl beim Senden als auch beim Empfangen von Paketen auf ein Minimum reduzieren, sodass CPU-Zyklen fast ausschließlich für die Anwendungslogik und nicht für das Betriebssystem aufgewendet werden. Dies validiert die konsequente Nutzung moderner Kernel-Mechanismen (Abschnitt 4.1.2) unter Berücksichtigung der technologischen Einschränkung auf das Linux-Ökosystem (Abschnitt 4.1.2) und bringt große Performanzpotenziale mit sich.

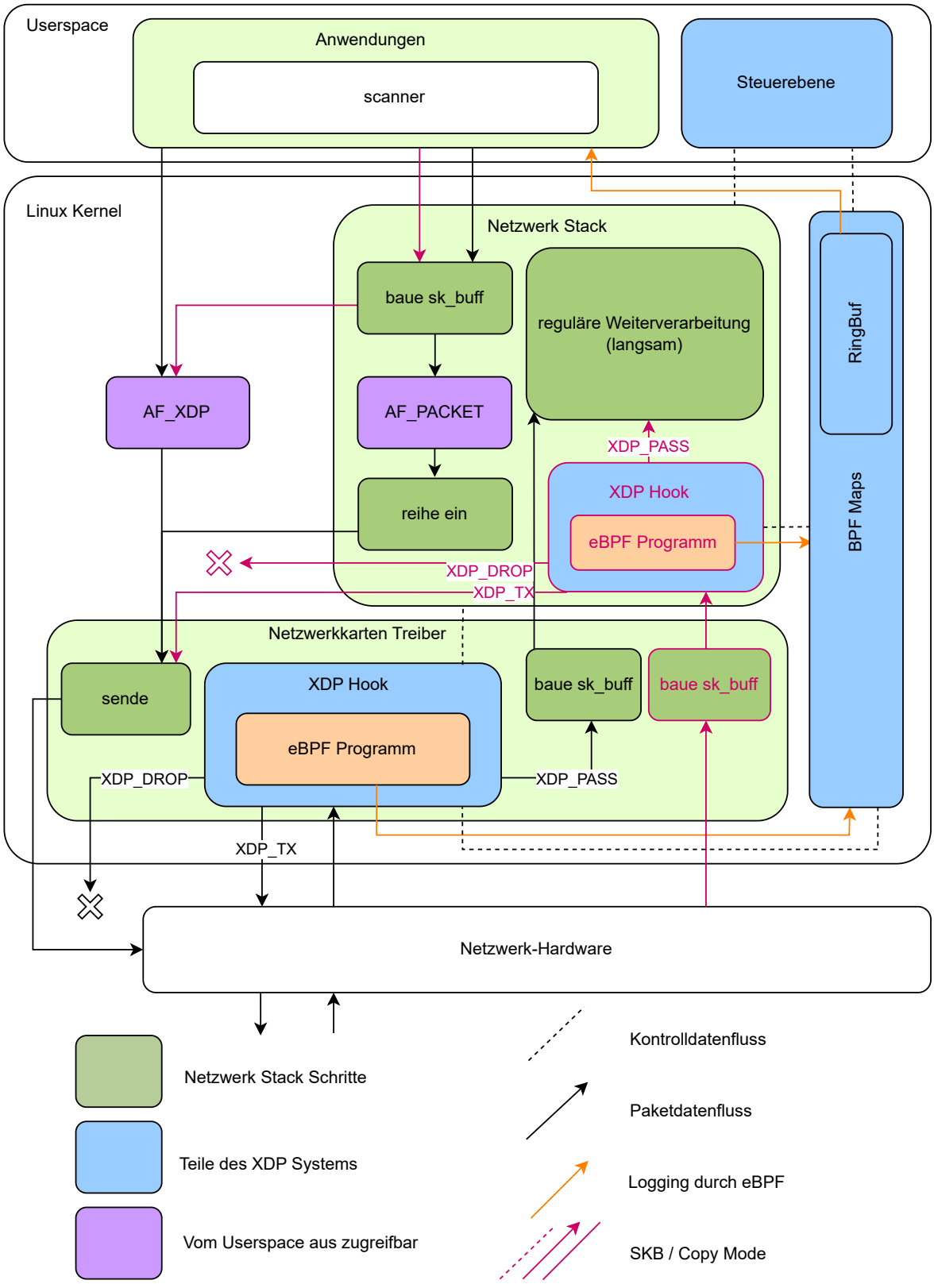

Abbildung 5.2: Weg der Pakete durch den Linux-Kernel im SYN-Rust Scanner (vereinfacht)

5.1.2 Performancesteigernde Maßnahmen

Neben der in Abschnitt 5.3.1 gezeigten Kernelumgehung werden zur Maximierung des Durchsatzes (Abschnitt 4.1.2) und Minimierung der Ressourcennutzung (Abschnitt 4.1.2) zwei weitere Maßnahmen verfolgt:

1. ***Batching*:** Sowohl im Datenaustausch zwischen den Komponenten mithilfe von *Channels* als auch beim Versenden von Paketen werden die Daten immer in *Batches* übertragen. Im Kontext der *Channels* bedeutet dies, dass statt einzelner Nachrichten Vektoren mit einer Vielzahl von Objekten übermittelt werden. Dies senkt den Synchronisationsaufwand pro Element erheblich, da die Anzahl der Interaktionen mit dem *Channel* minimiert wird. Für das Senden von Paketen reduziert das *Batching* die Anzahl der *System Calls*. Durch die gebündelte Verarbeitung mehrerer Pakete in einem Durchlauf wird der *Overhead* durch *Context Switches* zwischen *User-Space* und *Kernel-Space* verringert.
2. **Vermeidung von Kopieroperationen und *Context Switches*:** Zur Reduktion dynamischer Speicherzuweisungen (Allokationen) werden Vektoren, falls möglich, mit einer festen Kapazität initialisiert, um Reallokationen zur Laufzeit zu vermeiden. Bereits verwendete Vektoren werden nicht verworfen, sondern lediglich geleert und wiederverwendet. Zudem wurde beim Design auf die Vermeidung von Kopieroperationen geachtet. Das `eBPF`-Programm (siehe Abschnitt 5.3) arbeitet beispielsweise direkt auf Referenzen des Speicherbereichs (*Zero-Copy*), und auch die für das *Parsing* der IP-Adressen vom Startprogramm zuständige Komponente nutzt Mechanismen, um Kopieroperationen zu vermeiden (siehe Abschnitt 5.2.5).

5.2 Implementierung und Funktionsweise der Komponenten

Um die Funktionsweise des Programmes im Detail zu erklären, wird zuerst der Projektaufbau erklärt und anschließend die einzelnen Quelldateien vorgestellt und Besonderheiten bezüglich performanzsteigernder oder ressourcensparender Umsetzungen erläutert.

5.2.1 Projektstruktur Basisimplementierung

Die Verzeichnisse sind nach Aufgabenbereichen gegliedert, um klar zeigen zu können, welches Verzeichnis für welche Aufgabe zuständig ist und somit die Übersichtlichkeit zu steigern. Das Rust-Projekt hat folgende Ordnerstruktur:

Codeauszug 5.1: Ordnerstruktur des SYN-Scanners (gekürzt)

```
1 /scanner
2     /src
```

```
3         /bin
4             mock_programm.rs
5         /scan utils
6             /capturing_packets
7                 bucket.rs
8                 receiver.rs
9             /emitting_packets
10                assembler.rs
11                rate_limiter.rs
12                sender.rs
13            /job_controlling
14                parser_std_in.rs
15                scan_job.rs
16            /shared
17                helper.rs
18                types_and_config.rs
19        main.rs
20    Cargo.toml
21 /xdp-common
22    /src
23        lib.rs
24 /xdp-ebpf
25    /src
26        main.rs
27 ...
```

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

Die Zuordnung der Verzeichnisse zu den logischen Komponenten ist in Abb. 5.1 zu finden. Die Verzeichnisse `xdp-ebpf` und `xdp-common`, welche dort nicht explizit genannt werden, da sie im Kernel-Kontext ausgeführt werden, beschreiben das `eBPF`-Programm, welches Antwortpakete abfängt, auswertet und nur die relevanten Informationen an den *User-Space* weiterleitet. Das Verzeichnis `shared` dient lediglich der Steigerung der Übersichtlichkeit. Es enthält Hilfsfunktionen sowie Typenbeschreibungen, die mehrfach im Projekt genutzt werden. Somit dient es rein der Code-Organisation.

5.2.2 Übersicht genutzter *Crates*

Für die Umsetzung der Komponenten sind die in Tabelle 5.1 beschriebenen *Crates* aufgrund ihrer Wichtigkeit hervorzuheben.

Crate	Version	Nutzung
`tokio`	1.47.1	Nutzung für asynchrone Komponenten, Kommunikation über *Channels*, *Parsing* des *Standard Input* und Starten mehrerer asynchron laufender *Tasks*
`nix`	0.30.1	Erstellen der `AF_PACKET`-Schnittstelle und Versenden darüber
`xdp-socket`	0.1.4	Erstellen der `AF_XDP`-Schnittstelle und Versenden darüber
`aya`	0.13.1	Erstellung und Nutzung von `eBPF`-Programmen mittels bereitgestellter Werkzeuge und Strukturen
`dashmap`	6.1.0	Nutzung von für asynchrone Umgebungen optimierten *HashMaps* mit selbstständigem *Locking*
`pnet`	0.35.0	Nutzung als abstrahiertes Netzwerkwerkzeug zur Erstellung der *Packet Templates*
`network_types`	0.1.0	*Parsing* der *Header*-Strukturen aus rohem Speicherbereich, ohne diese zu kopieren

Tabelle 5.1: Genutzte *Crates*

5.2.3 Paketemissionierung (`emitting_packets`)

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

Rate Limiter (`rate_limiter.rs`)

Wie in Abb. 5.3 zu sehen, führt der *Rate Limiter* (`rate_limiter.rs`) die Funktion der Durchsatzlimitierung (Abschnitt 4.1.1) aus. Zuerst nimmt er die zu scannenden IP-Adressen vom *Parser* (`parser_std_in`) entgegen, bestimmt die Puffergröße anhand der in dieser Sekunde bereits gesendeten Datenmenge, füllt einen Puffer und erstellt für jeden Puffer einen `tokio` *Task* mit einem *Assembler* (`assembler.rs`). Sobald alle IP-Adressen initial verarbeitet sind, erfolgt der Durchlauf für die übrigen Zielports. Dabei wird ressourcenschonend auf den internen Puffer zurückgegriffen, wodurch erneutes Parsen entfällt.

`tokio` *Tasks*, auch als *Green-Threads* bekannt, sind leichtgewichtige, von der Laufzeitumgebung verwaltete Ausführungseinheiten. Im Gegensatz zu Betriebssystem-*Threads* blockieren sie bei Warteoperationen nicht, sondern geben Ressourcen dynamisch frei, was eine effiziente, asynchrone Nebenläufigkeit ermöglicht [56].

Diese Nebenläufigkeit wird hier genutzt, um entsprechend der aktuellen Senderate *Assembler* zu erzeugen, die nicht den gesamten *Thread* des Betriebssystems blockieren, wenn die Pakete eines spezifischen *Assemblers* nicht als Erstes vom *Sender* entgegengenommen werden. Stattdessen wartet jeder *Assembler*, ohne andere Teile der Software zu beeinträchtigen. So wird sichergestellt, 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 wie folgt ermittelt:

$$S_{opt}(R, B) = \begin{cases} 65536 & \text{falls } R = 0 \\ \left\lceil \frac{\text{clamp}(N_{target}, N_{min}, N_{max})}{B} \right\rceil & \text{sonst} \end{cases} \tag{5.1}$$

Wobei N_{target} die ideale Anzahl an Paketen pro Verarbeitungszyklus beschreibt:

$$N_{target} = \frac{R \cdot 10^6}{8 \cdot 60 \cdot 10} \tag{5.2}$$

Die Variablen sind hierbei wie folgt definiert:

- S_{opt}: Die berechnete optimale Puffergröße (in Anzahl der Batches).
- R: Die gewünschte Scan-Rate in Mbit/s.
- B: Die konfigurierte Batch-Größe (Anzahl Pakete pro Batch).
- N_{min}, N_{max}: Heuristische Grenzen für die Puffergröße (2048 bzw. 262144 Pakete).

Die numerischen Konstanten in Gleichung (5.2) haben folgenden Zweck:

- Der Faktor $\frac{10^6}{8}$ konvertiert die Rate R von Mbit/s in Byte/s.
- Der Wert 60 repräsentiert die Größe eines TCP-SYN-Pakets in Byte.
- Der Divisor 10 definiert die Ziel-Frequenz der *Wakeups* (10 Hz). Dies bedeutet, dass der Puffer so dimensioniert wird, dass er Daten für ein Zeitintervall von 100 ms vorhält.

Die optimale Puffergröße für die Paketerstellung wird dynamisch auf Basis der gewünschten Senderate und der konfigurierten *Batch*-Größe berechnet. Ziel ist es, genügend Daten für ein Verarbeitungsintervall von 100 ms vorzuhalten. Dies minimiert die CPU-Last, indem Kontextwechsel gespart werden, da die `tokio`-Runtime nur wenige *Assembler-Tasks* aufwecken muss.

Assembler (`assembler.rs`)

Die Rolle des *Assemblers* ist recht simpel: Jeder *Assembler* iteriert über die ihm verfügbaren IP-Adressen, füllt *Templates* mit der Ziel-IP-Adresse, dem Ziel-Port sowie der *Sequence Number* und berechnet die Checksummen des *IP-Header* und *TCP-Header* neu. Dies dient zur Erfüllung der Anforderung Abschnitt 4.1.1. Die *Sequence Number* wird wie folgt berechnet:

$$\text{ISN} = \text{SipHash}_K(\texttt{src\_ip}, \texttt{dst\_ip}, \texttt{src\_port}, \texttt{dst\_port}) \tag{5.3}$$

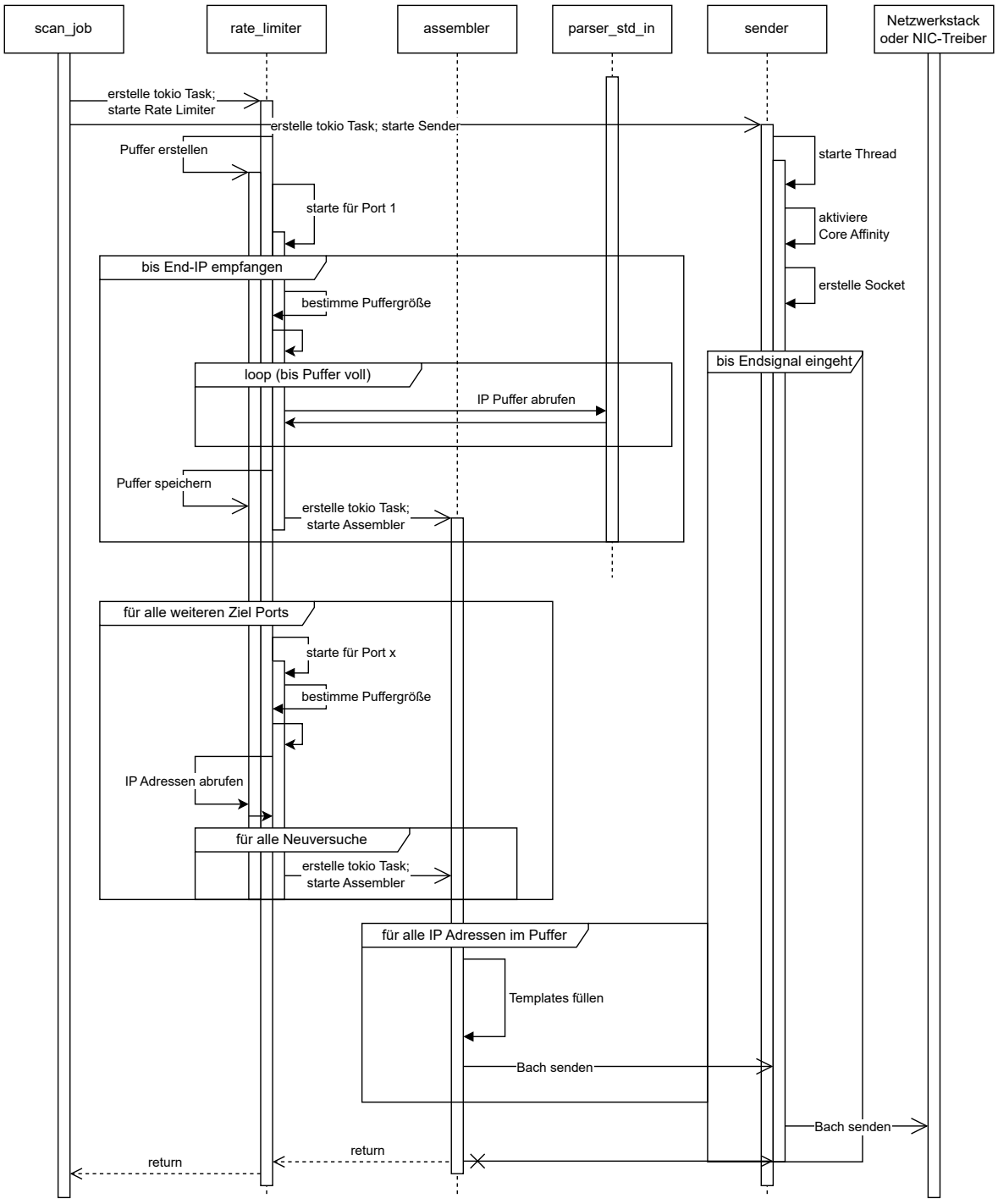

Abbildung 5.3: Ablauf und Funktionsweise der `emitting_packets`-Komponente (vereinfacht)

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 Hash-Funktion entsprechend bei gleichem Input immer den gleichen Wert zurückgibt [25]. Damit dies konsistent funktioniert, muss allerdings ein geheimer Schlüssel genutzt werden, welcher der Paketemissionierungs- sowie der `eBPF`-Komponente bekannt ist. In den *Templates* sind die restlichen Werte bereits vorhanden. Die Änderungen werden direkt auf Byte-Ebene umgesetzt, da die Feldzuweisungen der *Header*-Felder immer gleich sind [22] [21]. Somit können vollständige Pakete in sehr wenigen Schritten und ohne aufwendiges *Parsing* oder gar komplette Neuerstellung genutzt werden. Diese Pakete werden anschließend je nach Konfiguration in *Batches* an den *Sender* weitergeleitet.

Sender (`sender.rs`)

Im Gegensatz zu den übrigen Subkomponenten operiert der *Sender* in einem dedizierten Betriebssystem-*Thread*. Dies gewährleistet die exklusive Nutzung der verfügbaren Rechenkapazität und minimiert Prozesskonflikte. Zur weiteren Optimierung wird mittels der `core_affinity`-*Crate* (siehe Tabelle 5.1) eine feste Bindung an einen CPU-Kern erzwungen. Dies verbessert die *Cache*-Lokalität und verhindert teure Wechsel zwischen Kernen, was für konsistent hohe Senderaten von Vorteil ist [52, S. 181]. Der *Sender* verarbeitet in einer Endlosschleife eingehende Paket-*Batches* über die beim Start initialisierte Netzwerkschnittstelle (vgl. Abschnitt 4.1.1), bis die Kommunikationskanäle geschlossen werden.

Um eine fundierte Vergleichsbasis zu schaffen und den tatsächlichen Mehrwert komplexer Verfahren zu evaluieren, unterstützt die Implementierung sowohl `AF_PACKET` als auch `AF_XDP` als Sende-Backend.

5.2.4 Ergebnisverarbeitung (`capturing_packets`)

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

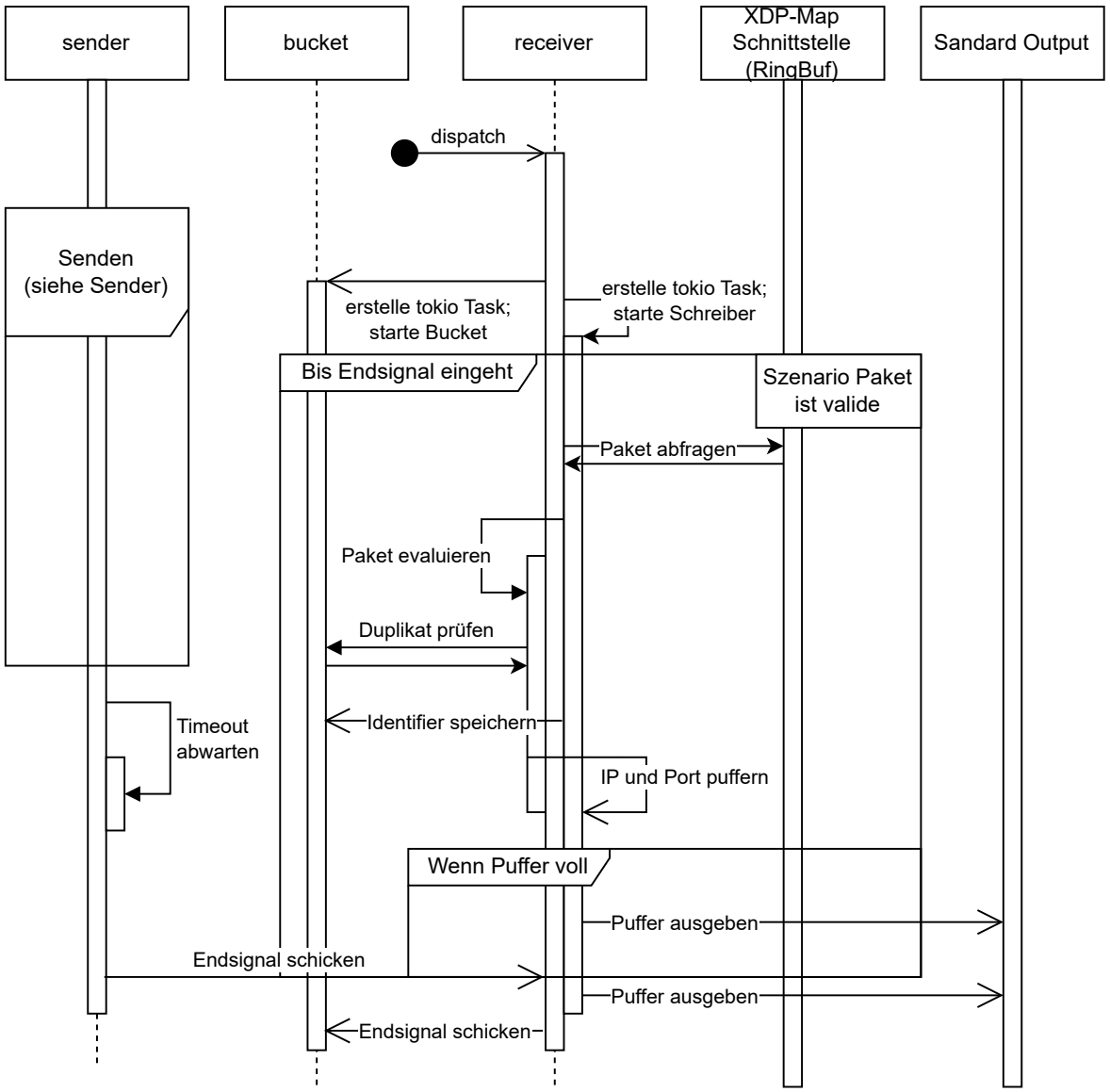

Abbildung 5.4: Exemplarisches Diagramm zur Funktionsweise der `capturing_packets`-Komponente (vereinfacht)

Receiver (`receiver.rs`)

In der initialen Entwicklungsphase (siehe Abb. A.1) operierte der *Receiver*, ebenso wie der *Sender*, in einem dedizierten Betriebssystem-*Thread*. Um Anforderung Abschnitt 4.1.1 zu erfüllen, kam dabei die Bibliothek `pcap` zum Einsatz, welche eine komfortable Abstraktion für den Netzwerkzugriff bietet und die Filterung via *Berkeley Packet Filter* ermöglicht. Ein- und ausgehende Pakete wurden empfangen, mittels `etherparse` analysiert und anschließend auf Duplikate geprüft.

Trotz der entwicklerfreundlichen Handhabung wurde die Architektur zugunsten des deutlich leistungsfähigeren `eBPF`-Ansatzes umgestellt, um den *High-Performance*-Ansprüchen dieses Projekts zu genügen. `pcap` basiert intern auf *Raw-Sockets* (`PF_PACKET`[1]) [57], weshalb Pakete deutlich mehr Schritte als bei Nutzung eines `eBPF`-Programmes durchlaufen müssen (vgl. Abb. 5.2).

Im aktuellen Ansatz läuft der *Receiver* stattdessen in einem `tokio` *Task*, 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 gültiger Daten. Im ersten Schritt werden Daten aus dem `RingBuf` abgerufen. Da der Zugriff auf den `RingBuf` über einen Unix-Dateideskriptor erfolgt, dessen Leseoperationen standardmäßig den *Thread* blockieren, muss dieser in das asynchrone Modell der Anwendung integriert werden. Dafür bietet `tokio` eine Lösung, welche es ermöglicht, durchgehend auf neue Pakete zu warten, ohne den ausführenden *Thread* zu blockieren. Anschließend wird die Ziel-IP-Port-Kombination der Duplikatprüfung, welche im nächsten Absatz genauer beschrieben wird, unterzogen. Bei gültigen Ergebnissen (kein Duplikat) werden diese in einem Puffer gesammelt, welcher entweder bei Erreichen der maximalen Größe geleert wird. Die Daten werden über einen *Channel* an einen separaten Schreiber-*Task* übergeben, welcher die Ergebnisse effizient blockweise in den *Standard Output* schreibt. Dies minimiert *System Calls* für I/O-Operationen und erfüllt Anforderung Abschnitt 4.1.1.

Die Weiterleitung der Ergebnisse in eine Datei erfolgt anschließend durch das aufrufende Programm (siehe `mock_program.rs`).

Bucket (`bucket.rs`)

Zur effizienten Duplikaterkennung wird ein *Timed Bucket System* genutzt, in welchem mehrere *Buckets*, welche intern eine *DashMap* nutzen (siehe Tabelle 5.1), als Zwischenspeicher für die bisherigen Antworten dienen. Es wird nur in den derzeit aktiven *Bucket* geschrieben, doch aus allen wird gelesen. Nach einer festen Zeiteinheit[2] wird der nächste *Bucket* aktiv und der am längsten inaktive geleert.

[1] Funktional äquivalent zu `AF_PACKET` [27]
[2] $1/x, x =$ Anzahl der Buckets

Durch die Aufteilung in mehrere *Buckets* sollen Lastspitzen durch das Leeren einer sehr aufgeblähten *DashMap* verhindert werden. Außerdem werden dadurch längere *Locking*-Zeiten bei asynchronen Schreib- und Lesezugriffen vermieden. Die Suche nach Duplikaten gestaltet sich dabei schnell, da *DashMaps* eine Suche der Zeitkomplexität $O(1)$ ermöglichen.

5.2.5 Programmstart und Jobverwaltung (`job_controlling`)

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

Startprogramm (`mock_program.rs`)

Um die Anforderung Abschnitt 4.1.1 zu erfüllen, nimmt der Scanner die IP-Adressen der Ziele über den *Standard Input* entgegen. Für die Evaluation in dieser Arbeit wurde, um die Vergleichbarkeit herzustellen, dieses Startprogramm erstellt. Es hat die Aufgabe, den Scanner zu starten, *Ethernet Templates* zu erstellen und Daten in den *Standard Input* zu schreiben sowie Daten aus dem *Standard Output* des Scanners zu lesen. Im Startprogramm werden auch die Konfigurationsparameter eingetragen.

Die *Ethernet Templates*, welche das Fundament zur Erfüllung der Anforderung Abschnitt 4.1.1 darstellen, werden mithilfe des `pnet` *Crates* erstellt, da dieser eine entwicklerfreundliche Schnittstelle dafür bereitstellt. Dort werden alle Parameter für eine reguläres `TCP-SYN`-Paket bis auf die Ziel-IP, den Ziel-Port und die *Sequence Number* gesetzt. Es wird für jede Quell-IP ein *Template* angelegt, um die Streuung der Paketquellen zur Verschleierung des Scans und somit die Trefferrate zu erhöhen.

Einstiegspunkt (`main.rs`)

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

Standard Input Parser (`parser_std_in`)

Der *Parser* parst zum einen die Konfigurationsparameter aus dem *Standard Input* und zum anderen die Ziel-IP-Adressen. Das *Parsing* der im Binärformat übertragenen Daten erfolgt unter Berücksichtigung verschiedener Optimierungsmaßnahmen. So wird ein asynchroner `tokio`-Reader eingesetzt, um zu gewährleisten, dass der Einleseprozess auch

bei ausgelasteten Kommunikationskanälen andere Programmteile nicht blockiert. Gemäß den in Abschnitt 5.1.2 definierten Prinzipien werden zudem *Batching* und *Zero-Copy*-Techniken angewandt (vgl. **??** 5.2). Letzteres wird unter anderem durch die Nutzung von `std::mem::replace` realisiert, womit Speicherpuffer effizient durch den Austausch von *Ownership* verwaltet werden. Dies ersetzt kostenintensive Kopieroperationen.

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

```
/* Weiterer Code */
// feste Kapazitaet
let mut ip_batch: Vec<[u8; 4]> = Vec::with_capacity(BATCH_SIZE / 4);

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

    // Verarbeite vollstaendige IP-Pakete
    while cursor + 4 <= valid_data_len {
        let bytes: [u8; 4] = buffer[cursor..cursor + 4].try_into().unwrap();
        cursor += 4;
        if bytes == [0, 0, 0, 0] { /* Terminator -> return */ }
        ip_batch.push(bytes);
    }

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

    /* Weiterer Code */
```

Scanjob (`scanjob.rs`)

Ein *Scanjob* fungiert als zentrale Orchestrierungseinheit für alle *User-Space*-Komponenten des SYN-Scans. Wie in Abb. A.2 dargestellt, beginnt der Ablauf mit der Initialisierung der `tokio` *Channels*, um die spätere Vernetzung der Komponenten zu gewährleisten. Anschließend werden die einzelnen Module nacheinander als asynchrone `tokio`-Tasks gestartet: Zunächst der `parser_std_in`, gefolgt von *Receiver* und *Sender*. Danach wird der *Rate Limiter* gestartet, woraufhin der *Scanjob* in einen Wartezustand übergeht, bis alle Subsysteme ihre Arbeit durch entsprechende Signale als beendet melden.

Da das Anheften eines neuen `XDP`-Programmes einen Neustart des Netzwerkkartentreibers erfordert, wird vor der Erstellung der Sende-*Sockets* eine kurze Zeit gewartet. Auch nach dem Starten des *Senders* und des *Receivers* wird kurz gewartet, damit alles auf Abruf

ist, sobald der *Rate Limiter* mit der Produktion der SYN-Pakete beginnt. Dies dient der allgemeinen Stabilität des Programmes. Wie bereits in den meisten anderen Komponenten wird auch hier tokio zur nebenläufigen Ausführung der verschiedenen Programmteile genutzt. Dies ermöglicht das unabhängige Handeln der einzelnen Bestandteile, was einerseits der Notwendigkeit des gleichzeitigen Sendens und Empfangens entspringt und andererseits der *Performance*-Steigerung durch Einsparung von Wartezeiten dient.

5.3 eBPF

Der eBPF dient dem Scanner als Empfangspunkt für eingehende Pakete und realisiert die Anforderung an moderne Kernel-Mechanismen (siehe Abschnitt 4.1.2). Je nachdem in welchem XDP-Modus das Programm ausgeführt wird, agiert dieses direkt im Treiber der Netzwerkkarte oder an der ersten Stelle nach Erstellung eines Puffers im Netzwerkstack. Dadurch können die Pakete bereits am frühestmöglichen Punkt evaluiert, deren relevante Daten extrahiert und direkt ohne Umweg über den geteilten Speicher der eBPF-Maps in das *User-Space*-Programm übertragen werden (siehe Abb. 5.2). Dies führt zu einer massiven Ressourceneinsparung und das wiederum zu einer Zeiteinsparung, da etliche Zwischenschritte, welche normalerweise durchlaufen werden müssten, um das Paket durch den Netzwerkstack zum *User-Space* zu leiten und das Wiederversenden ohne Kopieraufwand oder *Context Switches* passiert. Über die XDP_TX-Funktion werden RST-Pakete direkt im eBPF-Programm erstellt und wieder verschickt, sodass der Netzwerkstack sowie *User-Space* komplett vermieden werden.

5.3.1 XDP-Programm

Die Integration des eBPF-Programms in den Kernel-*Hook* erfordert die Kompilierung in das *ELF*-Format. Diese Aufgabe, inklusive dem Laden des Programms und der Verwaltung der eBPF Maps (siehe Tabelle 5.2), wird durch das aya-*Crate* abstrahiert und vereinfacht. Die Bindung an das Netzwerkinterface erfolgt in der main.rs.

Name	Typ	Nutzung
STATS	PerCpuArray<u64>	Effiziente, lockfreie Protokollierung von Statistiken pro CPU [58]
WHITELIST_IPV4	HashMap<[u8; 4], u8>	*HashMap* zum Abgleich der für den Scan genutzten Quell-IP-Adressen
EVENTS	RingBuf	Effizienter, geteilter Puffer-Ring [34] zur Übermittlung der extrahierten Zielinformationen valider Pakete an den *Receiver*
SIPHASH_KEY	Array<u64>	Schlüssel zur korrekten Auswertung des SYN-Cookies

Tabelle 5.2: Genutzte eBPF Maps

5.3.2 Funktionsweise

Das eBPF-Programm (siehe Abb. 5.5) extrahiert und validiert initial die Ethernet-, IP- und TCP-Header aller eingehenden Pakete. Pakete, die kein valides *IPv4*-SYN-ACK darstellen oder deren SYN-Cookie-Prüfung (mittels *SipHash*), beziehungsweise der Abgleich mit der Quell-IP-Whitelist (WHITELIST_IPV4) fehlschlägt, werden unverändert mittels XDP_PASS an den regulären Netzwerkstack übergeben. Dies gewährleistet, dass der Standard-Netzwerkverkehr unbeeinträchtigt bleibt, während valide Antworten gemäß Abschnitt 4.1.1 weiterverarbeitet werden.

Die Programmierung im Kernel-Kontext unterliegt strikten Restriktionen (vgl. Abschnitt 2.3.3). Da weder Systemaufrufe noch eine Speicherverwaltung verfügbar sind [59, S. 59, 206], kommen dynamische Datenstrukturen nicht in Frage. Zur typsicheren Verarbeitung ohne Kopieroperationen wird daher das network_types-*Crate* eingesetzt, welches rohe Speicherbereiche direkt auf typisierte Strukturen abbildet. Die Navigation im Speicher erfolgt mittels Zeigerarithmetik durch die ptr_at-Funktion, welche außerdem die Grenzen des Datenbereichs validiert.

Codeauszug 5.3: ptr_at-Funktion zum Navigieren durch Speicherbereiche

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

Folgendes Beispiel demonstriert die Extraktion des *IP-Headers* unter Verwendung des Offsets des vorangegangenen *Ethernet-Headers* (16 Byte):

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

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

Dabei ist zu beachten, dass diese Speicherzugriffe durch unsafe-Blöcke umschlossen sind. Gemäß Abschnitt 4.1.2 ist dies zulässig, da der direkte Zugriff auf Kernel-Speicher für den angestrebten *Zero-Copy*-Ansatz zwingend notwendig ist. Die Standard-Sicherheitsgarantien von Rust basieren üblicherweise auf Laufzeitüberprüfungen (z.B. *Bounds Checks* bei *Slices*),

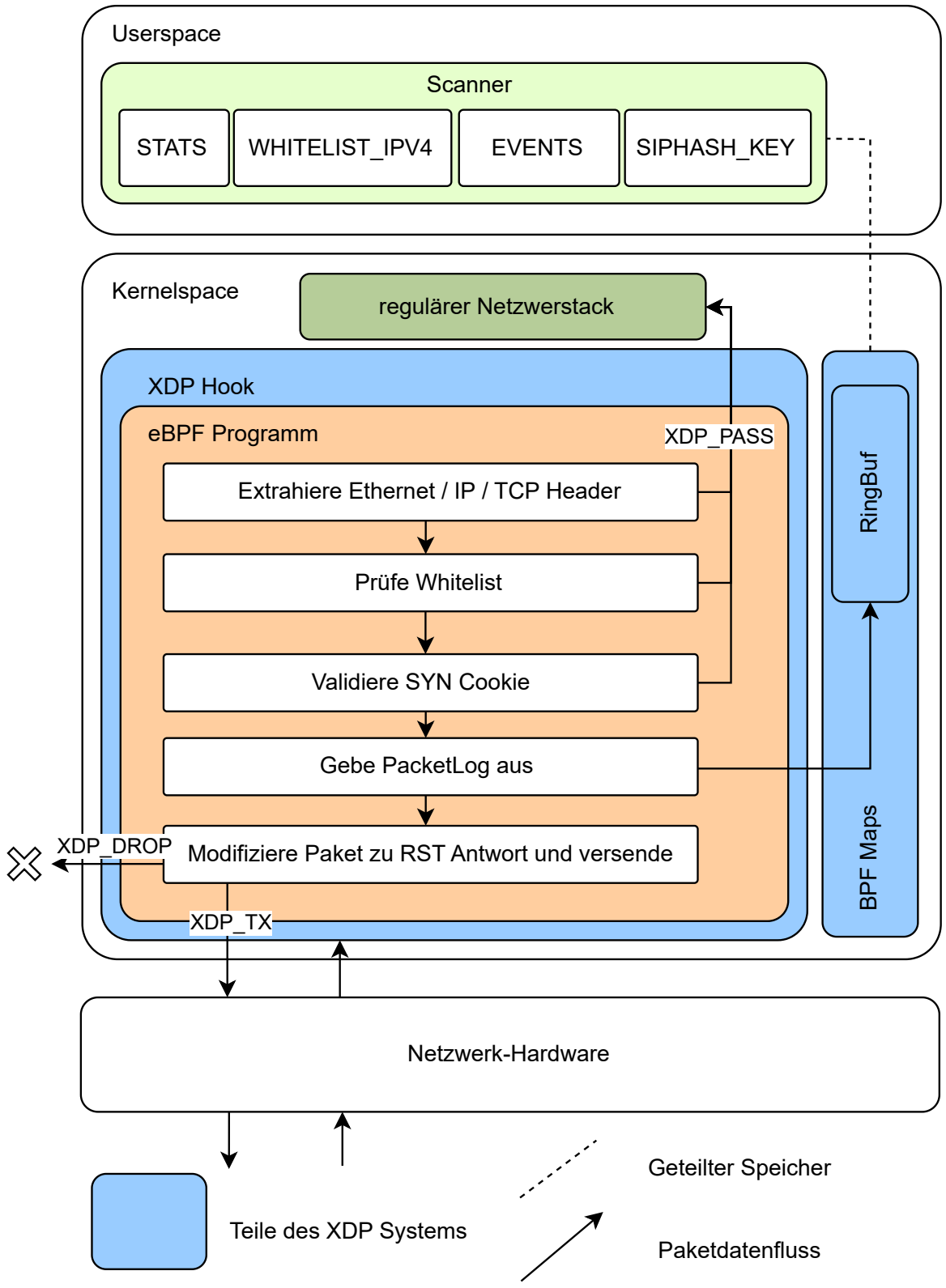

Abbildung 5.5: Funktionsweise des eBPF-Programmes (vereinfacht)

die bei Fehlern in einen Programmabbruch (`panic!`) resultieren. Da ein solcher Abbruch im Kernel-Kontext unzulässig ist und das alternative Anlegen einer sicheren Speicherkopie nicht den Performanzzielen dieser Arbeit entsprechen würde, wird von den Vorteilen der spezifischen Restriktionen der `eBPF`-Laufzeitumgebung (vgl. Abschnitt 2.3.3) Nutzen gezogen. Obwohl der *Borrow-Checker* lokal umgangen wird, übernimmt der `eBPF`-*Verifier* des Linux-Kernels die globale Sicherheitsgarantie. Dieser führt bereits zur Ladezeit eine strenge statische Code-Analyse durch und verweigert die Ausführung des Programms, falls theoretisch ungültige Speicherzugriffe möglich wären. Somit dient Rust in dieser Architektur primär der Typsicherheit und Strukturierung, während der *Verifier* die Rolle der Instanz zur Durchsetzung der Speichersicherheit übernimmt.

Als gültig identifizierte Antworten werden über eine `PacketLog`-Struktur im `RingBuf` effizient in den *User-Space* übertragen. Anschließend erfolgt die Modifikation des Pakets zu einem `RST`-Paket direkt im Speicher (siehe Abb. 5.6). Durch *In-Place*-Manipulation der *Header*-Felder und Neuberechnung der Prüfsummen wird die Verbindung zum Zielsystem standardkonform beendet (Abschnitt 4.1.1), ohne dass neue Speicherallokationen nötig sind.

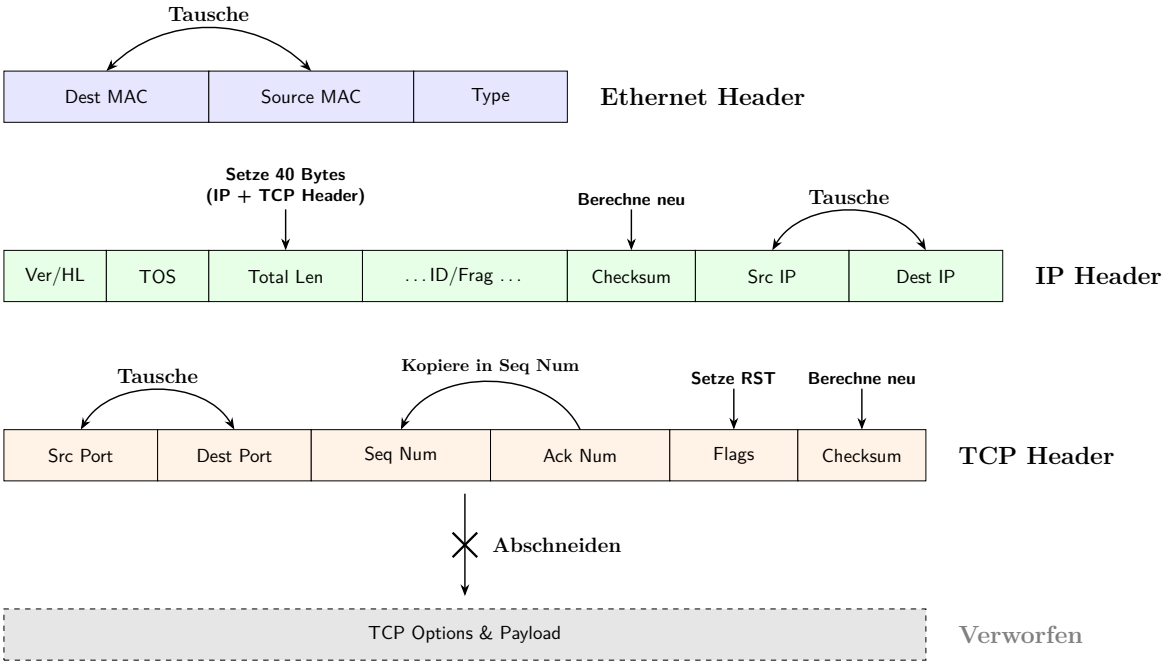

Abbildung 5.6: *In-Memory*-Modifikation des `SYN-ACK`-Pakets zum `RST`-Paket

Das fertige Paket wird anschließend per `XDP_TX` direkt in den Puffer der Netzwerkkarte geschrieben und durch die Modifikationen als gültiges `RST`-Paket an den Absender der ursprünglichen `SYN-ACK`-Antwort zurückgeschickt. Dies dient der Erfüllung der Anforderung Abschnitt 4.1.1.

Kapitel 6: Testumgebung und Durchführung

Dieses Kapitel beschreibt den konzipierten Versuchsaufbau sowie die methodische Durchführung der Tests. Es werden die Hardware- und Softwareumgebung spezifiziert, der Ablauf der Benchmarks dargelegt und bekannte Limitierungen der Testumgebung erörtert.

6.1 Versuchsaufbau

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. Außerdem wird auf beiden eine statische IP-Adresse eingerichtet, sodass eine stabile Netzwerkschicht-Verbindung besteht und eine konstante Zieladresse für die Reproduzierbarkeit der Messungen definiert ist.

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

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

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

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

Für das *Proof of Concept* (siehe Kapitel 4) werden die Scanausgaben, sowie die in Tabelle 6.1 beschriebenen Tools genutzt, um zusätzlich zu den Ausgaben des *Dummy-Receiver*-Programmes eine exakte Validierung anstellen zu können.

Tool	Nutzung
Netcat	Fungiert als Referenz-Responder zur Validierung der Paketstruktur. Da Netcat nur auf standardkonforme Anfragen reagiert, bestätigt eine erfolgreiche Kommunikation die korrekte Konstruktion der vom Scanner gesendeten Pakete.
`xdpdump`	Dient der Erfassung des Netzwerkverkehrs direkt an der `XDP`-Ebene. Dies ist essenziell, da der Scanner die Pakete bereits auf Treiberebene verarbeitet und diese somit für herkömmliche *Sniffer* (die auf dem *Socket-Layer* arbeiten) teilweise nicht sichtbar wären.
Wireshark	Wird zur detaillierten Analyse und Validierung der mit `xdpdump` aufgezeichneten `.pcap`-Dateien verwendet. Es ermöglicht die manuelle Überprüfung der *Header*-Felder, *Sequence Numbers* und *Flags* auf Konformität mit den Protokollspezifikationen.

Tabelle 6.1: Genutzte Tools zur Validierung der funktionalen Anforderungen (siehe Kapitel 4)

In den Evaluationsszenarien (Abschnitt 4.2.2) erfolgt die Messung der Metriken (siehe Kapitel 4) mit einer Abtastrate von 10 Hz. Da die Scanner Subprozesse und mehrere *Threads* starten, wird die gesamte Systemlast gemessen. Das *Benchmarking*-Programm verzichtet auf *High-Level*-Tools, um den *Overhead* der Messung selbst zu minimieren und liest die erforderlichen Kernel-Statistiken direkt aus `procfs`. Dies ist ein virtuelles Dateisystem, welches der Anzeige und Änderung von Systemparametern dient [60] und eine direkte Schnittstelle zu den internen Datenstrukturen des Linux-Kernels bietet.

6.1.1 Aufbau des Ziel-Knotens

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

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

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

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

6.1.2 Hardware-Spezifikation

Für die Umsetzung des Testes wurden die in Tabelle 6.2 spezifizierten Systeme verwendet. Es wurden keine Veränderungen vorgenommen, die das Betriebssystem von seinem nativen Zustand abbringen, um Anforderung /NF-06/ zu genügen. Die einzige Konfigurationsmaßnahme außerhalb des Programmes ist eine softwareseitige Anpassung der Sende-, sowie Empfangsringe der Netzwerkkarte auf das Maximum der bereitgestellten Hardware. Dies ist in diesem Szenario zu empfehlen, da es sich explizit um ein *High-Performance*-Szenario handelt. Das Anpassen der genutzten Puffer dient dem Abfangen von Lastspitzen, indem mehr Pakete vom Netzwerkkartentreiber gepuffert werden können. Für die genutzten Netzwerkkarten bedeutet dies konkret eine Erhöhung der sogenannten Deskriptoren von 256 auf 4096 (siehe **Abschnitt A.2**).

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

Tabelle 6.2: Hardware- und Software-Spezifikationen der Testumgebung im Vergleich

6.2 Versuchsablauf

Die `benchmark_suite.py` [55], welche für Test 1 (Evaluationstest 1), Evaluationsszenario 1 (Evaluationsszenario 1) und Szenario 2 (Evaluationsszenario 2) genutzt wird, startet die Scanner-Prozesse nacheinander und misst mit einem parallelen *Monitoring-Thread* die Systemressourcen in folgenden Phasen:

1. **Ruhezustands-Messung:** Vor den eigentlichen Tests wird über einen Zeitraum von fünf Sekunden der Systemzustand ohne Last gemessen. Dieser Durchschnittswert dient

als Referenzpunkt, um das Grundrauschen des Betriebssystems später herausrechnen zu können.

2. **Aufzeichnung:** Die Aufzeichnung der Metriken beginnt eine Sekunde vor dem Prozessstart, um das Anlaufverhalten und Initialisierungsspitzen der Scanner vollständig zu erfassen.

3. **Aktive Phase:** Während der Scanner läuft, werden regelmäßig Daten ausgelesen und persistiert. Der Scanner gilt als aktiv, sobald die Senderate 100 *PPS* überschreitet und als inaktiv, sobald die Grenze wieder unterschritten wird.

4. **Externes Beenden:** Sollte die Rate nach dem Start für mehr als sieben Sekunden unter einen Schwellenwert von 100 *PPS* fallen, wird der Prozess terminiert. Dies ist notwendig, da Masscan sich häufig nicht von alleine beendet.

In Test 2 (Evaluationstest 2) werden die Programme und Tools (siehe Tabelle 6.1) manuell gestartet und ausgewertet.

6.2.1 Datenaufbereitung und -erhebung

Um die Scanergebnisse zu plausibilisieren wird zusätzlich via `validate_responses.py` [55] die Anzahl der ausgegebenen Ergebnisse gezählt. Die Rohdaten werden nach der Messung mithilfe des `plot_benchmark_suite.py` [55] statistisch bereinigt und visualisiert. Eine einfache Mittelwertbildung über die gesamte Laufzeit alleine ist nicht zielführend, da Start- und Stopp-Phasen die Ergebnisse verzerren würden. Stattdessen werden die Ergebnisse mit folgenden Vorgehensweisen aufbereitet:

- **Isolation der Hochlastphase:** Für die Berechnung des durchschnittlichen Durchsatzes und der Effizienz (*PPS*/CPU-Auslastung) werden nur jene Zeitfenster berücksichtigt, in denen der Scanner aktiv sendet.

- **Netto-Ressourcenberechnung:** Von den gemessenen CPU- und RAM-Werten wird der in Phase 1 (siehe Kapitel 4) ermittelte *Baseline*-Wert subtrahiert. Dies stellt sicher, dass die dargestellten Ergebnisse ausschließlich den Ressourcenbedarf des Scanners abbilden und unabhängig von Hintergrundprozessen des Betriebssystems sind.

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

In Test 1 (Evaluationstest 1) wird zur Auswertung `validate_scanner.py` [55] genutzt, welches die Ausgaben der Scanner bezüglich der Anzahl, Duplikate und der Korrektheit (Ein Paket pro gerader *IP*-Adresse des genutzten Adressraumes) validiert. Außerdem werden die *Logs* des Ziel-Knotens genutzt. Die Metriken (siehe Kapitel 4) und das `plot_benchmark_suite.py`-Programm sind hier nicht notwendig.

In Test 2 (Evaluationstest 2) stellen die Ausgaben von `xdpdump` und der Betrachtung dessen in Wireshark die Testergebnisse dar.

6.3 Genutzte Parameter

Im Folgenden werden die relevanten zur Umsetzung der Tests genutzten Parameter erläutert.

6.3.1 *Proof of Concept*

Um die Tests zur Validierung (siehe Kapitel 4) der Funktionsweise korrekt auszuführen, werden folgende Parameter genutzt:

1. Hierbei wird ein kleiner *IP*-Adressraum (`/20`) gescannt. Der Ziel-Knoten wird so konfiguriert, dass er auf alle *IP*-Adressen, welche mit einer geraden Zahl enden, antwortet, um die anschließende Auswertung zu vereinfachen. Dabei wird das Senden von `RST`-Paketen aktiviert.

2. Im zweiten Test werden nur jeweils 4 Pakete verschickt. Dies genügt, um zu erkennen, ob die Pakete korrekt sind und fördert die Übersichtlichkeit. Um auch die `RST`-Pakete zu testen, wird dieser Test einmal mit der `RST`-Funktion und einmal ohne durchgeführt.

6.3.2 Evaluationsszenarien

Im Folgenden wird die technische Umsetzung der in Kapitel 4 definierten Evaluationsszenarien beschrieben. Die Szenarien werden mit allen drei zu evaluierenden Scanner-Varianten (Rust-`XDP-Copy`, Rust-`XDP-ZeroCopy`, Rust-`AF_PACKET`) sowie den Vergleichstools (ZMap, Masscan) durchgeführt.

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

Evaluationsszenario 1 - Performanzgrenze

Um die *Performance* und Effizienz der Scanner zu validieren, werden alle bremsenden Faktoren, die nicht essenziell für das reine Versenden und Empfangen von Paketen sind, oder die Vergleichbarkeit stören könnten deaktiviert:

- **Senderate:** Die Senderate wird so gewählt, dass sie das theoretische Limit der Gigabit-Leitung übersteigt.
- ***Features*:** Es wird auf rechenintensive Funktionen wie die Deduplizierung von Antworten, sowie das Senden von `RST`-Antworten verzichtet.
- ***IP*-Raum:** Als Ziel dient ein `/6`-Netzwerk[1], um eine hinreichend lange Laufzeit für die Erfassung stabiler Messwerte zu gewährleisten.
- **Zielport:** Ein einzelner Zielport (80) wird genutzt.
- **Quellport:** Ein einzelner Quellport (60000) wird genutzt.

Evaluationsszenario 2 - Reale Umstände

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

- **Senderate:** Die Senderate wird auf 500.000 *PPS* fixiert, da sehr hohe Raten die Wahrscheinlichkeit erhöhen, dass Scan-Muster von *Firewalls* oder *IPS* erkannt und blockiert werden.
- ***Features*:** Die Deduplizierung wird aktiviert, auch wenn der Ziel-Knoten nur eine Antwort pro Paket schickt. Das Senden von `RST`-Antworten wird aktiviert.
- ***IP*-Raum:** Der Zielbereich wird auf ein `/10`-Netzwerk[2] beschränkt, um die Gesamtdauer des Tests in einem praktikablen Rahmen zu halten.
- **Zielports:** Der Scan erfolgt auf die Ports 80 und 443, um das Scan-Verhalten zu diversifizieren und weiter zu verschleiern.
- **Quellports:** Es wird ein Bereich von 128 *Source-Ports* $(60000 - 60127)$ verwendet. Dies dient der besseren Lastverteilung auf der Empfängerseite und Verschleierung des Scans.

[1] 67.108.864 *IP*-Adressen
[2] 4.194.304 *IP*-Adressen

6.4 Inkompatibilitäten und Limitierungen

Bei der Realisierung der Testumgebung wurden spezifische hardware- und treiberbedingte Einschränkungen identifiziert. Dieser Abschnitt beleuchtet deren Auswirkungen auf die Testdurchführung, insbesondere im Hinblick auf den *Zero-Copy*-Modus und die maximal erzielbaren Paketraten.

6.4.1 *Zero-Copy*-Modus

Die Evaluation unterliegt Einschränkungen durch die verwendete Hardware (vgl. Tabelle 6.2). Der effiziente *Zero-Copy*-Modus ist bei Netzwerkkarten mit dem `igb`-Treiber nur bedingt nutzbar, da die geringe Anzahl verfügbarer Hardware-Ringe keine exklusive Zuweisung von *Queues* an das `XDP`-Programm erlaubt. Dies erzwingt das Teilen der Sende-Ringe zwischen dem Betriebssystem und dem `AF_XDP`-*Socket*, was unweigerlich zu *Lock Contention*[3] führt [61].

Dieser Ressourcenkonflikt äußerte sich in internen Tests. Bei Nutzung von vier *Queues* fehlten unter Last konstant rund 25 % der `RST`-Pakete am Ziel-Knoten. Da der Scanner primär über eine dedizierte *Queue* sendet, kollidiert der Antwortverkehr auf genau dieser einen von vier *Queues*, während die anderen drei *Queues* die `RST`-Pakete ungehindert verarbeiten konnten. Gegenproben mit einer Limitierung der Hardware-Ringe (`ethtool -L ...`) verifizierten dies: Bei Reduktion auf eine einzige *Queue* (totale Kollision) stieg der Verlust auf nahe 100 %, bei drei *Queues* (Kollision auf einer von drei) lag er bei ca. 33 %.

Ein per `XDP_TX` generiertes `RST`-Paket wird standardmäßig über dieselbe *Queue* ausgesendet, auf der das auslösende Paket empfangen wurde [62]. Da der eingehende Antwortverkehr durch *Receive Side Scaling* (RSS) mittels eines Hash-Verfahrens gleichmäßig auf alle verfügbaren *Queues* verteilt wird [63, S. 306], trifft ein Teil des Verkehrs auch die *Queue*, die der Scanner bereits unter Volllast zum Senden nutzt. Die Kombination aus hoher Senderate, *Lock Contention* und den vergleichsweise kleinen Puffern der Hardware-Ringe (max. 4096 Deskriptoren, vgl. Abschnitt A.2) führt in diesem Fall zum Überlauf des Rings und somit zum Verwerfen der `RST`-Antworten.

Aufgrund dessen wird der *Zero-Copy*-Modus für die Evaluationsszenarien (Abschnitt 4.2.2) ausschließlich ohne das Senden von `RST`-Paketen getestet.

6.4.2 Paketrate

Die Durchsatzrate ist durch die genutzten Netzwerkkarten sowie durch das *LAN*-Kabel auf das Limit einer Gigabit-Verbindung beschränkt. Die Tests zur maximalen Durchsatzrate können deshalb nur eingeschränkt durchgeführt werden. Das theoretische Limit einer

[3] Der Zugriff auf den Ring ist durch einen *Spinlock* geschützt und muss bei jedem Zugriff ausgehandelt werden. Wollen mehrere Parteien gleichzeitig senden, müssen sie aufeinander warten.

Gigabit-Leitung liegt nach *IEEE* 802.3 [64], bei einer Paketgröße von 64 Byte plus 20 Byte *Overhead*, bei 1,488 Millionen *PPS*. Deshalb wird der Fokus dort verstärkt auf die gemessene Ressourcenauslastung gelegt.

Kapitel 7: Evaluation und Ausblick

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

7.1 Darstellung und Reproduzierbarkeit der Messergebnisse

Die Messergebnisse wurden mittels der in Abschnitt 6.2.1 beschriebenen Skripte aufbereitet und stehen im Anhang zur Verfügung. Im Folgenden wird nur auf die ausschlaggebenden Ergebnisse eingegangen. Für jede Messung liegen allerdings umfangreiche Daten sowie Diagramme und Tabellen im bereitgestellten GitHub Repository [55] zur Verfügung. Der Ablauf, inklusive Erhebung und Ausgaben der Tests, lässt sich dort unter `logs_benchmark_suite.txt` und `logs_dummy_receiver.txt` nachvollziehen. Mithilfe dieser Dateien und der `README.md`-Datei können die *Benchmarks* exakt reproduziert und nachvollzogen werden.

7.1.1 Ergebnisse der Evaluationstests: *Proof of Concept*

Die Auswertung der Ausgabedateien der Scanner für Evaluationstest 1 zeigt bei allen Varianten für den ersten Test die in Tabelle 7.1 gezeigten Ergebnisse. Außerdem ergab die Auswertung, dass exakt jede gerade IP-Adresse des genutzten IP-Raumes erfasst wurde und keine Duplikate oder *False Positives* auftraten [55].

Bei Evaluationstest 2 zeigten ebenfalls alle Variationen das gleiche Verhalten. Die Tabelle 7.2 steht also stellvertretend für alle Testabläufe. Die Ergebnisse für den Ziel-Knoten sind in den Dateien `capture_no_rst.pcap` und `capture_with_rst.pcap` [55] zu finden. Die Daten des Scanner-Knotens liegen in `xdpdump_no_rst.pcap` und `xdpdump_with_rst.pcap` [55] vor.

Pakettyp	**Erwartet**	**Gesendet**	**Empfangen**
		Scanner-Knoten	*Ziel-Knoten*
`SYN`-Pakete	4096	4096	4096
`RST`-Pakete	2048	2048	2048
		Ziel-Knoten	*Scanner-Knoten*
`SYN-ACK`-Pakete	2048	2048	2048

Tabelle 7.1: Validierung der Paketmengen in Evaluationstest 1

Pakettyp	**Gesendet**	**Empfangen**
Deaktivierte RST-Funktion		
	Scanner-Knoten	*Ziel-Knoten*
`SYN`-Pakete	4	4
`RST`-Pakete	0	0
	Ziel-Knoten	*Scanner-Knoten*
`SYN-ACK`-Pakete	4	4
Aktivierte RST-Funktion		
	Scanner-Knoten	*Ziel-Knoten*
`SYN`-Pakete	4	4
`RST`-Pakete	2	2
	Ziel-Knoten	*Scanner-Knoten*
`SYN-ACK`-Pakete	2	2

Tabelle 7.2: Vergleich der Paketflüsse mit und ohne RST-Logik in Evaluationstest 2

7.1.2 Ergebnisse Evaluationsszenario 1: Performanzgrenzen

Gemäß Abschnitt 6.2.1 wird in *Aktiv* (Zeitraum während Paketfluss besteht) und *Gesamt* (Gesamte Laufzeit des Programmes) unterschieden. *Netto* beschreibt dabei, dass die Werte von der Grundlast bereinigt wurden. Aus den Ergebnissen des Tests zum Evaluationstest 2[55] erschließen sich nach Evaluationsszenario_1/zusammenfassung_ergebnisse_final.csv und Evaluationsszenario_1/validierung_hits.csv die in Abschnitt 7.1.2 dargestellten Werte. Aufgrund von Einschränkungen durch die eigene *Blacklist* hat ZMap weniger IP-Adressen gescannt, weshalb es weniger Ergebnisse hervorbrachte. Außerdem ist zu beachten, dass sowohl Masscan als auch ZMap keine Option zur Vermeidung des Sendens von `RST`-Antworten bieten. Da Masscan die Pakete in dessen eigens implementierten *User-Space*-TCP-Stack erstellt und versendet, fließen diese auch in die *PPS*-Metrik ein. Die von ZMap verursachten `RST`-Antworten werden automatisch vom Kernel gesendet und nicht in den genutzten Kernel-Logs erfasst.

Scanner	**Effizienz**	**PPS**	**Netto (Aktiv)**		**Netto (Gesamt)**		**Erg.**
	[*PPS*/%]	(aktiv) [Mio]	CPU [%]	RAM [MB]	CPU [%]	RAM [MB]	[Mio]
SYN-Rust (`XDP`, *Zero-Copy*)	258513	1,48	5,7	190,8	4,7	160,7	13,42
SYN-Rust (`XDP`, Copy)	133989	1,23	9,2	237,7	7,7	203,7	13,42
SYN-Rust (`XDP`, *Generic Mode*)	118852	1,23	10,4	263,8	8,7	231,8	13,42
Masscan	77717	1,05	13,5	42,6	12,3	42,1	13,42
SYN-Rust (`AF_PACKET`)	74689	1,06	14,2	265,6	12,1	240,9	13,42
ZMap	64018	1,35	21,1	13,0	17,8	12,4	10,07

Tabelle 7.3: Vergleich der *Performance*-Metriken

Zur Berechnung der in Abb. A.3 gezeigten Werte zur Effizienz der Scanner während der aktiven Phase, wurde der Durchsatz pro CPU-Prozent in jedem Durchlauf berechnet und daraus anschließend der Durchschnittswert gebildet.

7.1.3 Ergebnisse Evaluationsszenario 2: Reales Szenario

Die Ergebnisse mancher Variationen des SYN-Rust in Abschnitt 7.1.3 weichen bezüglich der *PPS*-Metrik von der Durchsatzlimitierung (500.000 *PPS*) ab. Dies ist in diesem Szenario explizit erwünscht. Die Daten der Tabelleneinträge sind in Evaluationsszenario_2/zusammenfassung_ergebnisse_final.csv und Evaluationsszenario_2/validierung_hits.csv[55] zu finden. Für ZMap gilt bezüglich der `RST`-Antworten das gleiche wie auch in Abschnitt 7.1.2.

Scanner	**Effizienz**	**PPS**	**Netto (Aktiv)**		**Netto (Gesamt)**		**Erg.**
	[*PPS*/%]	(aktiv) [Mio]	CPU [%]	RAM [MB]	CPU [%]	RAM [MB]	[Mio]
SYN-Rust (`XDP`, *Zero-Copy*, kein `RST`)	184353	0,51	2,8	82,2	1,8	62,0	1,64
SYN-Rust (`XDP`, Copy)	106118	0,60	5,7	77,0	3,6	57,7	1,64
SYN-Rust (`XDP`, *Generic Mode*)	103087	0,60	5,9	96,9	3,7	74,7	1,64
SYN-Rust (`AF_PACKET`)	99871	0,61	6,1	112,2	3,8	88,3	1,64
Masscan (ohne Deduplizierung, `RST` automatisch)	75534	0,50	6,6	34,8	4,8	31,4	1,68
ZMap (`RST` automatisch)	13272	0,49	37,2	62,3	26,5	58,3	1,68

Tabelle 7.4: Vergleich der *Performance*-Metriken (unter Berücksichtigung der `RST`-Pakete)

7.2 Diskussion der Ergebnisse

Nachfolgend werden die Messergebnisse interpretiert und in den Kontext der Forschungsziele gesetzt. Die Diskussion gliedert sich in die funktionale Validierung des *Proof of Concept* sowie die detaillierte Analyse der *Performance*-Effizienz im Vergleich zu bestehenden Lösungen.

7.2.1 *Proof of Concept*

Die Ergebnisse zu den Evaluationstests zeigen, dass der implementierte Scanner in allen Varianten die erwartete Verhaltensweise umsetzt. Entsprechend den Ergebnissen zu Evaluationstest 1 wurden alle Pakete gesendet und exakt die erwarteten Pakete empfangen. Es gab keinen Paketverlust oder anderweitiger Fehler.

Dabei ist anzumerken, dass in `log_dummy_receiver.txt` zu sehen ist, dass mehr als die geplante Menge an Paketen empfangen wurde, diese aber keine korrekten `SYN`-Pakete darstellen. Dies ist gewollt und passiert, da der Scanner in den Modi, in welchen diese Ausgabe zu beobachten ist, am Ende des Scans ein paar ungültige Pakete über den *Socket* verschickt. Bei vorherigen Tests wurden manche Pakete nicht mehr versendet, weshalb diese Maßnahme notwendig ist, um sicherzustellen, dass alle regulären Pakete versendet werden.

Die Ergebnisse zu Evaluationstest 2 zeigen, dass die `SYN`-Pakete korrekt gebaut werden, da Netcat und Wireshark sie als gültig anerkennen. Auch die `RST`-Pakete werden korrekt gebaut, was daran zu erkennen ist, dass der Netcat-Server nach dem Erhalt dieser Pakete aufhört, weitere `SYN-ACK`-Antworten zu senden, was im Beispiel ohne `RST`-Antwort nicht passiert.

7.2.2 *Performance*-Effizienz

In diesem Abschnitt erfolgt die Auswertung der erhobenen Systemmetriken. Der Fokus liegt auf der Gegenüberstellung von Durchsatz und Ressourceneffizienz unter Volllast sowie in realitätsnahen Szenarien, um die Vorteile der genutzten Techniken zu quantifizieren.

Evaluationsszenario 1

Gemäß der Ergebnisse des Evaluationsszenario 1 ist eine deutliche Steigerung der *Performance*-Effizienz gegenüber den bewährten Vergleichsobjekten ZMap und Masscan zu erkennen. Dies unterstreicht die Effizienz der genutzten *Kernel-Bypass*-Techniken `AF_XDP` und `eBPF`. Besonders im *Zero-Copy*-Modus sind deutliche Effizienzgewinne in Form einer Vervielfachung der Pakete pro CPU um den Faktor 3 und mehr im Vergleich zu Masscan sowie um den Faktor 4 und mehr im Vergleich zu ZMap zu erkennen. Doch auch die Konfigurationen des Rust Scanners, welche den *Copy Mode* des `AF_XDP`-*Sockets* nutzen, zeigen eine deutliche Effizienzsteigerung zu den externen Vergleichsobjekten. Die `AF_PACKET`-Konfiguration ist bezüglich der Effizienz und Gesamtlast der CPU mit Masscan vergleichbar. Das ist ein sehr interessantes Ergebnis, da Masscan einen eigenen TCP-Stack implementiert, um den Kernel zu umgehen und durch die Nutzung von *Memory Mapping* (`mmap`) einen gemeinsamen Speicherbereich zwischen *User-Space* und *Kernel-Space* implementiert. Die SYN-Rust Variante implementiert keine dieser Optimierungen. Allerdings implementieren alle SYN-Rust Scanner im Gegensatz zu den Vergleichsobjekten keine *Busy Loop*[1] beim Senden, sondern nutzen einen linearen *Backoff*[2], welcher deutlich ressourceneffizienter ist.

Bezüglich des absoluten Durchsatzes erreichte lediglich der *Zero-Copy*-Modus das physikalische Limit der Gigabit-Verbindung. Die Varianten im *Copy*- sowie *Generic-Mode* stagnierten hingegen reproduzierbar bei 1,23 *Mpps*[3]. Da die CPU-Auslastung nicht der limitierende Faktor war, deutet es darauf hin, dass dieser Engpass auf die Latenz der Kopiervorgänge zwischen *User-Space* und *Kernel-Space* sowie auf Limitierungen des verwendeten `igb`-Treibers zurückzuführen ist. Da ZMap, welches ebenso wie Masscan *Memory Mapping* nutzt, müssen keine Kopien angelegt werden, um Pakete in den Kernel zu übergeben, was der Latenz zugutekommt. Der Unterschied zwischen dem Durchsatz von ZMap und Masscan ist darauf zurückzuführen, dass Masscan nur mit maximal einem Thread sendet, während ZMap bei hoher Last mehrere nutzt.

Die Vergleichsobjekte zeigen bezüglich des RAM-Verbrauchs bessere Ergebnisse als die Rust-Implementierung (siehe Abb. A.5). ZMap und Masscan zeigen nahezu keinen RAM-Verbrauch, während die Rust-Implementierungen eine moderate RAM-Nutzung haben, die im Zuge heutiger Speicherkapazitäten aber als gering einzustufen ist.

[1] Es wird dauerhaft ohne Verzögerung Kapazität abgefragt.
[2] Die Wartezeit zwischen Abfragen steigt linear an, sollte mehrere Male nacheinander keine Kapazität zur Verfügung stehen.
[3] Millionen Pakete pro Sekunde

Anhand von Abb. A.4, welche die Verteilung der CPU-Gesamtlast auf die verschiedenen Bereiche *User-Space*, *Kernel-Space* und *SoftIRQ* verbildlicht, lässt sich gut erkennen, dass ZMap und SYN-Rust (`AF_PACKET`), welche die gleiche Adressfamilie nutzen, einen ähnlichen Verbrauch im *Kernel-Space* haben und Scanner, die `XDP` nutzen, einen deutlich niedrigeren. Dies verdeutlicht die Effizienz der *Kernel-Bypass*-Technik `AF_XDP`. Die Scanner, die `XDP` im *Native Mode* nutzen, zeigen einheitlich eine verminderte *SoftIRQ*-Auslastung, da Schritte wie das Erstellen eines `sk_buffs` vermieden werden. An der *User-Space*-Auslastung ist zu erkennen, dass die Maßnahmen zur Effizienzsteigerung (vgl. Kapitel 5) wirkungsvoll sind. Auch die fehlende Notwendigkeit, empfangene Pakete im *User-Space* parsen zu müssen, da es bereits im `eBPF`-Programm getan wird, spielt dem zu.

Um die Ergebnisse in das korrekte Verhältnis zu setzen, ist es wichtig, die Unterschiede zwischen der Funktionsweise aufzuzeigen. Im Gegensatz zu ZMap und Masscan hat SYN-Rust die zu scannenden IP-Adressen nicht randomisiert. Dies erfordert für jede Ziel-IP-Adresse eine einmalige Berechnung. ZMap nutzt beispielsweise eine zyklische multiplikative Gruppe über einem endlichen Körper, welche eine Multiplikation und eine Modulo-Operation erfordert.

Evaluationsszenario 2

In Evaluationsszenario 2 schlossen die SYN-Rust Varianten, die `XDP` nutzen, schlechter ab als zuvor. Dies ist einerseits den aktivierten *Features* wie der Duplikationserkennung und dem Senden von `RST`-Antworten zuzuschreiben. Andererseits könnte es auch bedeuten, dass diese bei einer niedrigeren Durchsatzlimitierung die Ressourcen etwas schlechter verwalten. Trotzdem sind die Ergebnisse sehr gut und die RAM-Auslastung skaliert den Ergebnissen zufolge gut bei gedrosselten Raten. Interessant ist, dass die `AF_PACKET`-Konfiguration bezüglich der Effizienz hier deutlich besser als im Performancetest abschneidet. Dies ist mit einer verminderten *Lock Contention* zu erklären, da in diesem Szenario nur einer statt zwei *Sender-Threads* benötigt wurde. Auch interessant ist, dass ZMap hier deutlich schlechter abschneidet und mehr als dreizehnmal mehr CPU bei nur etwas geringerem RAM-Verbrauch als die *Zero-Copy*-Lösung aufzeigt. Selbst die `AF_PACKET`-Lösung, welche die ineffizienteste der SYN-Rust Konfigurationen ist, verbraucht sechsmal weniger CPU.

Die Ergebnisse der Effizienz von ZMap und Masscan sind in diesem Fall nur bedingt mit den restlichen (bis auf *Zero-Copy*) vergleichbar, da SYN-Rust eine andere Designphilosophie verfolgt und die gesendeten `RST`-Pakete nicht in das Durchsatzlimit mit einbezieht. Außerdem gibt es bei Masscan keine Option zur Deduplizierung der Antworten, was die Vergleichbarkeit weiter schmälert.

In Abschnitt 7.1.3 ist zu sehen, dass die Rust Scanner mit rund 1,64 Millionen Ergebnissen rund 2,4 % weniger Ergebnisse als die erwartete Menge (rund 1,68 Millionen[4]) erkannt haben. Laut der Datei Evaluationsszenario_2/ethtool.csv[55] weicht die Gesamtanzahl der versendeten Pakete (inklusive `RST`-Pakete) sowohl beim Zähler der Netzwerkkarte, als auch

[4] 2^{28} (IP-Adressraum) $\cdot\, 0,2$ (Antwortrate) $\cdot\, 2$ (Ports) $\approx 1,68$ Mio.

beim Zähler von `procfs`, um rund 2,3 % vom erwarteten Wert (10.066.330) ab. Aufgrund dieser Informationen und da dieses Problem in keinem anderen der Tests auftrat, ist es wahrscheinlich, dass es sich um einen Bug im *User-Space*-Programm handelt, welcher vermutlich bei der Durchsatzlimitierung auftritt und möglicherweise im Zusammenhang mit der Konfiguration mehrerer Ziel-Ports steht.

7.2.3 Abgleich mit den Anforderungen

Abschließend erfolgt in Tabelle 7.5 eine Bewertung der in Abschnitt 4.1 definierten Anforderungen. Die Überprüfung basiert auf den Ergebnissen der dynamischen Tests (Kapitel 6) sowie der statischen Inspektion der Implementierung (Kapitel 5).

7.2.4 Wirtschaftliche und betriebliche Implikationen

Die in dieser Arbeit erarbeiteten Ergebnisse zeigen, dass der Einsatz von Rust und modernen Kernel-Techniken eine überaus sinnvolle Alternative für bisherige Ansätze darstellt. Dies impliziert auch signifikante ökonomische Vorteile für Unternehmen und Organisationen, die Internet-weite Messungen oder Sicherheitsanalysen durchführen. Darunter fallen:

- **Risikominimierung durch Sicherheitsgarantien:** Die in Abschnitt 2.4 beschriebenen Speichersicherheitsgarantien eliminieren ganze Fehlerklassen, die in C-basierten Systemen häufig zu Sicherheitslücken führen (vgl. Abschnitt 3.2). Dies minimiert das Risiko teurer Sicherheitsvorfälle und ungeplanter Ausfallzeiten. Dieser präventive Ansatz deckt sich mit den Empfehlungen des *National Institute of Standards and Technology* (NIST), welches die Integration von Sicherheitsanforderungen in den gesamten Systemlebenszyklus (*Security-by-Design*) als kosteneffektivste Methode zur Umsetzung von Schutzstrategien identifiziert [65, S 66]. Für Unternehmen resultiert dies in einer erhöhten Betriebsstabilität, dem Schutz der Reputation sowie der Einhaltung relevanter *Compliance*-Vorgaben.
- **Senkung der Betriebskosten und Nachhaltigkeit:** Durch die in Abschnitt 7.1.2 nachgewiesene massive Steigerung der *Performance*-Effizienz lassen sich gleichbleibende Scan-Leistungen mit signifikant geringerem Hardwareaufwand realisieren. Dies ermöglicht eine direkte Reduktion der operativen Ausgaben, da preiswertere Server oder kleinere Cloud-Instanzen für dieselbe Arbeitslast ausreichen. Gleichzeitig wird der Energiebedarf reduziert, was in Anbetracht steigender Energiekosten einen weiteren monetären Vorteil darstellt.
- **Senkung der Wartungskosten und Investitionsschutz:** Der Verzicht auf proprietäre Kerneltreiber zugunsten von *Mainline-Kernel*-Techniken (vgl. Abschnitt 3.2) minimiert externe Abhängigkeiten und verhindert wartungsintensive Anpassungen bei Betriebssystem-Updates. Dies führt zu einer nachhaltigen Senkung der *Total Cost*

Anforderung	**Status**	**Nachweis / Anmerkung**
Funktionale Anforderungen (siehe Abschnitt 4.1.1)		
Konstruktion valider Pakete	Erfüllt	Durch Abschnitt 7.1.1 und Netcat-Validierung (siehe Tabelle 6.1) bestätigt
Senden von Paketen	Erfüllt	Durch Abschnitt 7.1.1 nachgewiesen
Empfang von Paketen	Erfüllt	Erfolgreicher Empfang in Abschnitt 7.1.1 nachgewiesen
Zustandsloses Scanning	Erfüllt	Durch den Aufbau der logischen Komponenten erfüllt (Abschnitt 5.1.1)
Validierung von Antworten	Teilw. erfüllt	`SYN-ACKs` Abschnitt 7.1.1 korrekt erkannt; Simulation mit falschen Paketen nicht getestet
Schließen der Verbindung	Erfüllt	Funktional in Abschnitt 7.1.1 bestätigt; Paketverluste im *Zero-Copy*-Modus unter Volllast durch Hardwarelimitierung
Endausgabe	Erfüllt	Validiert durch *Parsing*-Skripte (Abschnitt 7.1.1)
Durchsatzlimitierung	Teilw. erfüllt	Abweichung der Limitierung um weniger als 3 % in Abschnitt 7.1.3 nachgewiesen; Gesamtzahl der Pakete hat sich allerdings um circa 2,3–2,4 % verringert
Eingabeschnittstelle	Erfüllt	Implementierung des *Standard Input Parser* (Abschnitt 5.2.5)
Nicht-funktionale Anforderungen (siehe Abschnitt 4.1.2)		
Maximierung Durchsatz	Erfüllt	1,48 *Mpps* (Abschnitt 7.1.2) bei einer Paketgröße von 60 Byte entspricht rund 95 % des theoretischen Limits einer Gigabit-Leitung
Asynchrone Architektur	Erfüllt	Umsetzung mittels `tokio`-Runtime (vgl. Kapitel 5)
Moderne Kernel-Mechanismen	Erfüllt	Nutzung von `XDP` und `eBPF` (vgl. Kapitel 5)
Speichersicherheit	Erfüllt	*eBPF-Verifier* und externe Bibliotheken[1] als Sicherheitsmaßnahme für `unsafe`-Operationen verwendet (vgl. Kapitel 5)
Minimale Ressourcennutzung	Erfüllt	Höchste *Performance*-Effizienz im Vergleich (siehe Abb. A.3) bei moderater RAM-Nutzung
Technische Einschränkung	Erfüllt	Keine proprietären Treiber verwendet (vgl. Kapitel 5)

[1] Diese kapseln `unsafe`-Blöcke intern häufig sicher [38], sodass Sicherheitsniveau erhalten bleibt.

Tabelle 7.5: Zusammenfassender Abgleich der Anforderungen

of Ownership und bietet Investitionsschutz durch die Nutzung eines zukunftssicheren Technik-Stacks.

7.3 Fazit

Diese Arbeit untersuchte das Potenzial der Programmiersprache Rust für die Entwicklung von Hochleistungs-Netzwerkscannern, ein Feld, das bisher primär von C-basierten Anwendungen dominiert wird. Ausgangspunkt war die Forschungsfrage, inwieweit ein in Rust implementierter Scanner hinsichtlich Durchsatz und Ressourceneffizienz mit etablierten Tools konkurrieren kann, ohne dabei auf die sprach-eigenen Sicherheitsgarantien zu verzichten.

Zur Beantwortung dieser Frage wurde der Prototyp SYN-Rust entwickelt, welcher eine asynchrone Architektur auf Basis der `tokio`-Runtime mit modernen Linux-Kernel-Schnittstellen wie `AF_XDP` und `eBPF` kombiniert. Die Evaluation demonstrierte, dass diese Kombination nicht nur funktional robust ist, sondern auch signifikante *Performance*-Vorteile bieten kann.

Die in Kapitel 7 vorgestellten Messergebnisse belegen, dass der entwickelte Scanner im `AF_XDP` *Zero-Copy*-Modus eine deutlich höhere CPU-Effizienz aufweist als die etablierten Referenzwerkzeuge. Spezifisch konnte eine Steigerung der verarbeiteten Pakete pro CPU-Prozent um den Faktor drei gegenüber Masscan und den Faktor vier gegenüber ZMap nachgewiesen werden. Die theoretische Durchsatzgrenze der verwendeten Gigabit-Hardware konnte dabei problemlos ausgeschöpft werden. Selbst unter Nutzung der herkömmlichen `AF_PACKET`-Schnittstelle, welche den Kernel-Stack nicht vollständig umgeht, erzielte die Rust-Implementierung Ergebnisse, die mit den hochoptimierten C-Scannern vergleichbar waren.

Einschränkend muss angemerkt werden, dass die Hardwarelimitierung der Testumgebung eine Ermittlung der maximalen Durchsatzgrenze des *Zero-Copy*-Modus verhinderte. Zudem zeigten sich im gedrosselten realitätsnahen Szenario geringfügige Abweichungen in der Paketanzahl von ca. 2,4 %, welche auf Optimierungsbedarf in der *User-Space*-Logik hindeuten. Dennoch validieren die Ergebnisse, dass der moderat höhere Arbeitsspeicherbedarf der Rust-Anwendung durch die massiven Gewinne in der CPU-Effizienz gerechtfertigt ist.

Zusammenfassend lässt sich die Forschungsfrage positiv beantworten: Ein in Rust implementierter asynchroner `SYN`-Scanner stellt eine absolut tragfähige Alternative für den produktiven Einsatz dar. Die Arbeit zeigt, dass die Nutzung von Rust und seiner Sicherheitsgarantien nicht im Widerspruch zu hoher *Performance* steht. Vielmehr ermöglicht die Integration moderner *Kernel-Bypass*-Techniken wie `XDP` eine Effizienz, die klassische C-basierte Ansätze sogar übertreffen kann. Rust empfiehlt sich somit als zukunftssichere Technologie für die Entwicklung systemnaher Netzwerkanwendungen.

7.4 Ausblick

Die vorliegende Arbeit hat die Eignung von Rust für die Entwicklung hochperformanter `SYN`-Scanner unter Nutzung moderner Kernel-Mechanismen demonstriert. Aus den gewonnenen Erkenntnissen und den methodischen Grenzen dieser Untersuchung leiten sich diverse Anknüpfungspunkte für zukünftige Forschungsarbeiten ab:

- **Detaillierte Stabilitätsanalyse:** In Abschnitt 7.2.2 wurden unter den genutzten Konfigurationen Anomalien in der Funktionsweise beobachtet. Eine tiefergehende Analyse dieser Randfälle ist notwendig, um die Ursache zu isolieren und die Robustheit des Systems für den Dauerbetrieb sicherzustellen.
- **Evaluation der Scangenauigkeit:** Der Fokus dieser Arbeit lag primär auf der Maximierung der *Performance*-Effizienz. Für den produktiven Einsatz ist jedoch auch die Scangenauigkeit von Bedeutung. Eine weiterführende Untersuchung sollte analysieren, wie sich die hier genutzte *Kernel-Bypass*-Technik auf die Zuverlässigkeit der Paketerkennung unter variierenden Netzwerklasten auswirkt.
- **Hardware-Anpassung und -Skalierung:** Wie in Abschnitt 6.4 diskutiert, stellte der verwendete Netzwerkkartentreiber sowie die Hardwarebeschränkung auf 1 Gbit/s (Tabelle 6.2) einen Flaschenhals dar (vgl. Abschnitt 7.1.2). Zukünftige Evaluationen sollten den Scanner in 10 Gbit/s-Umgebungen oder noch schnelleren testen. Dabei wäre besonders interessant herauszufinden, bei welcher Durchsatzrate das Programm an seine Grenzen stößt und welche Faktoren letztendlich den Flaschenhals bilden.
- **Validierung im realen Anwendungsfall:** Da die Evaluation in einer kontrollierten Laborumgebung stattfand, steht ein Test in einem realen Szenario, wie dem Scan großer Teile des öffentlichen IPv4-Adressraums, noch aus. Dies würde Rückschlüsse auf das Verhalten des Scanners bei realer Latenz, Paketverlusten und Sicherheitsmechanismen (z. B. *Firewalls*) von Zielsystemen ermöglichen.
- **Erweiterung auf IPv6:** Diese Arbeit beschränkt sich auf den IPv4-Adressraum. Da der IPv6-Adressraum durch die zunehmende Verbreitung an Relevanz gewinnt [66], stellt die Anpassung des Scanners, sodass auch dieses Protokoll unterstützt wird, einen logischen nächsten Schritt dar. Hierbei wäre insbesondere zu untersuchen, wie sich die vergrößerten Adressstrukturen (128 Bit) auf die Speicherverwaltung im `eBPF`-Programm, der veränderte *Parsing*-Aufwand aufgrund von *Next Header*-Verkettungen und die damit verbundene Rechenlast für *Hashing*-Operationen auf den Durchsatz sowie die Effizienz im Kontext eines `SYN`-Scanners auswirken.
- **Adaption an weitere Scan-Methoden:** Die in dieser Arbeit entwickelte Architektur zur Kernelumgehung ist prinzipiell protokollunabhängig. Zukünftige Forschungen könnten untersuchen, wie effizient sich das System auf alternative verbindungslose Szenarien (z. B. UDP-Scans) oder andere TCP-Scan-Techniken (z. B. `ACK`- oder `FIN`-Scans) übertragen lässt. Dies würde validieren, ob die *High-Performance*-Vorteile von

Rust und *Kernel-Bypass*-Ansatz auch bei veränderter logischer Komplexität Bestand haben.

Anhang A: Anhang

A.1 Ergänzende Diagramme

A.2 Netzwerkkarten-Konfiguration (Ethtool)

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

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

A.3 Scanergebnisse Evaluationsszenario 1

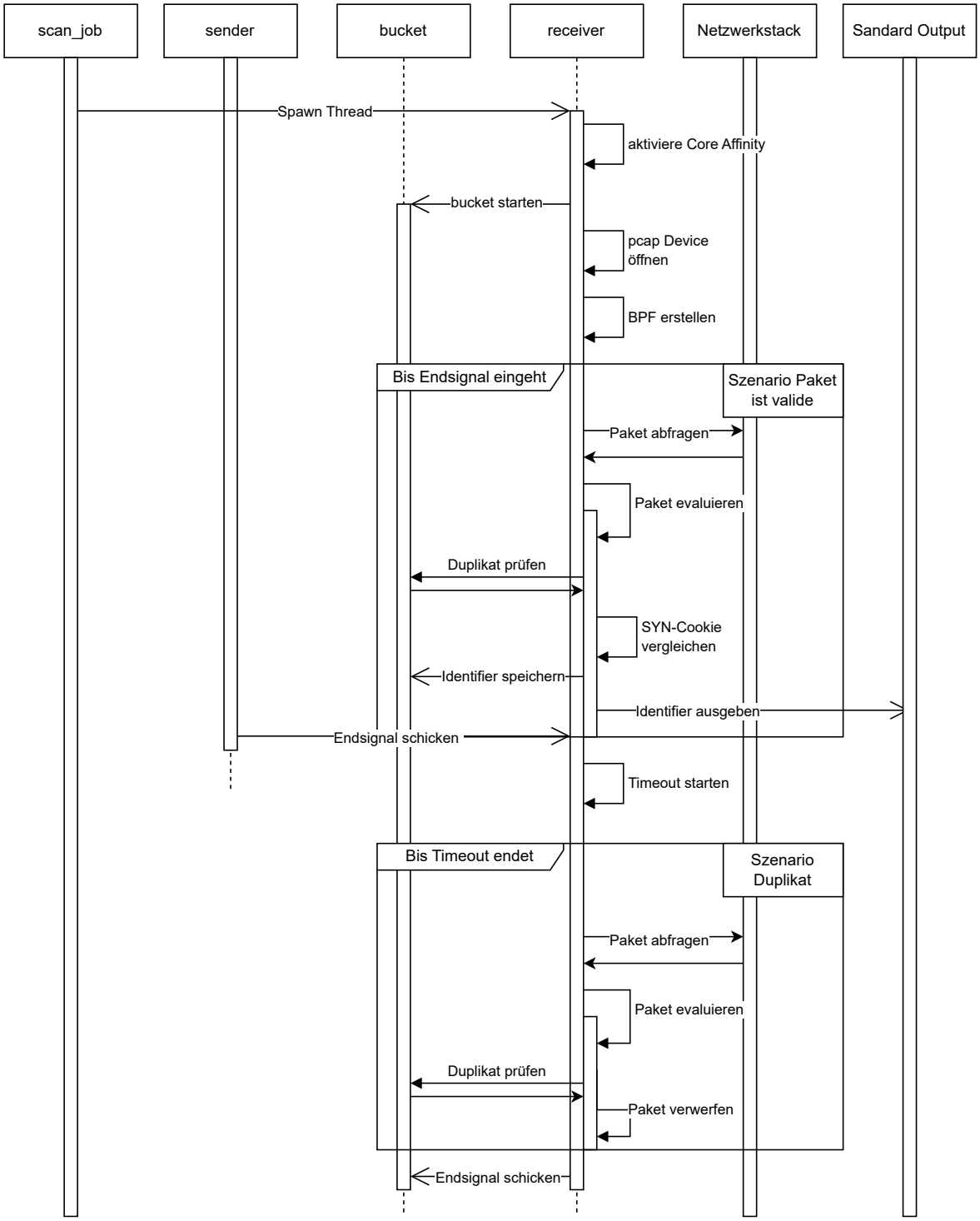

Abbildung A.1: Funktionsweise der alten `receiver.rs`-Datei (vereinfacht)

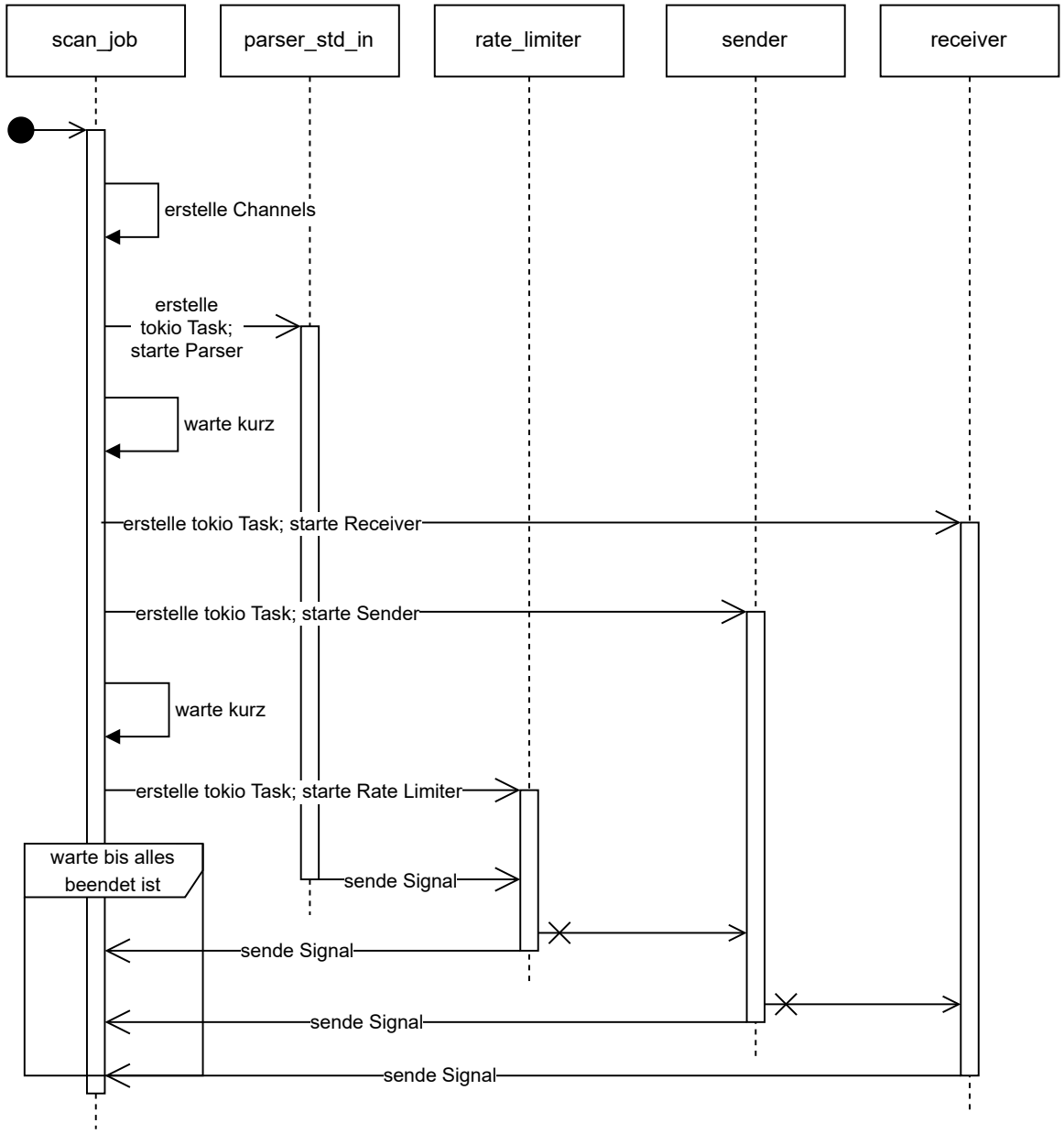

Abbildung A.2: Funktionsweise der `scan_job.rs`-Datei (vereinfacht)

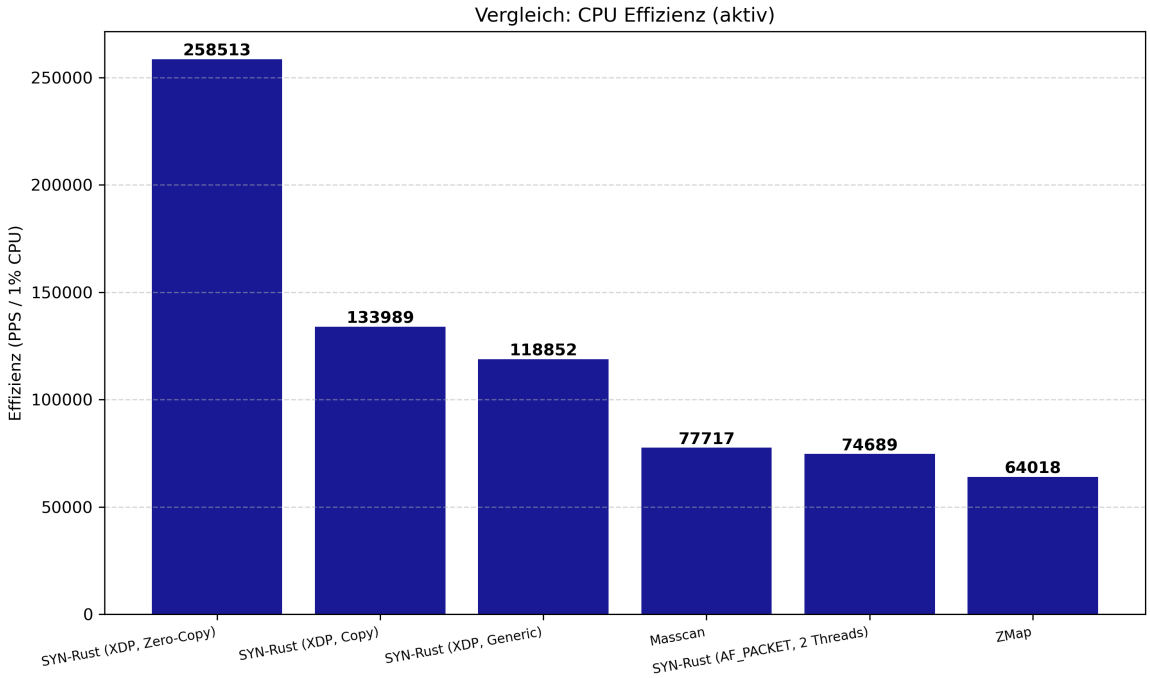

Abbildung A.3: Effizienz der SYN-Scanner im Benchmark (aktiv)

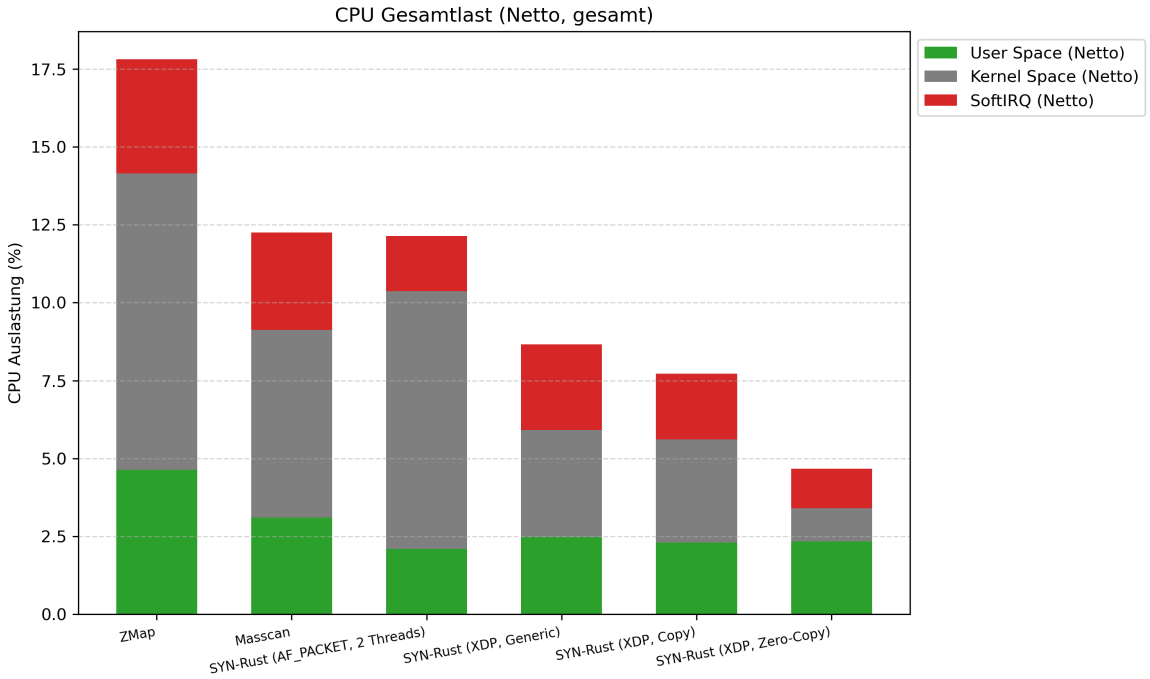

Abbildung A.4: CPU-Auslastung der SYN-Scanner im Benchmark (gesamt)

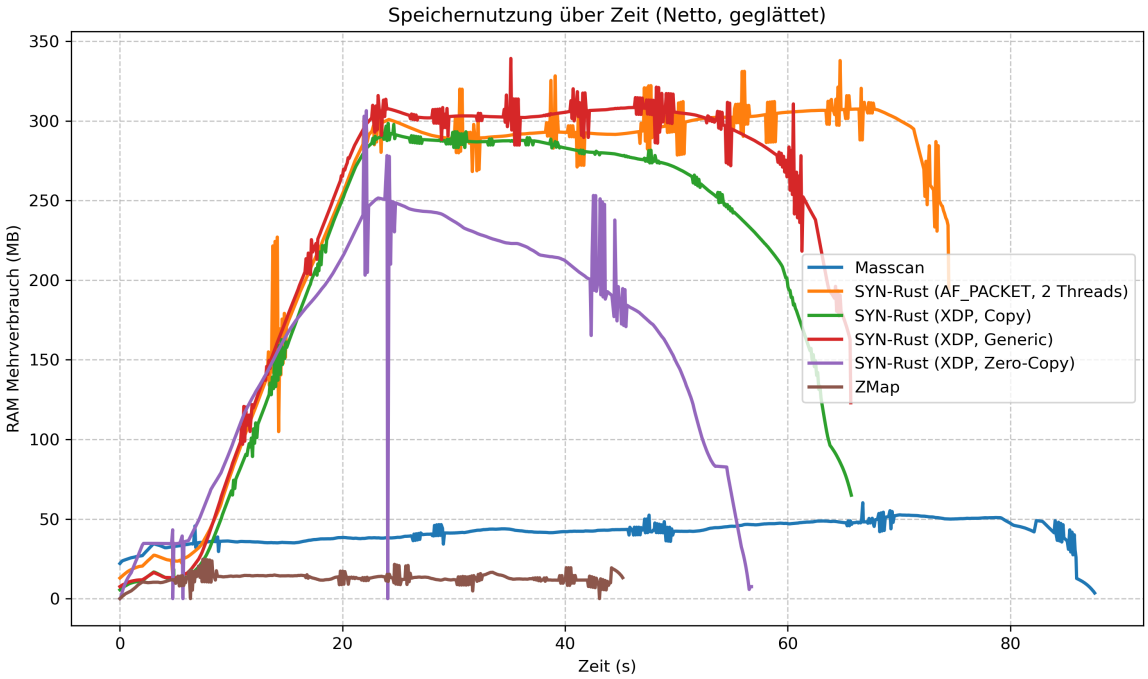

Abbildung A.5: Effizienz der SYN-Scanner im Benchmark (gesamt)

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

KI-Verzeichnis

KI-Tool	Teil der Arbeit	Verwendungszweck
GitHub Copilot	Gesamtes Dokument	Formattierung der Fremdwörter, programmierspezifischer Wörter, sowie sprachliche Nachbearbeitung von Textpassagen
GitHub Copilot	Kapitel 5	Unterstützung beim Debugging im Implementierungsprozess
Google Gemini	Abb. 2.2, Abb. 2.1, Abb. 5.6	Unterstützung bei der Erstellung der Grafiken
Google Gemini	Kapitel 6	Initiale Erstellung der Benchmarking- und Auswertungs-Skripte
Google Gemini	Gleichung (5.1), Gleichung (5.2)	Initiale Erstellung der Gleichungen
Consensus	Gesamtes Dokument	Unterstützung bei der Literaturrecherche

Literaturverzeichnis

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

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

[3] R. D. Graham, *robertdavidgraham/masscan*, C, Accessed: 2026-02-03 11:30, Jan. 2026. Adresse: https://github.com/robertdavidgraham/masscan

[4] Z. Durumeric, E. Wustrow und J. A. Halderman, „ZMap: Fast Internet-wide Scanning and Its Security Applications," in *22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C.: USENIX Association, Aug. 2013, S. 605–620, ISBN: 978-1-931971-03-4. Adresse: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric

[5] 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/

[6] 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

[7] X. Li, *idealeer/xmap*, C, Accessed: 2026-02-02 14:15, 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] S. Peta, „C Programming Language - Still Ruling the World," en, *International Journal of Science and Research (IJSR)*, Jg. 11, Nr. 4, S. 548–552, Apr. 2022, ISSN: 23197064. DOI: 10.21275/SR22403142926

[10] A. Al-Boghdady, K. Wassif und M. El-Ramly, „The Presence, Trends, and Causes of Security Vulnerabilities in Operating Systems of IoT's Low-End Devices," en, *Sensors*, Jg. 21, Nr. 7, März 2021, Company: Multidisciplinary Digital Publishing Institute Distributor: Multidisciplinary Digital Publishing Institute Institution: Multidisciplinary Digital Publishing Institute Label: Multidisciplinary Digital Publishing Institute publisher: publisher, ISSN: 1424-8220. DOI: 10.3390/s21072329 Adresse: https://www.mdpi.com/1424-8220/21/7/2329

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

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

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

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

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

[16] Accessed: 2026-02-11 17:40. Adresse: https://nmap.org/book/port-scanning.html#port-scanning-port-intro

[17] en, Accessed: 2026-02-10 16:30. Adresse: https://www.hanser-elibrary.com/doi/epdf/10.3139/9783446484856

[18] IANA, *Service Name and Transport Protocol Port Number Registry*, Accessed: 2026-02-01 16:45. Adresse: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

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

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

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

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

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

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

[25] Accessed: 2026-02-07 19:10. Adresse: https://docs.kernel.org/security/siphash.html

[26] 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

[27] *socket(2) - Linux manual page*, man7.org, Accessed: 2026-02-05 10:20. Adresse: https://man7.org/linux/man-pages/man2/socket.2.html

[28] *raw(7) - Linux manual page*, man7.org, Accessed: 2026-02-05 10:25. Adresse: https://man7.org/linux/man-pages/man7/raw.7.html

[29] *address_families(7) - Linux manual page*, man7.org, Accessed: 2026-02-05 10:30. Adresse: https://man7.org/linux/man-pages/man7/address_families.7.html

[30] Accessed: 2026-02-12 13:55. Adresse: https://man7.org/linux/man-pages/man7/packet.7.html

[31] 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

[32] 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

[33] 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

[34] en, Accessed: 2026-02-08 12:15. Adresse: https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_RINGBUF/

[35] 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/

[36] 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

[37] 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

[38] en, Accessed: 2026-02-06 13:10. DOI: 10.1145/3158154 Adresse: https://dl.acm.org/doi/epdf/10.1145/3158154

[39] 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/

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

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

[42] en, Accessed: 2026-02-04 15:00. Adresse: https://go.dev/blog/waza-talk

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

[44] 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

[45] 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

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

[47] 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

[48] 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/

[49] 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

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

[51] 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/

[52] 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

[53] *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*, Accessed: 2026-02-11 19:38. Adresse: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-2:v1:en

[54] R. Love, *Linux kernel development: a thorough guide to the design and implementation of the Linux kernel* (Developer's library), eng, 3. ed., 3. printing. Upper Saddle River, NJ: Addison-Wesley, 2011, ISBN: 978-0-672-32946-3.

[55] L. A. Dubhorn, *Bachelor-Thesis-Code*, Zugriff: 2026-02-14, Feb. 2026. Adresse: https://github.com/F4c3hugg3r/Bachelor_Doku

[56] Accessed: 2026-02-07 18:45. Adresse: https://docs.rs/tokio/latest/tokio/task/

[57] C, Accessed: 2026-02-12 15:10, Feb. 2026. Adresse: https://github.com/the-tcpdump-group/libpcap

[58] en, Accessed: 2026-02-08 12:00. Adresse: https://docs.ebpf.io/linux/map-type/BPF_MAP_TYPE_PERCPU_ARRAY/

[59] L. Rice, *Learning eBPF: programming the linux kernel for enhanced observability, networking, and security*, en, First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2023, ISBN: 978-1-0981-3512-6.

[60] Accessed: 2026-02-10 10:05. Adresse: https://www.kernel.org/doc/Documentation/filesystems/proc.txt

[61] Accessed: 2026-02-09 11:50. Adresse: https://github.com/torvalds/linux/commit/9cbc948b5a20c9c054d9631099c0426c16da546b

[62] Accessed: 2026-02-13 11:20. Adresse: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[63] Intel Corporation, *Intel® Ethernet Controller I350 Datasheet*, Rev. 2.6, 2017.

[64] Accessed: 2026-02-09 14:20. DOI: 10.1109/IEEESTD.2022.9844436 Adresse: https://ieeexplore.ieee.org/document/9844436/

[65] J. T. F. T. Initiative, *Risk management framework for information systems and organizations: a system life cycle approach for security and privacy*, en. Gaithersburg, MD, Dez. 2018, NIST SP 800–37r2. DOI: 10.6028/NIST.SP.800-37r2 Adresse: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-37r2.pdf

[66] S. T. Valapu und J. Heidemann, „Towards a Non-Binary View of IPv6 Adoption," in *Proceedings of the 2025 ACM Internet Measurement Conference*, arXiv:2507.11678 [cs], Okt. 2025, S. 727–745. DOI: 10.1145/3730567.3764467 Adresse: http://arxiv.org/abs/2507.11678

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 14.02.2025

Lennard Alexander Dubhorn