Hochschule für Technik und Wirtschaft Berlin
Fachbereich 1 - Energie und Information
Informations- und Kommunikationstechnik

Automatisierte Netzwerk-Belastungssimulation mit einem P4-programmierbaren Ethernet-Switch

Masterarbeit

Master of Engineering (M. Eng.)

von

Pascal Bast

Matrikelnummer: 558899

Erstprüfer
Prof. Dr. Thomas Scheffler

Zweitprüfer
Prof. Dr. Markus Nölle

12.08.2024

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Berlin, den 12.08.2024

Pascal Bast

Kurzfassung

In dieser Abschlussarbeit wurde ein System zur Simulation von Netzwerkbelastungen und DDoS-Angriffen auf Basis der Intel Tofino-Plattform entwickelt. Ziel war es, Netzwerkverkehr zu generieren und zu analysieren, um die Reaktion von eines Testsystems auf hohe Lasten zu testen und Schwachstellen zu identifizieren. Der entwickelte Prototyp wurde in Python und P4 umgesetzt und auf der im Netzwerklabor der HTW Berlin verfügbaren Hardware erfolgreich getestet.

Abstract

This thesis presents the development of a system for simulating network load and DDoS attacks using the Intel Tofino platform. The goal was to generate and analyze network traffic to test the response of a test system under high loads and identify vulnerabilities. The developed prototype was implemented in Python and P4 and successfully tested on the hardware available at the HTW Berlin network lab.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Ausgangslage

Mit der zunehmenden Digitalisierung und Automatisierung in sämtlichen Lebensbereichen steigt auch die Bedeutung der Informationssicherheit von Netzwerken kontinuierlich an. Der rasante Fortschritt im Bereich IT und die damit einhergehende Entwicklung neuer, komplexer Technologien und Anwendungen führen unweigerlich zu einer stetigen Erweiterung der Angriffsfläche bestehender Infrastrukturen. Infrastrukturen, die für unser tägliches Leben mittlerweile unverzichtbar geworden sind, da selbst kritische Systeme wie Energieversorgung, Verkehr oder Gesundheitswesen auf IT-Systemen basieren.

Immer mehr dieser Systeme sind heute miteinander vernetzt und tauschen in der Gesamtheit eine enorme Menge an Daten aus. Es ist daher nicht verwunderlich, dass die Anforderungen an Performance und Verfügbarkeit von Netzwerkinfrastrukturen ebenfalls stetig wachsen.

Besonders innerhalb der letzten zwei Jahre ist ein deutlicher Anstieg in Vielfalt und Menge von Cyberangriffen zu beobachten. Die aktuelle Bedrohungslage setzt sich hauptsächlich aus folgenden Angriffstypen zusammen:

- Ransomware
- Malware
- Social Engineering
- Angriffe auf Daten (Integrität)
- Angriffe auf Verfügbarkeit (Denial of Service)
- Angriffe auf Verfügbarkeit (Internetverfügbarkeit)
- Informationsmanipulation und Störung
- Supply-Chain-Angriffe

Die meisten der gemeldeten Angriffe (über 50 %) lassen sich heute auf Ransomware (31,32 %) und Distributed-Denial-of-Service (21,4 %) zurückführen. DDoS-Angriffe, welche hauptsächlich auf die Verfügbarkeit von Systemen abzielen, werden zudem immer größer und komplexer. Besonders besorgniserregend ist dabei, dass sie in den letzten Jahren für Angreifer paradoxerweise immer einfacher und kostengünstiger durchzuführen sind. Außerdem beschränken sich diese Angriffe mittlerweile nicht mehr nur noch auf Webservices. Immer häufiger gelangen auch Mobilfunknetze oder IoT-Systeme in das Visier von Angreifern. [Eur23]

Der starke Anstieg von DDoS-Angriffen seit 2023 lässt sich vermutlich auf die wachsende Popularität von Hacktivism sowie auf aktuelle geopolitische Spannungen (Russland-Ukraine-Konflikt, Spannungen zwischen China und Taiwan etc.) zurückführen. [Eur23]

Aufgrund der dargelegten Trends ist es von zunehmender Bedeutung, dass Unternehmen und Organisationen ihre Netzwerkinfrastrukturen auf solche Angriffe vorbereiten und entsprechende Gegenmaßnahmen ergreifen. Dies ist jedoch oft sehr schwierig und erfordert einen Mix aus präventiven und reaktiven Strategien.

Von zentraler Bedeutung ist hierbei der Aufbau einer robusten und widerstandsfähigen Netzwerkinfrastruktur, deren Ressourcen auf Redundanz und Skalierbarkeit ausgelegt sind. Ein weiterer wichtiger Aspekt ist die Verfügbarkeit von genügend Bandbreite, welche optimalerweise den durchschnittlichen Bedarf bei Normalbetrieb weit übersteigt. Je mehr Bandbreite zur Verfügung steht, desto schwieriger und teurer wird es für Angreifer, die Infrastruktur zu überlasten. [Clo22]

Um die Robustheit und Widerstandsfähigkeit von Netzwerkinfrastrukturen zu testen und zu quantifizieren, werden verschiedene Methoden eingesetzt. Eine dieser Methoden ist die Durchführung von Netzwerk-Belastungssimulationen, um die Reaktion von Systemen auf hohe Lasten zu testen und Schwachstellen zu identifizieren, bevor sie von Angreifern ausgenutzt werden können. Zusätzlich können so entsprechende Reporting- und Alerting-Mechanismen getestet und optimiert werden.

In der Praxis existieren hauptsächlich folgende Werkzeuge und Services für die Durchführung von Netzwerk-Belastungssimulationen:

- **Network Traffic Simulatoren**: Software- oder Hardware-basierte Tools zur Erzeugung von Netzwerkverkehr für Stresstests von Netzwerken und Servern.
- **Booter Services**: DDoS-as-a-Service-Angebote, die es ermöglichen, DDoS-Angriffe gegen Ziele zu starten. Meistens illegale Angebote, die von Kriminellen betrieben werden. Traffic wird in der Regel über Botnetze oder leistungsstarke Server generiert. [Bri23; Clo24]
- **Cloud-basierte DDoS-Simulationen**: Viele Cloud-Anbieter arbeiten heute mit zertifizierten Partnern zusammen, um DDoS-Simulationen in der Cloud durchzuführen. Diese Dienste sind jedoch an sehr spezifische Bedingungen gebunden und sind zusätzlich auf Cloud-Services (Amazon Web Service, Microsoft Azure, Cloudflare) beschränkt. [Abd24; Ama24; Clo24]

Die meisten dieser Tools und Services sind jedoch entweder sehr teuer oder erfordern spezielle Hardware, die nicht immer verfügbar ist. Darüber hinaus sind viele dieser Tools und Services nicht in der Lage, die spezifischen Anforderungen von Unternehmen und Organisationen zu erfüllen. [Key24; Spi24]

Möglicherweise bieten neue Entwicklungen im Bereich des Software-Defined Networking (SDN) nun eine Lösung für einige dieser Probleme. Die Verfügbarkeit von programmierbaren Ethernet-Switches ermöglicht es, die Vorteile von Software- und Hardware-basierten Lösungen zu kombinieren und so womöglich eine flexiblere und kostengünstige Lösung für die Durchführung von Netzwerk-Belastungssimulationen zu schaffen.

1.2 Aufgabenstellung

Im Rahmen dieser Abschlussarbeit soll ein System für Belastungs- und Angriffssimulationen auf Netzwerken entwickelt werden. Die Implementierung erfolgt dabei auf einem programmierbaren Ethernet-Switch, der auf der Intel Tofino-Plattform [Int24] basiert.

Das System soll in der Lage sein, verschiedene Arten von Netzwerkverkehr zu generieren und zu analysieren, um die Reaktion von Netzwerken auf hohe Lasten zu testen und Schwachstellen zu identifizieren. Darüber hinaus soll die Software in der Lage sein, bestimmte DDoS-Angriffe zu simulieren und zu analysieren.

Ein Teil der Aufgabenstellung beschäftigt sich entsprechend mit der Identifikation der Hardware-Limitationen der Intel Tofino-Plattform im Hinblick auf die Generierung von Netzwerkverkehr. So soll untersucht werden, welche Arten von Netzwerkverkehr generiert werden können und welche Bandbreiten und Paketgrößen dabei erreicht werden können.

1.3 Umsetzung und Herangehensweise

Zu Beginn sollen die Grundlagen moderner Netzwerktechnologien in Hinblick auf programmierbare Netzwerksysteme und Software-Defined Networking (SDN) erarbeitet und erläutert werden. Dabei wird besonders auf P4 als Programmiersprache für Data-Plane-Applikationen sowie die Intel-Tofino-Architektur als valide Zielplattform für die Implementierung eingegangen. Zur allgemeinen Orientierung wird das Konzept von Netzwerk-Belastungssimulatoren erklärt, wobei die Unterschiede zwischen Software- und Hardware-basierten Lösungen herausgearbeitet werden. Zur Vertiefung werden existierende Implementierungen von Traffic-Generatoren vorgestellt und miteinander verglichen. Für den spezifischen Use-Case von DDoS-Angriffssimulationen wird auf die verschiedenen Arten von DDoS-Angriffen und deren Eigenschaften sowie Auswirkungen auf Netzwerke eingegangen.

Auf Basis dieser bestehenden Projekte wird im praktischen Teil der Arbeit ein Prototyp auf der im Netzwerklabor der HTW verfügbaren Hardware entwickelt und getestet. Die Software soll in der Lage sein, Tests weitestgehend automatisch zu starten und in Bezug auf die Funktionalität flexibel und erweiterbar zu sein. Die Implementierung erfolgt in Python, da es sich um eine weit verbreitete Programmiersprache handelt, die eine einfache Integration von Bibliotheken, Application Programming Interfaces (API) und bestehendem Tooling ermöglicht. Die Data-Plane-Applikationen werden in P4 entwickelt, um die umfangreichen Möglichkeiten der Intel Tofino-Plattform voll auszuschöpfen und so hohe Datenraten zu erreichen.

Nach der Entwicklung eines funktionierenden Prototypen werden einige Simulationsszenarien durchgeführt, um die Leistungsfähigkeit des Systems zu testen und anschaulich zu demonstrieren.

Im Schlussteil soll auf Schwierigkeiten bei der Entwicklung und identifizierte Limitationen eingegangen werden. Darüber hinaus werden mögliche Erweiterungen und Verbesserungen des Systems vorgestellt, die in zukünftigen Arbeiten umgesetzt werden könnten.

2 Grundlagen

Die folgenden Abschnitte geben einen Überblick über die relevanten Grundlagen und Technologien der Netzwerktechnik, die für die Implementierung eines Systems für automatisierte Netzwerk-Belastungssimulationen notwendig sind.

So werden im Folgenden entsprechende Konzepte und Technologien im Bereich des Software-Defined Networking (SDN) und Open Source Networking vorgestellt und erläutert. Zusätzlich werden notwendige Kenntnisse der IT-Sicherheit ermittelt, um die Anforderungen an ein solches System zu verstehen und effektiv umsetzen zu können. Voraussetzung für das korrekte Verständnis der folgenden Abschnitte ist ein grundlegendes Verständnis von Netzwerktechnologien und -protokollen, auf die im Rahmen dieser Arbeit aufgrund des begrenzten Umfangs nicht eingegangen werden kann.

2.1 Software Defined Networking

Wie bereits zuvor erwähnt, kommen für die Umsetzung dieser Arbeit moderne Netzwerktechnologien zum Einsatz. Diese Technologien stammen hauptsächlich aus dem Bereich der Open Source Networking Technologien, speziell aus der Kategorie der Programmable Networks. Die Gruppe der Programmable Networks Projects wurde ursprünglich von der Open Networking Foundation (ONF) ins Leben gerufen und umfasst eine Vielzahl von Projekten, die dem Ansatz von Software-Defined Networking (SDN) folgen [ONF24]. Mittlerweile sind sämtliche dieser Open Source Networking Projekte offiziell in die Linux Foundation (LF) übergegangen [ONF23].

Open Source Networking und Software-Defined Networking (SDN) ergänzen sich gegenseitig in der Zielsetzung, Netzwerke flexibler, programmierbarer, effizienter und vor allem offener zu gestalten. SDN beschäftigt sich dabei mit der Konzeption und Umsetzung moderner Netzwerkinfrastrukturen [Pet+21], während Open Source Networking die zugrundeliegende Softwarebasis bereitstellt, die diese Flexibilität und Programmierbarkeit unterstützt [Ope24].

2.1.1 Ziel und Motivation von SDN

Traditionelle Netzwerkgeräte, wie sie von Herstellern wie Cisco, Juniper und anderen weiterhin angeboten werden, sind geschlossene Plattformen und entsprechen Kombinationen von proprietärer Software und Hardware. Diese Geräte kommen mit einem bestimmten proprietären Betriebssystem (engl.: Operating System, OS), das genau auf die Hardware dieses Geräts zugeschnitten ist. Der User kann in diesem Fall das OS nicht ändern oder frei wählen, sondern ist auf Software und Support des Herstellers angewiesen [Ove24]. Dieser entwickelt seine eigene proprietäre Firmware und auch Netzwerkprotokolle wie beispielsweise Open

Shortest Path First (OSPF) oder das Border Gateway Protocol (BGP) [TFW21]. Obwohl die meisten Netzwerkprotokolle auf offenen Standards basieren, werden sie von jedem Hersteller einzeln und daher aller Wahrscheinlichkeit nach im Detail unterschiedlich implementiert [Pet+21] .

Um zu verstehen, warum wir für Netzwerke solche geschlossenen, Mainframe-ähnlichen Systeme verwenden, muss die Geschichte des Computermarktes betrachtet werden. Ursprünglich war es für Kunden üblich, vertikal integrierte Systeme zu kaufen, die von einem einzigen Hersteller entwickelt und für einen ganz bestimmten Zweck hergestellt wurden (z.B. Finanzwesen, wo heute noch Mainframes zum Einsatz kommen [z/O23]). Mit der Entwicklung von Mikroprozessoren und frei verfügbaren Betriebssystemen wurde der Computermarkt allmählich in einen horizontalen Markt transformiert, in dem Kunden die Möglichkeit hatten, Hardware und Software von verschiedenen Herstellern zu kombinieren [Pet+21].

Ein wichtiger Schritt für die Umsetzung dieses Marktes war die Entwicklung des Basic Input/Output System (BIOS). Als standardisierte Hardware-Abstraktionsschicht ermöglicht das BIOS eines Computers, dass unterschiedliche Betriebssysteme auf der gleichen Hardwareplattform gestartet werden können [Pet+21].

Das Ziel von SDN und Open Source Networking ist es, die gleiche Transformation im Netzwerkmarkt zu erreichen. Der Wandel von vertikal integrierten, proprietären Netzwerksystemen zu horizontalen, offenen Netzwerksystemen mit offenen Standards, Schnittstellen und Betriebssystemen soll die Kontrolle über das Netzwerkequipment in die Hände der Netzwerkbetreiber legen, ähnlich wie es im Computermarkt geschehen ist. Um dies zu erreichen, etabliert SDN eine Reihe von Designprinzipien, auf die in den folgenden Abschnitten näher eingegangen werden soll [Pet+21].

2.1.2 Data Plane und Control Plane

Die Vermittlungsschicht (Network Layer) eines Netzwerks kann in zwei Ebenen aufgeteilt werden, die miteinander interagieren, um die notwendigen Protokolle für die Host-zu-Host-Kommunikation in einem Netzwerk zu implementieren [KR21].

- **Control Plane**: Steuert *wie* sich das Netzwerk verhalten soll (Beispiel: Routing von Datenpaketen)
- **Data Plane**: Umsetzung des Verhaltens durch entsprechende Verarbeitung und Weiterleitung von individuellen Datenpaketen

Die Grundidee hinter SDN ist die Entkopplung bzw. Disaggregation von Data Plane und Control Plane sowie die Einführung einer offenen Schnittstelle zwischen diesen Ebenen [Pet+21]. In traditionellen Netzwerken sind beide Ebenen fest in einem Gerät integriert. Das bedeutet, dass die Entscheidungen, die das Gerät trifft, um Datenpakete zu verarbeiten, auch auf dem Gerät getroffen werden.

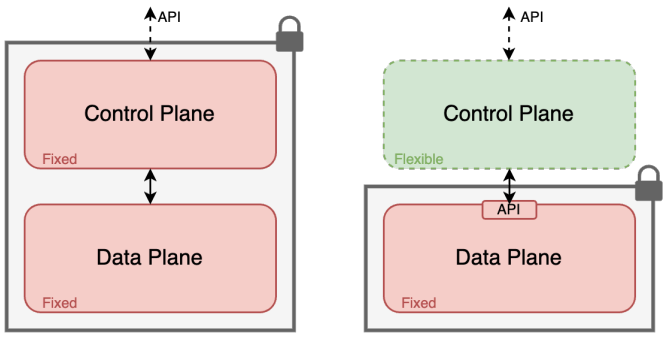

Abbildung 1: Klassisches Netzwerksystem (links) vs. Offenes Netzwerksystem (rechts)

Die Auftrennung hat zur Folge, dass die einzelnen Steuerelemente eines Netzwerks modularisiert werden können.

Was die genaue Umsetzung der Control Plane angeht, gibt es drei verschiedene Ansätze:

1. **Distributed**: Die Control Plane verbleibt auf den individuellen Netzwerkgeräten, die dementsprechend autonom voneinander agieren. Dies entspricht dem traditionellen Ansatz der letzten Jahrzehnte.
2. **Centralized**: Die Control Plane wird logisch zentralisiert und auf einen oder mehrere Server (Cluster) ausgelagert. Sie ist optimalerweise skalierbar und redundant ausgelegt, da sie sonst einen *Single Point of Failure* des Netzwerks darstellen würde. Der Vorteil dieses Ansatzes ist, dass die Control Plane eine Gesamtübersicht des Netzwerks hat und dieses so gegenüber Anwendungen abstrahieren und sogar global optimieren kann.
3. **Hybrid**: Eine Mischung aus den beiden Ansätzen, bei der die Control Plane sowohl auf den Netzwerkgeräten als auch auf zentralen Servern läuft [Dut19].

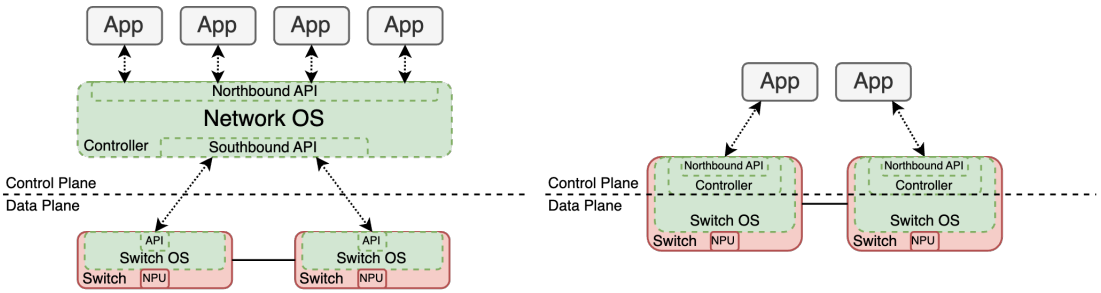

Abbildung 2: Centralized SDN (links) vs. Distributed SDN (rechts)

Dieser Aufteilung nach kann in einem SDN zwischen zwei verschiedenen Arten von *Betriebssystemen* unterschieden werden: dem *Switch OS*, welches auf den einzelnen Netzwerkgeräten (Switches) läuft und die Data Plane implementiert, und dem *Network OS*, einer Plattform für die zentrale Steuerung und Überwachung der Data Plane via APIs. Das Network OS verfügt ebenfalls über weitere APIs für andere Anwendungen, wie beispielsweise Netzwerk-Management- oder Monitoring-Systeme, die wiederum die Control Plane steuern und überwachen [Pet+21]. Die Schnittstelle zur Control Plane wird *Southbound API* genannt, während die Schnittstelle, über die Anwendungen mit dem Network OS kommunizieren, als *Northbound API* bezeichnet wird [KR21].

2.2 Disaggregated Network Hardware

Disaggregated Network Devices stellen die notwendigen Hardware-Ressourcen für SDN zur Verfügung. Sie unterscheiden sich erheblich von traditionellen Netzwerkgeräten und ähneln in ihrem Design eher Servern. Die Idee hinter Disaggregated Network Devices ist es, die Hardware von der Software zu trennen. Dies steht im Kontrast zu traditionellen Netzwerkgeräten, bei denen die Hardware (Switch, Router usw.) fest mit der Netzwerksoftware (Betriebssystem und Netzwerkmanagement-Tools) des Herstellers verbunden ist. [Dut19]

Die Hardware kann generische, hochleistungsfähige Komponenten umfassen, auf denen Software verschiedener Anbieter ausgeführt wird. Diese Flexibilität ermöglicht es den Nutzern, die Netzwerksoftware (Betriebssysteme, Steuerungssoftware) auszuwählen, die am besten zu ihren spezifischen Bedürfnissen passt. [Dut19; Pet+21]

Die ursprünglichen Hardware-Spezifikationen für Disaggregated Network Devices entstammen dem Open Compute Project (OCP), einer Initiative von Facebook und anderen Firmen, die darauf abzielte, die Hardware-Designs für Rechenzentren zu standardisieren und zu öffnen. Sie waren das Resultat des Redesigns von Facebooks Datacenter in Prineville, Oregon, im Jahr 2011 [Loo21].

Das Open Compute Networking Project umfasst sowohl Hardware- als auch Software-Designs für Netzwerkgeräte, hauptsächlich für Ethernet-Switches. Die Hardware-Spezifikationen für Disaggregated Network Devices umfassen fertige Merchant Silicon Chipsets von Herstellern wie Broadcom, Intel und Cavium sowie ein CPU-Board (x86 oder ARM) für den Betrieb eines Betriebssystems [Pet+21]. Es existieren viele verschiedene Betriebssysteme (Switch OS), die für den Einsatz auf dieser speziellen Netzwerkhardware ausgelegt sind. Bekannte Beispiele sind Open Network Linux (ONL) [ope24], SONiC (Software for Open Networking in the Cloud) [Son24] und Cumulus Linux [NVI24]. Sie haben die Aufgabe, die notwendigen Netzwerkapplikationen zu hosten, welche den Switching-ASIC steuern und so die gewünschten Netzwerkfunktionen implementieren.

Eine weitere wichtige Eigenschaft von Disaggregated Network Devices ist das Vorhandensein von BIOS-Firmware, die es ermöglicht, das gewünschte Switch OS zu booten. Das de-facto Standard-BIOS für Bare-Metal-Switches ist das Open Network Install Environment (ONIE), welches ebenfalls von der Open Compute Foundation entwickelt wurde [Ove24]. Es fördert disaggregierte Netzwerkansätze, indem es eine universelle Installationsumgebung für unterschiedliche Betriebssysteme auf der gleichen Netzwerkhardware ermöglicht.

Das Ziel des verwendeten Software-Stacks auf Bare-Metal-Switches ist es, die Hardwareebene so weit wie möglich zu abstrahieren, um so die Entwicklung von Netzwerkapplikationen zu erleichtern. Die Hardware-Abstraktionsschicht setzt sich dabei aus Hardware-Treibern und APIs zusammen [Dut19]. Die speziellen Hardware-Treiber, die für Bare-Metal-Switches benötigt werden, sind meistens proprietär und werden von den Herstellern der Switch-ASICs entwickelt und zur Verfügung gestellt. Ohne diese Treiber wäre die Kommunikation zwischen dem Switch OS und dem Switch-ASIC nicht möglich.

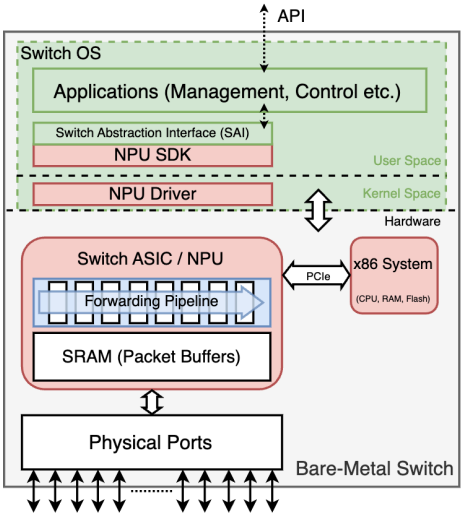

Abbildung 3: Bare-Metal-Switch: High-Level-Schema

In der oberen Abbildung ist das High-Level-Schema eines typischen Bare-Metal-Switches dargestellt. Proprietäre Software oder Systeme sind in der Abbildung rot markiert, offene Systeme grün.

- **Switch ASIC / Network Processing Unit (NPU)**: Verarbeitung (z. B. Parsen von Paket-Headern) und Weiterleitung von Datenpaketen
- **Forwarding Pipeline**: Logik für das Weiterleiten von Datenpaketen
- **SRAM**: Puffer für Datenpakete, die verarbeitet werden
- **x86-System**: Steuerung des Switch-ASICs und Ausführung des Switch OS. Kann außerdem Control Plane Funktionen (z. B. Routing-Protokolle) ausführen. Üblicherweise x86-basierter Chip, der via PCIe mit dem Switch-ASIC kommuniziert. Kann allerdings auch ARM-basiert sein.
- **Network Interface**: Schnittstelle zum Netzwerk (Ethernet). Die Ports unterstützen in der Regel standardisierte, steckbare Transceiver-Module (z. B. SFP/SFP+/QSFP/QSFP+).

Die Verwendung der Funktionen des Switching-ASICs wird erst mit dem sogenannten Switch Abstraction Interface (SAI) möglich, welches eine Brücke zwischen der geschlossenen Software des Chip-Herstellers und einem offenen Switch OS schlägt [Dut19]. Es entspricht einer Art Standard-Interface zur Hardware-Abstraktion von NPUs und wird ebenfalls vom Hersteller zur Verfügung gestellt.

Im Kontext von SDN sind diese offenen Interfaces wichtig, da diese es erst ermöglichen, die Control Plane von der Data Plane zu trennen und so die Steuerung des Netzwerks zu zentralisieren.

Die genannten Ebenen von Hardware und Software, die auf Bare-Metal-Switches zum Einsatz kommen, lassen sich in drei Ebenen einordnen [Odo16].

- **Control Plane**: Falls der Ansatz einer verteilten Control Plane gewählt wird, wird die Data Plane mithilfe der genannten Hardware-Abstraktionsmethoden direkt auf dem Switch gesteuert.

- **Management Plane**: Stellt ein Management-Interface für den Nutzer bereit, um Hardwarefunktionen direkt auf dem Switch zu konfigurieren. Umfasst beispielsweise die Port-Konfiguration, VLAN-Management etc.
- **Data Plane**: Hier findet die eigentliche Verarbeitung von Datenpaketen statt.

2.2.1 Programmable Switches

Programmable Switches sind eine spezielle Art von Bare-Metal Switches, die zusätzlich zu den Standardfunktionen eines Switches auch die Möglichkeit bieten, die Forwarding-Logik der NPU zu programmieren [Dut19; Pet+21]. Das ermöglicht es, die Switches an spezifische Anforderungen anzupassen und völlig neue Netzwerkfunktionen zu implementieren, ohne die Hardware austauschen zu müssen.

Wie bereits zuvor erläutert, wird die Data Plane eines Disaggregated Network Switches von der Control Plane via APIs gesteuert. Ein neuerer Ansatz ist es, dass zusätzlich die Data Plane selbst ebenfalls in Software definiert wird und direkt programmierbar ist.

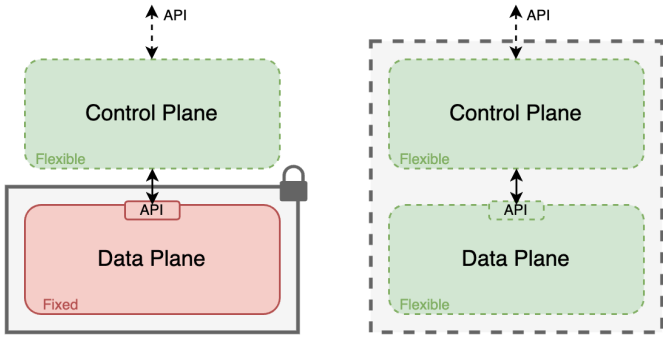

Abbildung 4: Disaggregiertes Netzwerksystem (links) vs. Programmierbares Netzwerksystem (rechts)

Diese Funktionalität erlaubt eine noch feinere Kontrolle über das Forwarding von Datenpaketen und ermöglicht es, spezielle oder sogar völlig neuartige Netzwerkfunktionen direkt auf dem Switch zu implementieren. Herkömmliche Switches haben eine feste Forwarding-Logik, die in der Regel auf dem Switch ASIC fest verdrahtet ist.

Diese Verarbeitungskette innerhalb traditioneller Switches wird daher auch als *Fixed Pipeline* bezeichnet [Ken22; Pet+21]. Diese Pipeline zerlegt und versteht nur eine feste (fixed) Anzahl von Bytes und Header-Feldern, die in eingehenden Datenpaketen enthalten sind. Die Funktionen des ASICs werden daher, wie in der Regel üblich, während der Designphase festgelegt und können nicht nachträglich geändert werden. Programmierbare Switches verfügen hingegen über eine *Flexible Pipeline*, deren Verarbeitungslogik frei programmierbar ist.

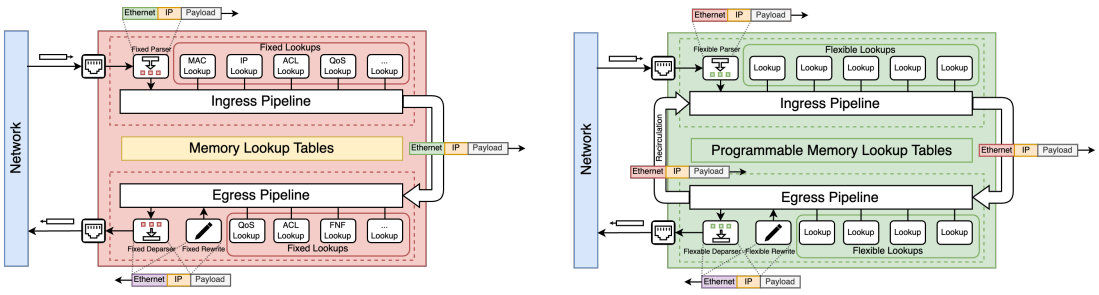

Abbildung 5: Fixed Forwarding Pipeline (links) vs. Flexible Forwarding Pipeline (rechts)

Ein Beispiel für eine programmierbare Switch-Platform ist die Intel-Tofino-Familie, eine Reihe von Switching-ASICs, die speziell für die Implementierung von programmierbaren Switches entwickelt wurde [Int24]. Intels Tofino-Chips gehörten zu den ersten verfügbaren programmierbaren ASICs, und die neuere Tofino-2-Generation erreicht Switching-Kapazitäten von bis zu 12,8 Tbit/s. Ein aktuelleres Beispiel für programmierbare Switches sind Modelle, die auf Ciscos Silicon One ASIC basieren (z.B. Cisco Catalyst 9600X). Das Flaggschiff-Modell dieser Chipfamilie, der Cisco Silicon One G200 ASIC [Cis24], bietet eine eindrucksvolle Gesamt-Switching-Kapazität von 51,2 Tb/s und stellt so einen wichtigen Schritt in der Einführung von programmierbarer 800G-Ethernet-Technologie dar.

Das Grundprinzip einer programmierbaren Forwarding-Logik kann mit dem der sogenannten Protocol Independent Switch Architecture (PISA) verdeutlicht werden, welche es ermöglicht, die Verarbeitungslogik von Datenpaketen in Software zu definieren [Noa18; Pet+21]. Die PISA-Architektur existiert unabhängig von der spezifischen Implementierung einer Pipeline und kann als Richtlinie für die tatsächliche Umsetzung und Definition einer entsprechenden Architektur dienen.

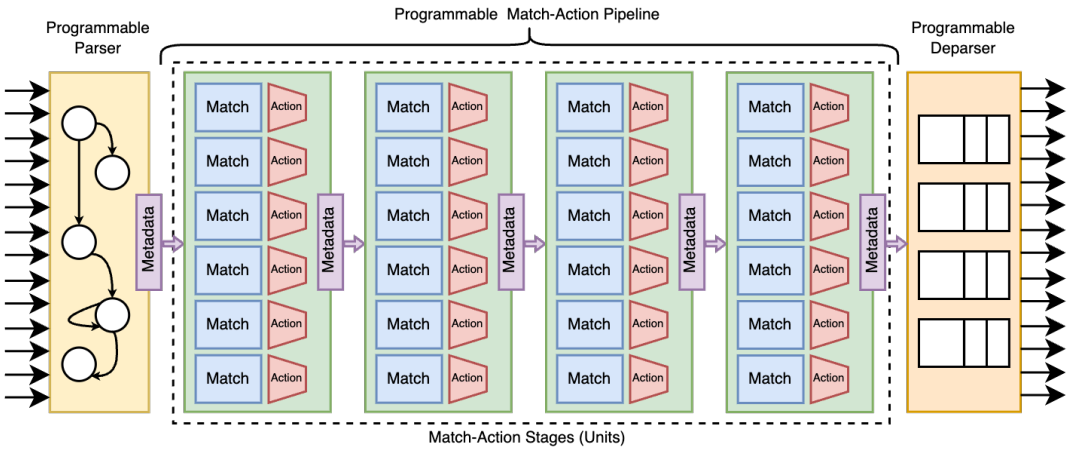

Abbildung 6: High-Level-Schema: PISA Multi-Stage-Pipeline

Die Verarbeitung der Datenpakete erfolgt in einer sogenannten Forwarding Pipeline, welche aus mehreren Stages bzw. Stufen besteht. Eine Forwarding Pipeline ist eine Sequenz von Verarbeitungsstufen, die ein Datenpaket durchläuft, bevor es weitergeleitet wird. Die PISA-Architektur ermöglicht es, diese Forwarding Pipeline selbst zu definieren, indem die einzelnen Stages der Pipeline programmiert werden können [Hau+23; Pet+21]. Innerhalb der Forwarding Pipeline wird zwischen folgenden Stages unterschieden:

1. **Parsing**: In der ersten Stufe wird das Paket analysiert. Hier werden die verschiedenen Protokoll-Header des Pakets, wie Ethernet, IP, TCP usw., extrahiert und für die weiteren Stufen verfügbar gemacht.
2. **Match-Action-Units**: Nachdem das Paket analysiert wurde, folgen eine oder mehrere Match-Stufen. In diesen Stufen werden die Headerfelder gegen vordefinierte Regeln abgeglichen (gematched). Abhängig vom Ergebnis dieses Matching-Prozesses werden bestimmte Aktionen ausgelöst. Diese Aktionen umfassen z.B. das Modifizieren von

Header-Feldern, das Umleiten von Paketen auf bestimmte Ports, das Anwenden von Quality-of-Service-Regeln oder das Verwerfen des Pakets.

3. **Deparsing**: In der letzten Stufe der Pipeline werden die ggf. modifizierten Paketdaten wieder in ein reguläres Datenpaket serialisiert, bevor es weitergeleitet wird.

Neben diesen Schritten werden für jedes Paket auch Metadaten gespeichert. Diese Metadaten umfassen u.a. Paketzustandsinformationen, Input- und Output-Ports sowie Timestamps. Die Metadaten werden in Datenregistern gespeichert und können von den verschiedenen Stufen der Forwarding Pipeline gelesen und geschrieben werden, um die Verarbeitung der Pakete zu steuern. [Hau+23; Pet+21]

Die Match-Action-Units bestehen aus Memory und ALUs. Die Memory-Units, welche technisch in Form von Static Random-Access Memory (SRAM) sowie Ternary Content-Addressable Memory (TCAM) implementiert sind, speichern die festgelegten Matching-Regeln in Form von Bitmustern ab. Die ALUs (Arithmetic Logic Units) führen gewünschte Verarbeitungsalgorithmen, also Actions, aus. Zusammen ergeben sich daraus sogenannte *Match-Action-Tabellen* (MAT). [Hau+23; Pet+21]

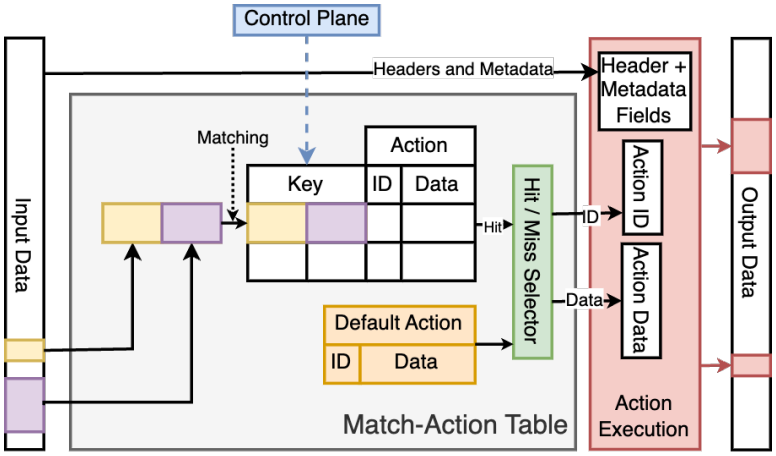

Abbildung 7: Funktionsweise einer Match-Action-Tabelle

Match-Action-Tables (MAT) sind im Prinzip Lookup-Tabellen. Sie sind der wichtigste Bestandteil der einzelnen Kontrollblöcke und bestehen hauptsächlich aus zwei Teilen:

- **Key**: Durch das Programm sowie via Control Plane definierte Felder, die gegen extrahierte Felder und Metadaten (Lookup Key) aus dem Paket abgeglichen werden.
- **Action**: Die Aktionen, die auf ein Paket angewendet werden, wenn eine Übereinstimmung gefunden wird. Sollte keine Übereinstimmung gefunden werden, wird eine Default-Action angewendet.

Eine Besonderheit dieser Tabellen ist, dass sie von der Control Plane modifizierbar sind. Das erlaubt dem Controller eine dynamische Steuerung des Verhaltens der Data Plane, da direkt Einfluss auf die Forwarding-Logik genommen werden kann. Außerdem können aktuelle Daten ausgelesen werden, um den Zustand der Pipeline zu überwachen.

Es stellt sich nun die Frage, wie genau der Entwickler die Forwarding-Pipeline in Software definieren kann. Schließlich wäre eine direkte Programmierung der beteiligten Hardwarekomponenten mit extrem hohem Aufwand verbunden. Das folgende Kapitel versucht diese

Frage zu beantworten, indem eine Programmiersprache vorgestellt wird, die speziell für diesen Zweck entwickelt wurde.

2.3 P4

Um die Komplexität der hardware-nahen Programmierung von Forwarding Pipelines zu umgehen, wurde eine spezielle, deklarative Programmiersprache namens P4 (Programming Protocol-independent Packet Processors) entwickelt [P4 24]. Neben Switches wie dem Intel Tofino, unterstützen mittlerweile auch Router, SmartNICs und FPGAs die Programmierung mit P4. Die Urheber dieser Sprache, das P4 Language Consortium, haben seit der initialen Bekanntgabe des Projekts im Jahr 2013 zwei verschiedene Standards veröffentlicht: P_{14} und P_{16} [Hau+23].

Letzterer Standard brachte einige Verbesserungen mit sich und ermöglichte einen umfangreicheren Support von programmierbaren Zielsystemen, indem Kernkomponenten der Sprache von Architektur-spezifischen Sprachkonzepten und Bibliotheken getrennt wurden. Aufgrund dieser erheblichen Vorteile ist P_{16} der aktuelle de-facto Hauptstandard, und die Weiterentwicklung von P_{14} wurde eingestellt [Gai20; Hau+23].

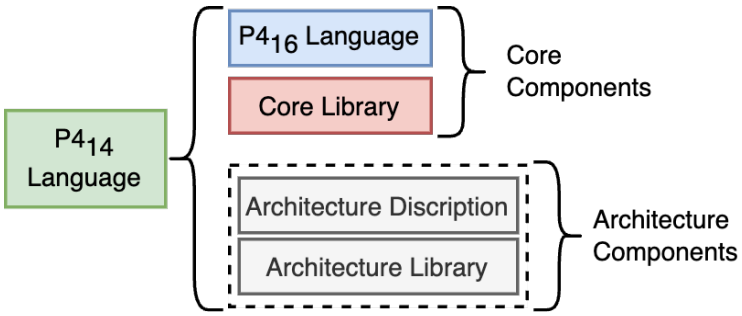

Abbildung 8: Entwicklung von P_{14} zu P_{16}

Das Ziel von P4 ist die vollständige Rekonfigurierbarkeit von Netzwerksystemen und die Befreiung von Einschränkungen fester Protokollspezifikationen. Die Sprache ermöglicht daher eine hohe Flexibilität bei der Definition der Netzwerkverarbeitungslogik, was wiederum die Reaktionsfähigkeit auf neue Anforderungen innerhalb moderner Netzwerke erhöht sowie die Entwicklung von innovativen Netzwerkdiensten erleichtert[Gai20].

Aus technischer Sicht werden diese Ziele umgesetzt, indem sämtliche Funktionalitäten der Data Plane eines programmierbaren Netzwerkgeräts durch ein P4-Programm definiert werden. Dabei soll das Kommunikationsprinzip zwischen Control Plane und Data Plane weitgehend unverändert und quasi äquivalent zu herkömmlichen Netzwerkgeräten mit fest definierter Forwarding-Logik bleiben. Das wird erreicht, indem der P4-Compiler neben einer ausführbaren Binary auch die entsprechende Data-Plane-API generiert, welche von einer Control Plane zur Steuerung des Programms verwendet werden kann. [Hau+23; Noa18; Pet+21]

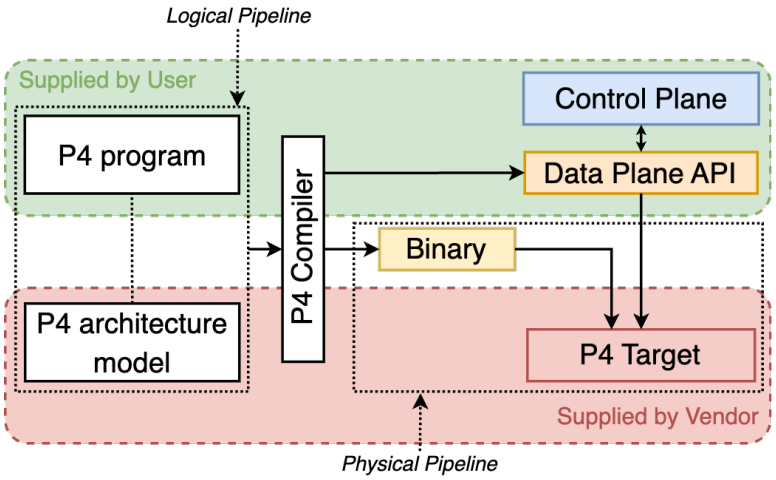

Abbildung 9: P4-Deployment

Wie in der oberen Abbildung dargestellt, wird für das Ausführen eines P4-Programmes auf einer bestimmten Zielhardware, neben einem entsprechenden P4-Compiler auch eine P4-kompatible Architekturdefinition benötigt [Hau+23]. Häufig existieren zudem hardwarespezifische API-Definitionen und Bibliotheken, um den vollen Funktionsumfang der Hardware zu nutzen. Aufgrund dessen kann nicht garantiert werden, dass ein P4-Programm auf unterschiedlichen Architekturen ohne Anpassungen funktioniert. Entsprechend trifft die Eigenschaft der Portierbarkeit lediglich auf Zielhardware zu, welche auf Basis der selben Architekturdefinition entwickelt wurde. [Pet+21]

Vereinfacht ausgedrückt kann man sagen, dass es sich bei P4-Programmen um die Definition einer *logischen Forwarding-Pipeline* handelt. Die Architekturdefinition entspricht einem Mapping auf die tatsächliche *physische Forwarding-Pipeline* des jeweiligen Switching-ASICs [Noa18; Pet+21]. Dieses Mapping erlaubt es dem P4-Compiler, die logische Pipeline in die physische Pipeline zu übersetzen, die auf der Zielhardware ausgeführt wird. Da es sich hier um vergleichsweise sehr neuartige Technik handelt, gibt es leider noch keine einheitliche P4-Architekturdefinition, die von allen Herstellern gleichsam unterstützt wird [Pet+21]. Der wahrscheinlich beste Kandidat für eine solche Definition ist die Portable Switch Architecture (PSA)[Pet+21; The21], die aktuell von dem P4 Language Consortium selbst entwickelt wird [Gai20]. Ein Beispiel für eine herstellerspezifische Architekturdefinition ist die *Tofino Native Architecture*, auf welche in den kommenden Abschnitten noch näher eingegangen wird.

Die P4-Sprache bringt für die Programmierung und Definition der Data Plane folgende Funktionen bzw. Abstraktionen mit sich [The24]:

- **Header Types**: P4 ermöglicht die Definition von benutzerdefinierten Header-Formaten, die die Struktur der Datenpakete festlegen.
- **Parser**: Extrahiert die Header-Felder aus empfangenen Datenpaketen und stellt sie für die weiteren Verarbeitungsschritte zur Verfügung.
- **Tables**: Assoziiert Aktionen mit User-spezifischen Regeln, die auf die Header-Felder der Pakete angewendet werden.
- **Actions**: Definieren die Manipulation der Header-Felder und Metadaten der Pakete.

- **Match-Action-Units**: Definieren die Verarbeitungslogik der Pakete, indem Actions auf Pakete angewendet werden, die bestimmte Regeln erfüllen.
- **Control Flow**: Ermöglicht die Definition von Kontrollstrukturen, um die Verarbeitung von Paketen zu steuern.
- **Extern Objects**: Ermöglicht die Integration von externen Bibliotheken und Funktionen in das P4-Programm, um hardware-spezifische Funktionen zu nutzen.
- **User-defined Metadata**: Benutzerdefinierte Datenstrukturen, die für die Verarbeitung von Paketen verwendet werden können.
- **Intrinsic Metadata**: Enthält Informationen über den Zustand des Pakets, wie z.B. den Input- und Output-Port, den Zeitstempel, den Pakettyp usw.

Diese Grundkonstrukte bzw. Funktionen der P4-Sprache stellen die Bausteine für eine logische Forwarding-Pipeline dar, welche dynamisch anpassbar ist. Folgendes kleine Code-Beispiel zeigt eine Switch-Definition, in der die verschiedenen Schritte einer Forwarding-Pipeline aufgerufen werden.

Diese Schritte entsprechen im Programm Funktionen, welche vom Entwickler frei implementierbar sind.

```
Switch(
    MyParser(),
    MyVerifyChecksum(),
    MyIngress(),
    MyEgress(),
    MyComputeChecksum(),
    MyDeparser()
) main;
```

Welche Schritte in der Switch-Definition vorhanden sein müssen, hängt von der Forward-Pipeline und damit von der Architektur des Zielsystems ab. Folgendes Beispiel zeigt die Implementierung eines Parsers:

```
parser MyParser(
  packet_in packet,
  out headers hdr,
  inout metadata meta,
  inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
}
```

Dieser verhältnismäßig simple Parser extrahiert den Ethernet- und den IPv4-Header eines Frames und stellt diese Werte dem nächsten Schritt der Pipeline für die weitere Verarbeitung zu Verfügung. Damit das funktioniert müssen allerdings die Header zuvor definiert werden.

```
#include <core.p4>

const bit<16> TYPE_IPV4 = 0x800;

typedef bit<48> macAddr_t;
typedef bit<32> ip4Addr_t;

header ethernet_t {
    macAddr_t dstAddr;
    macAddr_t srcAddr;
    bit<16>   etherType;
}

header ipv4_t {
    bit<4>    version;
    bit<4>    ihl;
    bit<8>    diffserv;
    bit<16>   totalLen;
    bit<16>   identification;
    bit<3>    flags;
    bit<13>   fragOffset;
    bit<8>    ttl;
    bit<8>    protocol;
    bit<16>   hdrChecksum;
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

struct headers {
    ethernet_t   ethernet;
    ipv4_t       ipv4;
}

struct metadata { /* empty */ }
```

Verschiedene Architekturen stellen dem Entwickler unterschiedliche Bausteine zur Verfügung, die in der Regel auf die spezifischen Eigenschaften der Zielhardware zugeschnitten sind. Diese zusätzlichen Funktionen sind als Bibliotheken in das P4-Programm integrierbar [The24].

All diese genannten Eigenschaften reihen die P4-Sprache als wichtige Technologie in das SDN-Paradigma ein und machen sie zu einem modernen Werkzeug für Wissenschaftler, Netzwerkbetreiber und Entwickler. Wie genau P4 allerdings SDN-Netzwerke bereichert, wird erst durch das korrekte Zusammenspiel mit der Control Plane ersichtlich, um welches es im nächsten Abschnitt gehen soll.

2.3.1 Data Plane APIs

Das Laufzeitverhalten einer P4-definierten Data Plane kann mit einer *Data Plane API* bzw. *Runtime API* gesteuert werden, welche es der Control Plane erlaubt, auf P4-Objekte, wie bestimmte Tabellen, zuzugreifen [Hau+23].

Folgende Abbildung zeigt das Grundprinzip und die verschiedenen Funktionen einer Data Plane API:

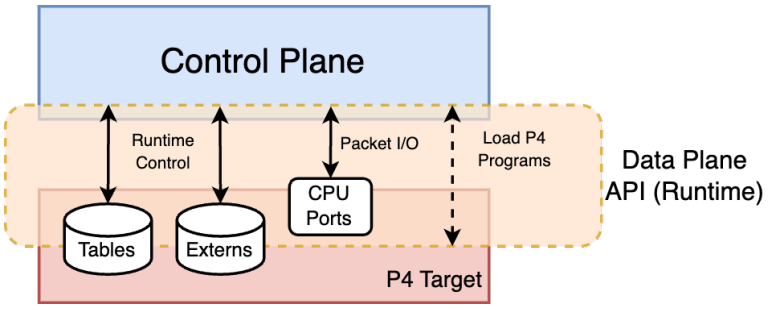

Abbildung 10: Data Plane API bzw. Runtime API eines P4-Programms

Die technische Implementierung dieser API bestimmt dabei, ob der Zugriff von einem entfernten System, also *remote*, oder nur lokal möglich ist [Hau+23]. APIs für rein lokale Kontrolle sind in der Regel durch entsprechende Treiber verfügbar. In den folgenden Abschnitten wird ein Fokus auf APIs für Remote Control gelegt, da diese für die weitere Umsetzung der Aufgabenstellung dieser Arbeit relevanter sind.

2.3.1.1 P4Runtime

Bei P4Runtime handelt es sich um eine herstellerunabhängige und protokollunabhängige Programmierschnittstelle (Application Programming Interface, API). Die Control Plane (z.B. SDN-Controller) kann diese Schnittstelle verwenden, um eine P4-definierte Data Plane zu steuern und so Einfluss auf die Verarbeitungslogik von Datenpaketen zu nehmen. [The24]

Die genaue technische Spezifikation ist relativ umfangreich und komplex, es soll daher hier nur kurz darauf eingegangen werden, um ein grobes Verständnis zu vermitteln. Die P4Runtime-Schnittstelle nutzt zur Kommunikation zwischen Control- und Data Plane das Konzept von Remote Procedure Calls (RPCs) [PD22; Pau23].

Bei RPCs handelt es sich um einen Mechanismus, der es Programmen erlaubt, Funktionen auf anderen Systemen auszuführen:

- Ein Prozess auf System A ruft eine Funktion auf System B auf.
- Der Prozess auf System A wird suspendiert und die Ausführung wird an System B übertragen.
- Nach Ausführung der Funktion wird der Rückgabewert an System A zurückgegeben und der Prozess auf System A wird fortgesetzt.

Damit das funktioniert, werden sogenannte *Stubs* benötigt, die die Kommunikation zwischen den Systemen übernehmen. Stubs sind Funktionen, die als Stellvertreter für die entfernte (remote) Funktion fungieren und die Netzwerkkommunikation übernehmen. Man unterscheidet dabei zwischen Client-Stubs (Proxy) und Server-Stubs (Skeleton). [PD22; Pau23]

Die tatsächliche Realisierung dieses Konzepts erfolgt mit dem gRPC-Framework (gRPC Remote Procedure Calls) [gRP24]. Dabei handelt es sich um ein Open-Source-Projekt von Google, das eine plattformunabhängige Implementierung dieses Prinzips darstellt. Es nutzt HTTP/2 als Kommunikationsprotokoll für den bidirektionalen Datenaustausch zwischen Client und Server. Eine besondere Eigenschaft von gRPC ist, dass die Stubs automatisch auf Basis der API-Definition in einer beliebigen Programmiersprache generiert werden können. [PD22]

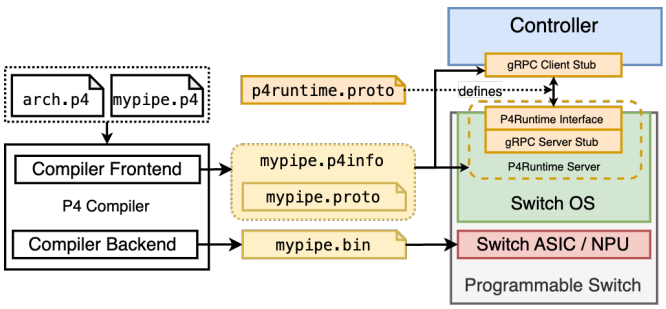

Abbildung 11: Grundlegende Architektur von P4Runtime

Die Kommunikation von strukturierten Daten erfolgt in dem, ebenfalls von Google entwickelten, Datenserialisierungsformat *Protocol Buffers* (Protobuf) [Goo24]. Die Daten werden dabei in ein Binärformat serialisiert, was eine sehr effiziente Datenübertragung ermöglicht [PD22]. Damit Client und Server die Daten serialisieren und deserialisieren können, muss die Struktur der Daten in einem Schema definiert werden. Diese Definition erfolgt in `.proto` -Dateien, die festlegen, wie eine Nachricht formatiert ist und entsprechend, wie die empfangenen Bytes zu interpretieren sind [PD22]. Daraus folgt, dass diese Spezifikation Client und Server bekannt sein muss, um eine korrekte Kommunikation zu gewährleisten.

Die P4Runtime-Schnittstelle entspricht dem gRPC-Server-Stub der Data Plane (z.B. Switch), der Client-Stub hingegen läuft auf der Control Plane (z.B. SDN-Controller) und muss daher vom User bereitgestellt werden. Beide Komponenten ermöglichen so die Kommunikation zwischen diesen beiden Netzwerkebenen. [Hau+23; Pet+21]

Die notwendige Protobuf-Spezifikation ist ein Artefakt des Kompilierungsprozesses eines P4-Programms und ist in der sogenannten *P4Info*-Datei enthalten, welche dessen Struktur und modifizierbare Elemente beschreibt [Hau+23; Pet+21] . Sie dient daher als Referenz für die korrekte Kommunikation zwischen Control- und Data Plane.

2.3.2 Tofino Native Architecture

Wie bereits erwähnt, besteht die Forwarding-Pipeline eines programmierbaren Netzwerkgeräts aus mehreren Stufen, die die Verarbeitung von Paketen steuern. Diese einzelnen Stufen können nun mit den Bausteinen, die uns die P4-Sprache zur Verfügung stellt, definiert und programmiert werden. Zudem wurden die Konzepte der physischen sowie einer logischen Forwarding-Pipeline erläutert.

Im Folgenden wird nun die Tofino Native Architecture (TNA) [Int21] vorgestellt, eine spezielle, hardware-spezifische Architekturdefinition, die von Barefoot Networks entwickelt wurde. Die Firma Barefoot Networks wurde 2019 von Intel übernommen [Hau+23]. Als Resultat existiert heute die Intel Tofino-Familie von Ethernet Switching-ASICs, die auf der TNA basieren.

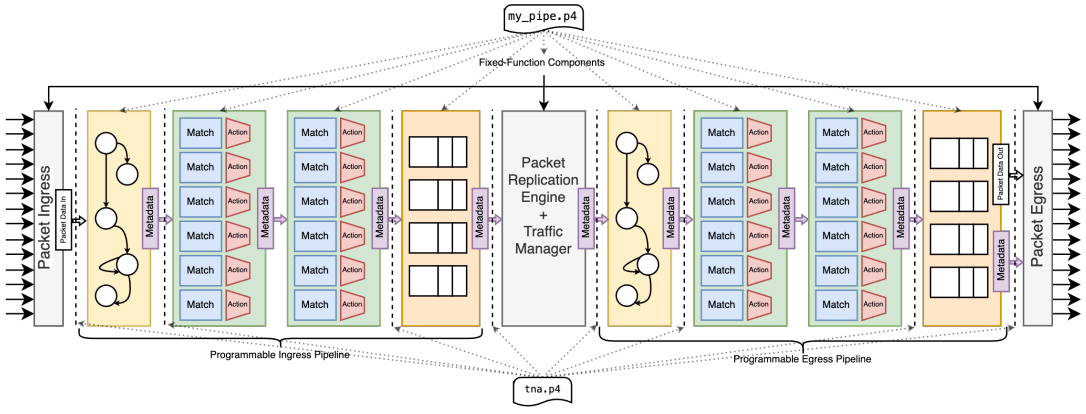

Abbildung 12: Grundlegende Architektur von P4Runtime

Im Gegensatz zu PISA, die eine eher abstrakte Spezifikation darstellt, handelt es sich bei der TNA um eine konkrete Architekturdefinition, die speziell für die Intel Tofino-Familie ent-

wickelt wurde [Pet+21]. Laut der offiziellen Dokumentation von Intel ähnelt die TNA zwar stark der PSA, sie sind jedoch nicht identisch. Die TNA ist in der Lage, P4Runtime als API für die Kommunikation zwischen Control- und Data Plane zu verwenden, bringt jedoch zusätzlich das Barefoot Runtime Interface (BRI) mit sich, welches speziell für die Tofino-Familie entwickelt wurde, P4Runtime aber sehr ähnlich ist [Hau+23].

Um die Eigenschaften und Besonderheiten der TNA zu verstehen, sollte zunächst die logische Forwarding-Pipeline betrachtet werden und inwiefern sie sich von der zuvor diskutierten, sehr grundlegenden Pipeline der PISA-Architektur unterscheidet und diese funktionell erweitert. Die beiden hauptsächlichen Unterschiede sind:

1. **Traffic Manager**: Eine zusätzliche Stufe innerhalb der Pipeline, die sogenannte *Fixed-Function Components* enthält. Dabei handelt es sich um Funktionen oder Komponenten, die zwar konfigurierbar, aber nicht reprogrammierbar (daher *fixed*) sind. Sie erlauben zum Beispiel Paket-Queuing, -Scheduling sowie Paket-Replikation durch Mirroring oder Cloning. Die TNA enthält zudem mehrere konfigurierbare Paket-Generatoren.
2. **Ingress- und Egress-Processing**: Die Forwarding-Pipeline wird in zwei Teile aufgeteilt. Der Ingress-Teil ist für die Verarbeitung von eingehenden Paketen zuständig, während der Egress-Teil für die Verarbeitung von ausgehenden Paketen zuständig ist. Beide Teile können unabhängig voneinander programmiert werden.

Je nach ASIC ist die individuelle Programmierung von bis zu vier identischen Pipelines möglich. Jede Pipeline besitzt 16 mal 100 Gbit/s Interfaces, was die Verarbeitung sehr hoher Datenraten ermöglicht [Int21]. In folgender Abbildung wird das Modell mit zwei Pipes einmal schematisch dargestellt. Dieses Modell entspricht der Hardware, die auch im Netzwerklabor der HTW vorhanden ist.

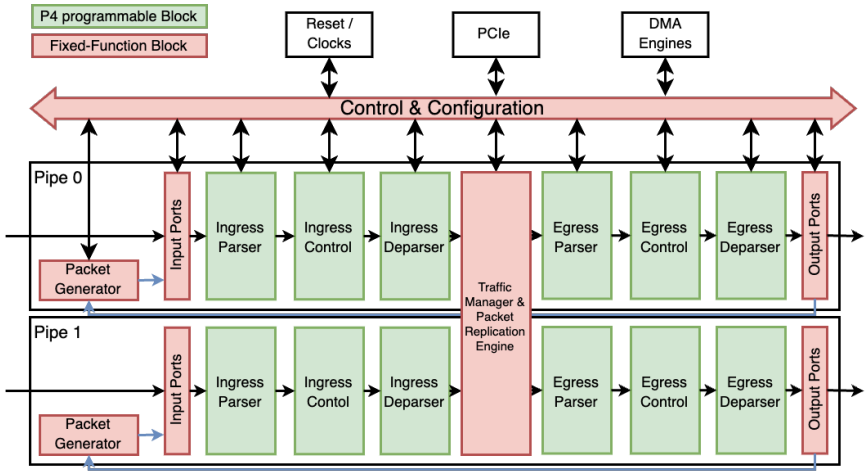

Abbildung 13: 2-Pipe Intel-Tofino Blockdiagramm

Es wird sichtbar, dass die TNA im Vergleich zu der zuvor betrachteten Forwarding-Pipeline der PISA-Architektur deutlich komplexer ist und mehr Funktionalitäten bietet. So werden nun sehr umfangreiche Verarbeitungsschritte ermöglicht. Einige Beispiele sind:

- **Resubmission**: Ein Paket kann vom Ende der Ingress-Pipeline zurück an den Anfang der Ingress-Pipeline geschickt werden, um es erneut zu parsen.

- **Recirculation**: Ein Paket kann vom Ende der Egress-Pipeline zurück an den Anfang der Ingress-Pipeline geschickt werden, um es erneut zu verarbeiten.
- **Mirroring**: Ein Paket kann innerhalb der Ingress- oder Egress-Pipeline dupliziert werden, wobei Original und Kopie unterschiedlich verarbeitet werden können.

Zusätzlich zu diesen Features bietet die TNA noch einige andere hardware-spezifische Funktionalitäten, die via Extern Objects oder kurz Externs in ein P4-Programm eingebunden werden können. Dazu gehören unter anderem:

- **Counters**: Paket- und/oder Byte-Zähler, die für Monitoring und Statistik verwendet werden können. Nur von der Control Plane auslesbar.
- **Digest**: Mechanismus zum Senden von Nachrichten von der Data Plane zur Control Plane.
- **Meters**: Messung und Erkennung von Paketflusseigenschaften nach RFC2698.
- **Checksum**: Verifizierung und Berechnung von Header-Checksummen.
- **Hash**: Berechnung von Hash-Werten aus Header- oder Metadaten-Feldern. Wird häufig für die Implementierung von Load-Balancing verwendet.
- **Mirror**: Duplizierung von Paketen. Erlaubt *Clone ingress to egress (CI2E)* und *Clone egress to egress (CE2E)*.
- **Register**: Allgemeine Speicherbereiche, die während des Paket-Forwardings gelesen und beschrieben werden können.
- **Random**: Generierung von gleichverteilten Pseudozufallszahlen.
- **Packet Generation**: Generiert Pakete innerhalb einer Pipeline. Zur Initialisierung dienen ein Paket-Buffer sowie viele Einstellungsmöglichkeiten.

Diese architektur-spezifischen Features machen die TNA zu einer sehr leistungsfähigen Plattform für die Entwicklung von P4-Programmen. Besonders die Möglichkeit, einzelne Pakete zu replizieren und an verschiedenen Stellen in der Pipeline zu verarbeiten, erlaubt die Implementierung von Data Planes mit sehr komplexer Verarbeitungslogik.

2.3.2.1 Barefoot Runtime Interface

Die Tofino Architecture bringt eine eigene, hardware-spezifische API mit sich, die als Barefoot Runtime Interface (BRI) bezeichnet wird. Das BRI setzt sich genau genommen aus zwei APIs zusammen [Hau+23]:

- **Bfrt API**: Eine API für die rein lokale Steuerung mit Unterstützung von Python, C und C++.
- **BF Runtime**: Eine API für asynchrone Remote-Steuerung via gRPC, die sehr ähnlich zu P4Runtime ist.

Es bietet zusätzliche Funktionen und Features, die speziell auf die Tofino-Architektur zugeschnitten sind. Die Entwicklungsumgebung für die Tofino-Familie kommt bereits mit einigen Software-Bibliotheken, die die Verwendung des BRI erleichtern und so die Implementierung von Control-Plane-Anwendungen vereinfachen.

Auch hier ermöglicht die Verwendung von gRPC die Entwicklung eigener Control-Plane-Anwendungen und macht es zudem möglich, eigene API-Clients zu entwickeln. Der Entwickler ist so relativ frei in der Wahl der Programmiersprache und der Implementierung eines

eigenen Runtime-Clients, solange die Protocol Buffer Definitionen eingehalten werden. Ein Beispiel dafür ist das Projekt *Rust BF Runtime Interface* (rbfrt).

2.3.2.2 Tofino Model

Für die Entwicklung von P4-Programmen für die Tofino-Familie bietet Intel eine eigene Entwicklungsumgebung an, die das sogenannte Tofino Model beinhaltet [Int24]. Dieses ermöglicht das Entwickeln und Testen komplexer Anwendungen für die Tofino-Architektur, ohne dass die physische Hardware vorhanden sein muss. Das Model entspricht dabei einer normalen User-Space-Anwendung, welche die Tofino-Architektur in Software emuliert. Zwar gibt es dabei natürlich einige Limitationen, wie z. B. die fehlende Unterstützung von sehr spezifischen Hardware-Funktionen, jedoch ist es ein sehr nützliches Werkzeug für die anfängliche Entwicklung von Prototypen und Schnittstellen.

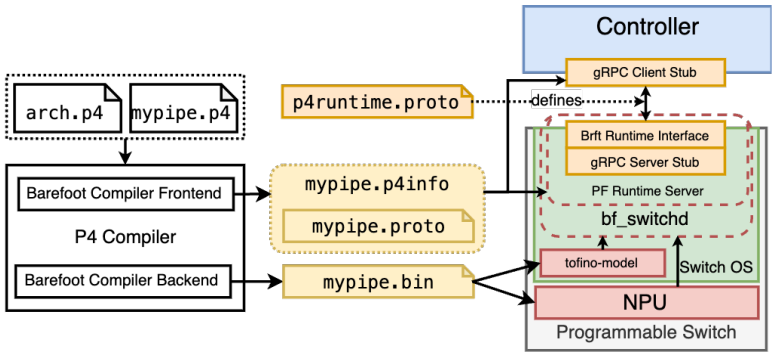

Abbildung 14: Grundlegende Architektur von des BRI mit optionalen Tofino-Modell

Die obere Abbildung soll noch einmal das Konzept des Tofino Models verdeutlichen und wie sich die Kernkomponenten von den zuvor diskutierten Komponenten unterscheiden. Der Runtime Server ist bei Tofino Teil des Switch-Prozesses `bf_switchd`, welcher zusätzlich viele Aufgaben der Management-Ebene übernimmt und über die Kommandozeile gesteuert werden kann.

2.3.3 Limitationen programmierbarer Switches

Die limitierte Anzahl an Registern, die für die Programmierung zur Verfügung stehen, macht es notwendig, dass P4-Programme effizient und sparsam mit den vorhandenen Ressourcen umgehen. Das bedeutet, dass insbesondere zustandsbehaftete Netzwerkprotokolle nur sehr schwer bzw. eingeschränkt implementiert werden können [Zho+19]. Um das verständlich zu machen, soll an dieser Stelle kurz der Unterschied zwischen zustandsbehafteten und zustandslosen Protokollen erläutert werden.

- **Zustandslose Netzwerkprotokolle**: Jede Anfrage wird unabhängig von den vorherigen Anfragen behandelt. Das bedeutet, dass die Kommunikationsteilnehmer keine Informationen über den Status der Kommunikation speichern. Einzelne Anfragen werden isoliert betrachtet und verarbeitet. Beispiele für zustandslose Protokolle sind HTTP, UDP oder DNS. [KR21; TFW21]
- **Zustandsbehaftete Netzwerkprotokolle**: Die Kommunikationsteilnehmer speichern Informationen über den Status der Kommunikation in Form einer Sitzung. Das bedeutet, dass die Kommunikationsteilnehmer Sitzungsdaten und Statusinformationen speichern

und verwenden, um die Kommunikation zu verwalten. Beispiele für zustandsbehaftete Protokolle sind TCP, FTP oder BGP. [KR21; TFW21]

Allgemein spricht man in diesem Kontext auch häufig von verbindungslosen und verbindungsorientierten Protokollen, wobei verbindungslose Protokolle keine Sitzungsdaten speichern, wohingegen verbindungsorientierte Protokolle dies tun [KR21; TFW21] . Das Speichern von Sitzungsdaten ist vergleichsweise ressourcenaufwendig, besonders wenn viele Sitzungen gleichzeitig verwaltet werden müssen. Praktisch ist zudem die Implementierung der einem verbindungsorientierten Protokoll zugrunde liegenden Zustandsmaschine in P4 aufgrund der Datenverarbeitung in Form von Forwarding-Pipelines sehr schwierig [Che+]. Hier ist zusätzlich zu beachten, dass innerhalb des Netzwerkstacks zustandsbehaftete und zustandslose Protokolle gleichzeitig auf verschiedenen Schichten existieren können. So ist beispielsweise das zustandslose Protokoll HTTP/2 [TB22] auf OSI-Schicht 7 (Anwendungsschicht) auf TCP angewiesen, welches wiederum ein zustandsbehaftetes Protokoll auf OSI-Schicht 4 (Transportschicht) ist. Möchte man also eine P4-Anwendung mit HTTP-Funktionalitäten entwickeln, so kann es je nach Anforderungen durchaus sein, dass man dafür zunächst eine P4-Implementierung von TCP benötigt.

Eine weitere Limitation ergibt sich aus der Tatsache, dass innerhalb der Forwarding-Pipelines lediglich Header- und Metadaten von Paketen verarbeitet werden können [Hau+23]. Das bedeutet, dass P4-Programme standardmäßig keine Möglichkeit haben, auf die Nutzdaten (Payload) von Paketen zuzugreifen. Sollen innerhalb der Data Plane bestimmte Trafficmuster in den Nutzdaten erkannt oder bestimmter Traffic generiert werden, so ist dies nicht ohne Weiteres möglich. Jedoch existieren mittlerweile einige experimentelle Lösungen für diese Problematik. Beispielsweise wurde im Jahr 2023 ein Ansatz für Deep Packet Inspection (DPI) vorgestellt [Gup+23], der mittels Recirculation und Truncation die Nutzdaten von Paketen extrahiert und analysiert.

2.4 Distributed Denial of Service (DDoS)

Ein Distributed Denial of Service (DDoS)-Angriff ist eine böswillige Cyber-Attacke, bei der ein Angreifer eine große Anzahl von kompromittierten Systemen verwendet, um die Ressourcen eines Servers, eines Dienstes oder eines Netzwerks zu überlasten. Das Ziel eines DDoS-Angriffs ist es, den normalen Betrieb des Zielsystems zu stören, indem dessen Kapazitäten erschöpft werden, was zu einer Verweigerung des Dienstes für legitime Benutzer führt. Bei den Systemen, die für einen DDoS-Angriff verwendet werden, handelt es sich häufig um Botnetze, die aus einer Vielzahl von infizierten Computern, Servern oder IoT-Geräten bestehen [DM04; Mah+17].

DDoS-Angriffe gehören heute zu den häufigsten Cyberattacken, und der Trend zeigt, dass sie immer häufiger und ausgefeilter werden. Dieser Umstand lässt sich einfach erklären, wenn man die Eigenschaften von DDoS-Angriffen betrachtet:

1. **Relativ einfache Durchführung**: DDoS-Angriffe erfordern oft weniger technisches Know-how als komplexere Hacking-Methoden, da sie in der Regel mit leicht verfügbaren Tools und Botnetzen durchgeführt werden können, welche u. a. im Darknet als *DDoS-as-a-Service* angeboten werden. [Eur23]

2. **Hohe Effektivität**: Angriffe können schnell zu sichtbaren Auswirkungen führen, wie Serviceausfällen oder Verlangsamungen. Die Folgen sind nicht selten finanzielle Verluste, Reputationsschäden oder sogar rechtliche Konsequenzen. [Eur23]
3. **Schwierige Abwehr**: Die verteilte Natur der Angriffe macht es schwer, den Verkehr zu filtern oder die Quelle zu blockieren. Selbst mit modernen Erkennungssystemen und Firewalls ist es oft schwierig, DDoS-Angriffe abzuwehren, da sie oft legitimen Verkehr imitieren [Eur23]. Eine verlässliche Differenzierung zwischen natürlichen Traffic-Spitzen (sogenannte *Flash Crowds Events*) und einem DDoS-Angriff ist weiterhin eine große Herausforderung [Ari+03].
4. **Trends in IoT**: Mit der zunehmenden Verbreitung von IoT-Geräten steigt auch die Anzahl von potenziellen Botnetzen, die für DDoS-Angriffe verwendet werden können. Viele IoT-Geräte sind schlecht gesichert und können leicht kompromittiert werden, um an einem Angriff teilzunehmen. [CM23]
5. **Kaum Konsequenzen für Angreifer**: DDoS-Angriffe sind oft schwer zu verfolgen, und die Strafverfolgung von Angreifern ist schwierig. Viele Angreifer bleiben unentdeckt oder ungestraft, was dazu führt, dass DDoS-Angriffe ein attraktives Mittel für Cyberkriminelle bleiben. [Mah+17]

Denial of Service (DoS)

Ein Denial of Service (DoS)-Angriff zielt darauf ab, ein Computersystem oder Netzwerk so zu überlasten, dass es nicht mehr in der Lage ist, normale Dienstleistungen bereitzustellen. Dies kann durch verschiedene Methoden erreicht werden, wie die Überlastung von Netzwerkbandbreite, die Erschöpfung von Systemressourcen oder die Ausnutzung von Schwachstellen in der Software. [Eur23]

Distributed Denial of Service (DDoS)

Im Gegensatz zu einem DoS-Angriff wird ein DDoS-Angriff von mehreren Quellen gleichzeitig ausgeführt. Diese Quellen sind oft mit Malware infizierte Computer, sogenannte Botnets, die der Angreifer kontrolliert. Da die Angriffe von vielen verschiedenen Standorten aus erfolgen, ist es schwieriger, sie abzuwehren und die Quelle des Angriffs zu identifizieren. Da es sich im Grunde um legitime Anfragen handelt, ist es zudem schwierig, sie von normalen Anfragen zu unterscheiden. [DM04; Eur23]

2.4.1 Architektur von DDoS-Angriffen

DDoS-Angriffe setzen sich aus mehreren Komponenten zusammen und verfolgen je nach Ziel des Angriffs unterschiedliche Strategien [DM04; Mah+17]. Die folgenden Komponenten sind typischerweise in einem DDoS-Angriff involviert:

- **Angreifer**: Die zentrale Instanz, die den Angriff initiiert und koordiniert. Der Angreifer kann eine Einzelperson, eine Gruppe oder eine Organisation sein, die ein bestimmtes Ziel angreifen möchte.
- **Handlers (Masters)**: In der Regel handelt es sich um Server oder Rechner, die vom Angreifer kontrolliert werden und die Befehle an die infizierten Systeme (Bots) weiterleiten. Die Handler dienen als Vermittler zwischen dem Angreifer und den Bots und sammeln je nach Implementierung ggf. Informationen über den Angriff.

- **Agenten (Bots, Zombies)**: Die Agenten sind die infizierten Systeme, die den eigentlichen Angriff durchführen. Die Agenten können Computer, Server, IoT-Geräte oder andere vernetzte Geräte sein, die vom Angreifer über Malware infiziert wurden. Die Agenten führen die Angriffe aus und senden eine große Anzahl von Anfragen an das Ziel des Angriffs.
- **Ziele (Opfer)**: Die Ziele sind die Systeme, Dienste oder Netzwerke, die vom Angriff betroffen sind. Die Ziele können Webserver, DNS-Server, Firewalls oder andere kritische Infrastrukturen sein, die durch den Angriff überlastet werden. Je nach Art des Angriffs werden Netzwerkressourcen, Systemressourcen oder Anwendungsressourcen des Ziels überlastet, was häufig zu Systemausfällen führt.

Der eigentliche Angriff kann zusätzlich in einzelne Phasen unterteilt werden, die je nach Art des Angriffs variieren können [DM04; Mah+17]. Die folgenden Phasen sind typisch für einen DDoS-Angriff:

1. **Reconnaissance**: In dieser Phase sammelt der Angreifer Informationen über mögliche Agenten. Diese haben optimalerweise eine hohe Bandbreite, genügend Systemressourcen und sind anfällig für kompromittierende Angriffe. Die Identifizierung von anfälligen Systemen kann mit Hilfe von Scannern oder anderen Tools automatisiert werden.
2. **Compromise**: In dieser Phase kompromittiert der Angreifer die ausgewählten Agenten und infiziert sie mit Malware. Die Malware etabliert ein sogenanntes Command-and-Control-Verhältnis (C2) und ermöglicht dem Angreifer so, die Kontrolle über die Systeme zu übernehmen. Die Infektion kann über verschiedene Angriffsvektoren erfolgen und läuft ebenfalls häufig automatisiert ab, da sich die Malware oft selbstständig verbreiten kann. Nicht selten merken die Besitzer der infizierten Systeme nicht einmal, dass ihre Systeme Teil eines Botnetzes sind, da entweder keine oder nur sehr subtile Anzeichen für die Infektion vorhanden sind. Im Falle von IoT-Geräten ist die Infektion besonders problematisch, da diese oft nicht über die notwendigen Sicherheitsmechanismen verfügen, um sich gegen Angriffe zu schützen oder diese zu erkennen.
3. **Attack**: In dieser Phase führen die Agenten den eigentlichen Angriff auf das Ziel durch, indem die Command-and-Control-Infrastruktur des Angreifers mit den notwendigen Informationen instruiert wird. Diese Informationen umfassen neben der Art des Angriffs auch die Dauer, Intensität und Ziele des Angriffs.

2.4.2 Arten von DDoS-Angriffen

Es existieren heute eine Vielzahl von DDoS-Angriffsmechanismen, die sich in ihrer Komplexität, Intensität und Zielsetzung unterscheiden. In diesem Abschnitt sollen einige der häufigsten Arten von DDoS-Angriffen vorgestellt und klassifiziert werden.

2.4.2.1 Ressourcenerschöpfungsangriffe

Ressourcenerschöpfungsangriffe zielen darauf ab, die verfügbaren Ressourcen eines Ziels zu überlasten, um es unbrauchbar zu machen. Dabei handelt es sich konkret um die Auslastung von Arbeitsspeicher und CPU eines Systems. Der Angreifer nutzt dabei bestimmte Eigenschaften von Netzwerkprotokollen aus oder sendet speziell präparierte bzw. fehlerhafte Pakete. Zunächst wird in Folgendem auf Protokoll-Angriffe eingegangen. [DM04; Mah+17]

- **TCP SYN Flood**: Bei einem SYN Flood-Angriff sendet der Angreifer eine große Anzahl von TCP-SYN-Paketen an das Ziel, ohne die Verbindung zu vervollständigen. Das Ziel antwortet mit einem SYN-ACK-Paket, um die Verbindung zu bestätigen, erhält jedoch nie eine Antwort vom Angreifer. Dadurch werden die Ressourcen des Ziels erschöpft, da es eine große Anzahl von halb geöffneten Verbindungen verwalten muss. Ähnliche Angriffe sind SYN-ACK- oder ACK-Floods, die auf dem gleichen Prinzip basieren.
- **TCP PUSH+ACK Flood**: Bei einem PUSH+ACK Flood-Angriff sendet der Angreifer eine große Anzahl von Paketen mit dem PUSH- und ACK-Flag an das Ziel. Das Ziel muss die Daten in den Paketen verarbeiten und die Verbindung bestätigen, was zu einer hohen Ressourcenauslastung führt, wodurch es zu Einschränkungen bei der Verarbeitung von legitimen Anfragen kommt.
- **HTTP Flood**: Bei einem HTTP Flood-Angriff sendet der Angreifer eine große Anzahl von HTTP-Anfragen an das Ziel, um die Webserver-Ressourcen zu überlasten. Dies kann durch das Senden von Anfragen an ressourcenintensive URLs, das Erzeugen von zufälligen Anfragen oder das Senden von Anfragen mit großen Headerfeldern erreicht werden. Beispielsweise kann mit einer hohen Anzahl von HTTP-GET-Requests die Bandbreite des Webservers erschöpft werden, wenn damit Downloads von großen Dateien initiiert werden.
- **Slow Request/Response Attack**: Eine besondere Art von ressourcenerschöpfenden Angriffen sind Slow Request/Response Attacks. Bei diesen Angriffen sendet der Angreifer Anfragen an das Ziel, die extrem langsam verarbeitet werden. Dies kann durch lange Wartezeiten zwischen den Paketen oder durch das Senden von vielen, sehr kleinen Anfragen erreicht werden. Das Ziel muss die Anfragen verarbeiten und die Verbindung offen halten, was unter Umständen die maximale Anzahl von Sockets oder erlaubten Sessions eines Servers beansprucht. Beispiele für solche Angriffe sind Slowloris, Slowpost oder Slowread.

Eine weitere Kategorie von Angriffsmethoden, welche auf die Ressourcenerschöpfung abzielen, umfasst das Senden von fehlerhaften Paketen. Diese Pakete führen bei der Verarbeitung auf dem Zielsystem zu Fehlern und im schlimmsten Fall zu Crashes. Es soll an dieser Stelle erwähnt werden, dass viele dieser Angriffe auf Schwachstellen in TCP/IP-Stacks veralteter Betriebssysteme abzielen, welche in modernen Implementierungen behoben wurden. Jedoch könnten sie im IoT-Bereich oder bei bestimmten Industrie-Anwendungen immer noch erfolgreich sein.

- **Ping Of Death**: Eine der bekanntesten Arten dieser Form von Angriffen trägt den dramatischen Namen *Ping Of Death*. Dabei sendet der Angreifer ein Paket an das Opfer, welches die maximal erlaubte Größe eines Pakets überschreitet. [Pin24]
- **Teardrop Attack**: Bei einem Teardrop-Angriff sendet der Angreifer fragmentierte Pakete mit sich überlappenden Offset-Feldern an das Ziel. Das Zielsystem kann die Pakete nicht korrekt reassemblieren und stürzt möglicherweise ab.
- **LAND Attack**: Bei einem LAND-Angriff sendet der Angreifer ein Paket an das Ziel, bei dem die Quell- und Ziel-IP-Adresse identisch sind. Die Folge war bei anfälligen Systemen oft ein Systemabsturz. [The24]

2.4.2.2 Bandbreitenererschöpfungsangriffe

Dieser Typ von DDoS-Angriff basiert auf dem simplen Prinzip, ein so großes Volumen an Datenverkehr zu erzeugen, dass die verfügbare Bandbreite des Ziels erschöpft wird. Dies kann durch das Senden von großen Datenpaketen, einer großen Anzahl von Paketen oder einer Kombination aus beidem erreicht werden [DM04; Mah+17]. Das Resultat dieser Angriffe ist, dass legitime Anfragen nicht mehr verarbeitet werden oder gar nicht erst beim Ziel ankommen. Besonders zustandslose Protokolle wie UDP oder ICMP sind anfällig für Bandbreitenerschöpfungsangriffe, da sie keine Mechanismen zur Verbindungskontrolle oder -verwaltung besitzen. Diese Angriffe sind schwer abzuwehren, da sie oft legitimen Traffic imitieren und die Quelle des Angriffs schwer zu identifizieren ist. Zudem können sie nicht durch simple Bugfixes eines Netzwerkprotokolls behoben werden, sondern erfordern spezielle Schutzmechanismen wie Firewalls oder DDoS-Mitigation-Systeme.

- **UDP Flood Attack**: Der Angreifer sendet eine große Anzahl von UDP-Paketen an einen bestimmten Port und damit Service des Ziels. Zielsysteme versuchen außerdem häufig, die Anwendung des Senders mittels ICMP zu identifizieren, was bei der Verwendung von gefälschten Quell-IP-Adressen den Angriff sogar noch verstärken kann.
- **ICMP/Ping Flood bzw. Smurf Attack**: Bei diesem Angriff werden ICMP-Pakete an die Broadcast-Adresse des Zielnetzwerks gesendet, was dazu führt, dass Pakete an alle Hosts innerhalb dieser Broadcast-Domäne gesendet werden. Die Hosts antworten mit einem Echo-Reply-Paket an die gefälschte Quell-IP-Adresse, was zu einer Überlastung des Ziels führt. Eine Variante dieses Angriffs ist der Smurf-Angriff, bei dem der Angreifer gefälschte ICMP-Pakete an die Broadcast-Adresse sendet, die die Quell-IP-Adresse des Opfers enthalten. Die Hosts in der Broadcast-Domäne senden dann ihre Echo-Reply-Pakete an das Opfer, was zu einer Überlastung führt. Heute sind Hosts daher häufig so konfiguriert, dass sie nicht mehr auf ICMP-Requests an der Broadcast-Adresse antworten, was zumindest die Amplifikation des Angriffs verhindert.

Das Prinzip der Amplifikation wird von Angreifern häufig genutzt, um die Effektivität eines DDoS-Angriffs zu erhöhen. Dabei werden *große* Antworten mit *kleinen* Anfragen provoziert und mittels IP-Spoofing der Quell-IP-Adresse an das Ziel geleitet [DM04; Mah+17]. Das Opfer erhält dann eine große Menge an Datenverkehr, ohne dass der Angreifer selbst eine große Bandbreite benötigt. Einige Beispiele für Angriffe, die sich dieses Prinzip zunutze machen, sind:

- **DNS Amplification Attack**: Bei einem DNS-Amplifikationsangriff sendet der Angreifer Anfragen an einen DNS-Server, wobei die Quell-IP-Adresse gefälscht wird, um die Antwort an das Opfer zu senden. Das DNS-Protokoll hat die Eigenschaft, dass die Antworten viel größer sind als die Anfragen, was zu der Amplifikation des Angriffs führt.
- **NTP Amplification Attack**: Dieser Angriff basiert im Grunde auf dem gleichen Prinzip. Der Angreifer fordert von NTP-Servern mittels `MON_GETLIST`-Befehl eine Liste der letzten 600 NTP-Queries. Diese vergleichsweise große Antwort wird via IP-Spoofing an das Opfer gesendet.

2.4.2.3 Zero-Day-Angriffe

Wie bei anderen Formen von Cyberattacken kann es sich auch bei DDoS-Angriffen um sogenannte Zero-Day-Angriffe handeln, bei denen eine Schwachstelle in einem System aus-

genutzt wird, bevor der Hersteller oder Betreiber des Systems von der Schwachstelle weiß oder bevor ein Patch oder eine Lösung verfügbar ist. Entsprechend sind Zero-Day-Angriffe besonders gefährlich, da sie oft schwer zu erkennen und zu stoppen sind.

Ein bekanntes Beispiel dafür ist die *HTTP/2 Rapid Reset Attack*, deren technischen Einzelheiten zuerst im Oktober 2023 von Cloudflare, Google und Amazon AWS veröffentlicht wurden [Clo23]. Diese Angriffsstrategie, welche zuvor für rekordbrechende DDoS-Angriffe auf Kunden von Cloudflare verantwortlich war, macht sich eine Schwachstelle in der Implementierung des HTTP/2-Protokolls sowie bestimmter Webserver zunutze [Goo24]. Das Ausmaß dieser Schwachstelle kann nicht unterschätzt werden, wenn man bedenkt, dass ca. 62 % des von Cloudflare erfassten Internet-Traffics das HTTP/2-Protokoll verwendet [Clo24].

3 Netzwerkbelastungssimulationen

Eine Netzwerkbelastungssimulation ist ein Verfahren oder eine Methode, mit der die Last (also der Datenverkehr) in einem Computernetzwerk unter kontrollierten Bedingungen nachgebildet wird. Ziel dieser Simulation ist es, die Leistungsfähigkeit und das Verhalten eines Netzwerks unter verschiedenen Belastungsbedingungen zu testen, ohne dass tatsächlich realer Datenverkehr generiert werden muss. Dabei können Netzwerkausfälle, Engpässe, Verzögerungen, Paketverluste und andere Netzwerkprobleme simuliert und analysiert werden. Diese Simulationen sind besonders nützlich, um Schwachstellen im Netzwerk zu identifizieren und zu beheben, bevor sie zu echten Problemen führen. [PBZ22]

3.1 Traffic Generation

Traffic Generation ist eng mit Netzwerkbelastungssimulationen verknüpft, da sie die kontrollierte Erzeugung von Datenverkehr in einem Netzwerk umfasst. Dieser Datenverkehr kann dazu verwendet werden, um die Leistungsfähigkeit und Funktionalität eines Netzwerks, eines Netzwerksystems oder eines Netzwerkdienstes zu testen, die Auswirkungen von Netzwerkänderungen zu analysieren oder um Sicherheitsmechanismen zu überprüfen [PBZ22]. Häufig soll der generierte Datenverkehr das realistische Verhalten von Nutzern oder Systemen in einem Netzwerk simulieren, indem bestimmte Eigenschaften wie Protokoll, Paketgröße, Paketrate oder Paketverteilung berücksichtigt werden. Traffic-Generatoren bzw. Network Traffic Generators (NTGs) sind daher ein wichtiges Werkzeug in der angewandten Forschung, Entwicklung und Administration von Netzwerken [ABG23]. Ein häufiger Anwendungsfall ist beispielsweise die Simulation von Angriffen, um die Resilienz eines Netzwerks gegenüber Cyber-Attacken zu testen. [Swa+20; Wan+11]

Die Popularität von Traffic-Generatoren hat in den letzten Jahren unter anderem durch die zunehmende Komplexität von Netzwerken sowie durch steigende Sicherheits- und Performanceanforderungen stark zugenommen [Kun+22]. Ein weiterer Treiber für die Verbreitung von NTGs sind neue Entwicklungen im Bereich der Netzwerkvirtualisierung und -automatisierung durch Technologien wie Software-Defined Networking (SDN) und Network Function Virtualization (NFV), die neue Anforderungen an die Test- und Validierungsprozesse von Netzwerken stellen [ABG23]. Die Notwendigkeit von NTGs ergibt sich zusätzlich aus der Problematik, dass echte Trafficdaten aus produktiven Netzwerken von entsprechenden Organisationen unter Verschluss gehalten werden und daher nur selten für Forschungs- und Entwicklungszwecke zur Verfügung stehen [SOG07]. Selbst wenn sogenannte Traces aus der *echten Welt* verfügbar sind, so sind diese nur bedingt repräsentativ und können nicht ohne Weiteres für Tests beliebiger Netzwerktopologien verwendet werden [ABG23]. Network

Traffic Generatoren sind vergleichsweise flexibler und können optimalerweise für eine Vielzahl unterschiedlicher Anwendungsfälle eingesetzt werden [BDP10].

Man unterscheidet grundsätzlich zwischen zwei Arten von Traffic-Generatoren: Software-basierte Traffic-Generatoren und Hardware-basierte Traffic-Generatoren. Beide Arten haben ihre Vor- und Nachteile, welche hier kurz erläutert werden sollen.

3.1.1 Software-basierte Traffic Generatoren

Software-basierte Traffic-Generatoren sind Software-Tools, die auf normaler Computerhardware laufen, um Netzwerkverkehr via Netzwerk-Interfaces zu generieren.

Diese Tools sind vergleichsweise kostengünstig oder sogar in Form von Open-Source-Software frei verfügbar. Sie zeichnen sich durch eine hohe Flexibilität aus, da sie in der Regel eine Vielzahl von Parametern für die Generierung von Traffic anbieten. Aufgrund dieser Eigenschaften eignen sie sich besonders für den Einsatz in Tests oder Experimenten mit komplexen und umfangreichen Traffic-Szenarien.

Herausforderungen und Probleme werden jedoch besonders im Hinblick auf Performance und Genauigkeit deutlich. Die Performance von Software-basierten Traffic-Generatoren ist in der Regel durch die Leistungsfähigkeit der Host-Hardware begrenzt [Emm+17]. Ein häufiger Ansatz ist daher horizontale Skalierung, also der Einsatz von mehreren Hosts, um die benötigte Performance zu erreichen [BDP10]. Jedoch ist dies nicht immer möglich oder wirtschaftlich und bringt die zusätzlichen Herausforderungen der Synchronisation und Koordination von verteilten Systemen mit sich. Die Genauigkeit von Software-basierten Traffic-Generatoren ist ebenfalls durch die Host-Hardware und -Software begrenzt. Besonders bei hohen Lasten können genaue Paket- und Bitraten nicht mehr garantiert werden, was einen großen Nachteil gegenüber Hardware-basierten Traffic-Generatoren darstellt [Emm+17]. Diese negative Eigenschaft, welche zur Verfälschung der Messergebnisse führen kann, wird auch als *Rate-Jitter* bezeichnet. Dieser Rate-Jitter hat direkte Auswirkungen auf die Performance des getesteten Netzwerks oder Systems, da er unter anderem höhere Latenzzeiten und Paketverluste verursachen kann.

Eine weitere Herausforderung ist die genaue Einhaltung von gewünschten Zeitintervallen zwischen Paketen, der sogenannten *Inter-Packet Time* (IPT), die die Charakteristik des generierten Traffics maßgeblich beeinflusst [MMS13]. Zuletzt muss die Tatsache betrachtet werden, dass Software-basierte Traffic-Generatoren in der Regel auf herkömmlichen Betriebssystemen wie z. B. Linux oder Windows laufen, die nicht für deterministische Echtzeitverarbeitung ausgelegt sind. Das bedeutet, dass Prozesse anderer Anwendungen sowie Prozesse oder Threads innerhalb des Traffic-Generators selbst (z. B. Logging) die Generierung von Traffic beeinflussen können [BDP10].

Neue Implementierungen von Software-basierten Traffic-Generatoren, wie z. B. *Moongen*, versuchen diese Probleme zu lösen, indem sie Technologien wie DPDK (Data Plane Development Kit) [DPD24] verwenden, um die Performance und Genauigkeit zu verbessern. DPDK ermöglicht es, den Netzwerk-Stack des OS-Kernels zu umgehen und so den Netzwerkverkehr direkt auf der Hardware-Ebene zu verarbeiten. Dies führt zu einer signifikanten Verbesserung der Performance und Genauigkeit von Software-basierten Traffic-Generatoren und verringert zudem die Hardwareanforderungen für die Generierung von hohen Datenraten

[Emm+15]. Nachteilig ist jedoch der relativ hohe Entwicklungsaufwand sowie der Verlust einiger Funktionen und Sicherheitsmechanismen des Betriebssystems. Sollen beispielsweise TCP-Verbindungen simuliert werden, so muss die TCP-Implementierung in der Software nachgebildet werden, was einen erheblichen Entwicklungsaufwand bedeutet [Gai20].

Zusammenfassend lässt sich sagen, dass Software-basierte Traffic-Generatoren eine Menge Vorteile bieten, jedoch muss sich der Anwender stets der genannten Limitationen bewusst sein und diese bei der Planung und Durchführung von Experimenten berücksichtigen.

Aufgrund der vergleichsweise besseren Zugänglichkeit in Bezug auf Einsatz und Entwicklung existieren heute viele Projekte und Tools im Bereich der Software-basierten Traffic-Generatoren. Daher soll an dieser Stelle auf einige Beispiele eingegangen werden, um einen Überblick über die Vielfalt der verfügbaren Tools und deren Eigenschaften zu geben.

3.1.1.1 Iperf3

Beliebtes Tool für Performance-Tests und Messungen in Netzwerken, welches TCP- und UDP-Datenverkehr generieren kann. Es bietet eine Vielzahl von Optionen zur Konfiguration von Datenraten, Paketgrößen und Protokolloptionen. Iperf3 ist Open-Source (BSD-Lizenz) und plattformübergreifend verfügbar [esn24]. Limitationen in Hinblick auf Datenraten und Durchsatz ergeben sich hauptsächlich durch den Netzwerk-Stack des Host-Systems. Daher sollten beispielsweise unter Linux einige Kernel-Parameter angepasst werden, um die maximale Performance zu erreichen. Seit 2023 unterstützt Iperf3 zudem Multi-Threading, was die Performance weiter verbessert.

3.1.1.2 Moongen

Moongen [Emm+15] ist ein script-basierter, flexibler Packet Generator, welcher für hohe Geschwindigkeiten entwickelt wurde und als Open-Source-Software (MIT-Lizenz) verfügbar ist [Emm24]. Für genaue Messungen und Datenraten verwendet Moongen hardware-basiertes Timestamping, was von den meisten modernen Netzwerkkarten unterstützt wird. Moongen erreicht diese Performance mittels einer speziellen Architektur, die auf dem Data Plane Development Kit (DPDK) basiert. Mit entsprechenden Netzwerkkarten und genügend CPU-Kernen kann Moongen Datenraten bis in den Bereich von 100 Gbit/s erreichen und ist damit auch für den Einsatz in heutigen High-Performance-Netzwerken geeignet.

3.1.1.3 Cisco TRex

Einen besonderen Platz unter den Software-basierten Traffic-Generatoren nimmt Cisco TRex ein. TRex ist ein extrem umfangreiches Open-Source-Tool (Apache-2.0-Lizenz) [cis24], das speziell für die Emulation von realistischem Netzwerkverkehr entwickelt wurde. Es unterstützt Datenverkehr von Layer-2 bis Layer-7 und kann Datenraten von bis zu 200 Gbit/s mit einem einzelnen Server erreichen.

Wie Moongen basiert TRex auf DPDK und nutzt Multi-Threading, um die Performance zu steigern, ist aber trotzdem in der Lage, Stateful Traffic zu generieren. Dies wird durch die Verwendung eines eigens entwickelten TCP-DPDK-Stacks im User-Space ermöglicht. Das Ergebnis ist der sogenannte *Advanced Stateful Mode* (ASTF), der die Emulation von komplexen Netzwerkprotokollen wie HTTP und HTTPS ermöglicht. Neben sehr weitreichenden Konfigurationsmöglichkeiten bietet TRex auch eine grafische Benutzeroberfläche (GUI) zur

einfacheren Bedienung. Die offizielle Dokumentation von TRex schildert auf beeindruckende Art und Weise, wie so selbst mit relativ limitierten Hardware-Ressourcen von vier CPU-Kernen und ca. 100 MB HTTP-Traffic mit Datenraten von 100 Gbit/s erreicht werden können [Com24].

3.1.1.4 Blitzping

Der letzte Software-basierte Packet Generator, der hier vorgestellt werden soll, ist ein relativ einfaches und sehr aktuelles Projekt namens *Blitzping*. Aufgrund seiner hohen Ähnlichkeit zu zwei anderen Projekten, *hping3* [San24] und nmaps *nping* [Npi24], kann es an dieser Stelle als Stellvertreter dieser Kategorie von simplen Netzwerktools verstanden werden.

Blitzping ist ein Open-Source-Tool (GPL-3.0-Lizenz) [Mem24], welches das Senden benutzerdefinierter IP-Pakete ermöglicht. Use-Cases umfassen simple Netzwerkperformance- und Sicherheitstests, wie beispielsweise das Testen von Firewallregeln oder Portscans. Blitzping unterscheidet sich von seinen Vorgängern durch seine bedeutend bessere Performance mit ca. 40-mal höheren Packet- und Bitraten [Mem24]. Generell lässt sich sagen, dass diese Programme aufgrund ihrer Limitationen nicht für High-Performance-Tests geeignet sind, jedoch für einfache Tests und Experimente eine gute Wahl darstellen und sich daher ihren Platz im Werkzeugkasten eines Netzwerkadministrators oder -entwicklers durchaus verdient haben.

3.1.1.5 Vergleich

Die folgende Tabelle fasst einige wichtige Eigenschaften der vorgestellten Software-basierten Traffic-Generatoren zusammen. Dabei wurde bewusst auf die Auflistung von Performanceeigenschaften wie Datenraten, Jitter und Latenz verzichtet, da verfügbare Benchmarks stark von der Host-Hardware und -Software abhängen und daher nur bedingt vergleichbar sind. Zudem existieren nur wenige Studien, welche die Performance von Software-basierten Traffic-Generatoren untereinander vergleichen, und selbst diese sind in ihrem Umfang beschränkt und vergleichsweise veraltet.

Funktion	iperf3	Moongen	Cisco TRex	Blitzping
TCP-Verbindungen	✓	–	✓	–
Traffic-Replay	–	✓	✓	–
L2-Customization	–	✓	✓	–
L3-Customization	begrenzt	✓	✓	begrenzt
Traffic-Randomization	–	✓	✓	nur Quell/Ziel-IP
HTTP-Traffic	–	–	✓	–
Metrik: Durchsatz	✓	✓	✓	✓
Metrik: Latenz	–	✓	✓	–
Metrik: Paketrate	–	✓	✓	–
Metrik: Pakete (Total)	✓	✓	✓	–

Tabelle 1: Vergleich der Software-basierte Traffic Generatoren

Insgesamt lässt sich sagen, dass Software-basierte Traffic-Generatoren eine Vielzahl von Funktionen und Eigenschaften bieten, die je nach Anwendungsfall und Anforderungen vari-

ieren. Die Auswahl des richtigen Tools hängt daher stark von den spezifischen Anforderungen und Zielen eines Experiments oder Testfalls ab.

Deutlich ist jedoch, dass Cisco TRex das umfangreichste und leistungsfähigste Tool in dieser Kategorie ist, das der Git-Historie nach auch kontinuierlich weiterentwickelt wird. Jedoch ist auch Moongen mit seiner hohen Flexibilität, seinem großen Funktionsumfang und seiner Performance ein sehr interessantes Tool, das trotz eines scheinbaren Mangels an Aktivität in der Community weiterhin eine gute Wahl für High-Performance-Tests darstellt. Beide Projekte verwenden das DPDK-Framework, um die Performance zu steigern, erfordern aber daher auch kompatible Netzwerkkarten und -treiber – ganz im Gegensatz zu Iperf3 und Blitzping, die auf herkömmlicher Netzwerk-Stack-Technologie basieren und auf quasi jedem modernen Linux-System ausführbar sind.

3.1.2 Hardware-basierte Traffic Generatoren

Bei Hardware-basierten Traffic-Generatoren handelt es sich um Geräte, die speziell für die Erzeugung von Netzwerkverkehr entwickelt wurden. Sie verfügen normalerweise über spezialisierte Hardware-Komponenten, die eine hohe Paketrate und -genauigkeit ermöglichen, sowie über proprietäre Firmware.

Die hohe Genauigkeit bei der Generierung von Traffic sowie beim Timestamping von Paketen macht Experimente und Messergebnisse reproduzierbar und verlässlich [BDP10]. Diese Eigenschaften sind, im Gegensatz zu Software-basierten Traffic-Generatoren, auch bei hohen Datenraten gegeben. Allerdings sind Hardware-basierte Traffic-Generatoren vergleichsweise weniger flexibel, und ihre Kontrolle über bestimmte Traffic-Parameter ist limitiert [Kun+22]. Zudem sind sie in der Anschaffung und im Unterhalt teurer als Software-basierte Traffic-Generatoren [LHM23]. Einige Beispiele für Hardware-basierte Traffic-Generatoren sind:

- **Spirent TestCenter**: Spirent TestCenter ist ein kommerzielles Produkt, das eine Vielzahl von Funktionen für die Erzeugung und Analyse von Netzwerkverkehr bietet. Die TestCenter-Hardwarefamilie umfasst Modelle für Layer-2- bis Layer-7-Traffic-Generation und -Analyse mit Datenraten bis zu 800 Gbit/s pro Port. [Spi24]
- **Keysight Network Test Hardware**: Keysight bietet ebenfalls eine breite Palette von Netzwerktestgeräten an, die für die Erzeugung und Analyse von Netzwerkverkehr verwendet werden können. Auch hier existieren Hardwarelösungen für die Erzeugung von Traffic auf Layer 2 bis Layer 7. Zudem sind modulare Systeme verfügbar, welche bis zu Datenraten von 3,2 Tbit/s und 5,1 Milliarden gleichzeitiger TCP-Verbindungen skalieren können. [Key24]

3.1.3 Kategorien von Traffic Generatoren

Neben der Implementierung von Traffic Generatoren, kann zusätzlich eine Kategorisierung nach Anwendungsfall und Anforderungen vorgenommen werden [ABG23]. Die folgenden Kategorien sind die häufigsten:

1. **Constant or Maximum Throughput Generators**: Erzeugen Pakete mit konstanter Rate oder maximal möglicher Rate. Beispiele: Iperf3, netperf.
2. **Application Level Synthetic Workload Generators**: Generieren Verkehr für bestimmte Anwendungen oder Protokolle. Beispiel: httperf.

3. **Trace File Replay Systems**: Spielen Pakete aus bestehenden Tracedateien ab. Beispiel: TCPreplay.
4. **Model-Based Traffic Generators**: Erzeugen Pakete basierend auf zufälligen Verteilungen der Zeitintervalle und Paketgrößen. Beispiel: Moongen.
5. **Trace Driven Model-Based Traffic Generators**: Nutzen Tracedateien oder Logdateien zur Erstellung von Modellen, die zur Paketerzeugung verwendet werden. Beispiele: harpoon, swing.
6. **Script Driven Traffic Generators**: Ermöglichen dynamische Modifikation der Paketinhalte durch komplexe Skripte. Beispiele: pktgen, Moongen.

Auf Grund der hohen Anzahl verschiedener Traffic Generatoren mit unterschiedlichen Eigenschaften und Funktionen, ist es sinnvoll die Auswahl des richtigen Tools anhand der spezifischen Anforderungen und Anwendungsfälle zu treffen [ABG23]. Die folgenden Kriterien können bei der Auswahl eines Traffic Generators hilfreich sein:

- **Anforderungen an Traffic**: Welche Art von Netzwerkverkehr soll generiert werden? Welche Datenraten und Paketgrößen sind erforderlich? Welche Protokolle und Anwendungen sollen unterstützt werden?
- **Verfügbarkeit**: Ist das Tool frei verfügbar oder kostenpflichtig? Kann das Tool ohne Einschränkungen in der gewünschten Testumgebung verwendet werden oder wird z.B. spezielle Hardware benötigt?
- **Auswahl nach Kategorie**: Welche Kategorie von Traffic Generator ist am besten geeignet für den geplanten Anwendungsfall? Welche Funktionen und Eigenschaften sind erforderlich?
- **Auswahl nach Funktionalität**: In Hinblick auf die zuvor definierten Anforderungen, welche Tools bieten die benötigten Funktionen und Eigenschaften? Welche Tools sind am besten geeignet für die geplante Testumgebung?

In der folgenden Tabelle werden die wichtigsten Unterschiede zwischen Software- und Hardware-basierten Traffic-Generatoren zusammengefasst:

Kategorie	Hardware-basierte NTGs	Software-basierte NTGs
Leistungsfähigkeit	Hohe Datenraten und präzise Zeitsteuerung	Durch Host-Hardware begrenzt, geringere Datenraten
Zuverlässigkeit	Stabile und konsistente Ergebnisse	Variiert je nach Host-Computer
Präzision	Hohe Präzision bei Zeitmessung und Analyse	Weniger präzise, abhängig von Host-Ressourcen
Kosten	Hohe Anschaffungs- und Unterhaltskosten	Kostengünstig, oft als Open-Source Software verfügbar
Flexibilität	Weniger flexibel, auf spezifische Hardware beschränkt	Hochgradig flexibel, leicht zu aktualisieren und modifizieren
Komplexität	Komplexe Einrichtung und Konfiguration	Einfache Installation und Nutzung
Ressourcenverbrauch	Spezialisierte Hardware, kein Ressourcen-Sharing	Teilt Ressourcen mit anderen Anwendungen
Einsatzgebiete	High-Performance-Netzwerke, Carrier-Grade, Sicherheit	Kleinere Netzwerke, Entwicklungsumgebungen, Forschung

Tabelle 2: Hardware-basierte Traffic Generatoren vs. Software-basierte Traffic Generatoren

3.1.4 Traffic Generation mit programmierbaren Netzwerkgeräten

Ein neuer Ansatz zur Traffic-Generierung ist die Verwendung von programmierbaren Netzwerkgeräten [KSK19], wie z. B. SmartNICs (Network Interface Cards) oder programmierbaren Switches [LHM23]. Die Idee ist es, die Vorteile von Hardware-basierten Traffic-Generatoren mit der Flexibilität und Programmierbarkeit von Software-basierten Traffic-Generatoren zu kombinieren, um eine sehr genaue, performante und anpassbare Traffic-Generierung zu ermöglichen. Zudem sind programmierbare Netzwerkgeräte in der Regel wesentlich günstiger als spezialisierte Hardware-Lösungen von Firmen wie Spirent, Keysight Technologies oder EXFO.

Erste Projekte dieser Art basieren auf NetFPGA-Systemen, einer Open-Source-Plattform für die Entwicklung von Prototypen von Netzwerkgeräten auf Basis von AMD (ehemals Xilinx) Virtex-FPGA-Technologie [Cov+09]. Mit der Einführung von P4 sowie der Verfügbarkeit von leistungsstarken programmierbaren Netzwerkprozessoren wie dem Intel Tofino war es naheliegend, dass auch Traffic-Generatoren auf Basis dieser Technologien entwickelt wurden. Einige Beispiele für solche Projekte sollen im Folgenden vorgestellt werden.

3.1.4.1 HyperTester

HyperTester ist der erste P4-basierte Traffic-Generator für die Intel Tofino-Familie von programmierbaren Switches [Zho+19]. Die Software läuft auf einem einzelnen Switch und verwendet die Tofino-ASICs, um Datenverkehr zu beschleunigen, zu replizieren und zu verarbeiten. Zudem wird die CPU des Switches als eine Art Co-Prozessor verwendet, um die Generierung von Traffic zu steuern und zu überwachen sowie die initialen Paketdaten zu erzeugen. Für die Definition von Tests stellt HyperTester eine einfache API namens *NTAPI* zur Verfügung.

HyperTester ermöglicht die Generierung von Layer-2- und zustandslosem Layer-3-Traffic mit anpassbaren Datenraten, Paketgrößen und Payloads. Der Traffic kann via Multicast auf mehrere Ports repliziert werden, um die erzeugte Gesamtlast zu erhöhen. Die Software bietet zudem eine einfache API zur Steuerung und Überwachung der Traffic-Generierung. HyperTester ist ein Open-Source-Projekt und steht auf GitHub zur Verfügung [hyp23], obwohl der Code quasi nicht dokumentiert ist und teilweise unvollständig erscheint.

3.1.4.2 P4STA

Einen anderen Ansatz verfolgt das P4STA-Projekt [Kun+20], welches eine umfassende, modulare Plattform für die automatisierte Test- und Analyse von Netzwerken bereitstellt. P4STA verfolgt den Ansatz, die eigentliche Erzeugung des Traffics auf herkömmlichen Servern mittels iPerf3 oder *Moongen* durchzuführen und die Messung mit P4-programmierbaren Geräten (*Stamper Targets*) vorzunehmen, wobei neben Tofino-Systemen auch Netronome SmartNICs unterstützt werden. Die Plattform bietet eine Vielzahl von Funktionen, darunter die automatisierte Konfiguration von Testfällen, die Durchführung von Tests, die Analyse von Ergebnissen und die Visualisierung von Messdaten.

P4STA ist ein Open-Source-Projekt und ebenfalls auf GitHub verfügbar [Kun24]. Die verschiedenen Systeme des Test-Setups können über einen zentralen Management-Server mittels CLI oder WebUI gesteuert werden. Diese Kombination aus Software-basierten Traffic-

Generatoren und programmierbaren Netzwerkgeräten ermöglicht eine sehr flexible und leistungsstarke Testumgebung für die Entwicklung und Validierung von Netzwerken.

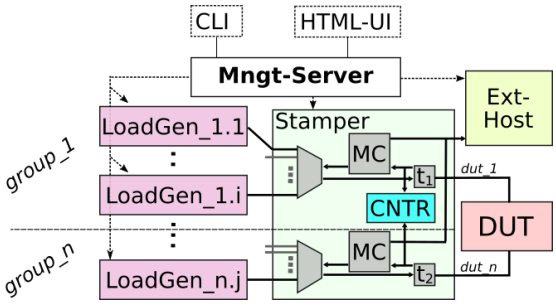

Abbildung 15: Systemdesign: P4STA, Quelle: https://github.com/ralfkundel/P4STA

3.1.4.3 P4TG

Dieses Projekt [LHM23] ähnelt HyperTester, verwendet aber den internen Packet Generator der TNA für die Erzeugung von Traffic. Zudem sind die Messdaten, die von P4TG gesammelt werden, wesentlich umfangreicher und detaillierter als bei HyperTester. P4TG nutzt neben Registern für das Speichern von Metriken zusätzlich das Prinzip von *Monitoring-Frames*, welche regelmäßig zusätzlich zum Test-Traffic generiert und an den Controller gesendet werden.

Der Controller verfügt zusätzlich über eine REST-API und ein Web-Interface zur Steuerung und Überwachung der Traffic-Generierung. Es existiert mittlerweile ein 2.0 Release, welches die Funktionalität erweitert und die Control Plane komplett in der Programmiersprache Rust implementiert. Generell macht die Software einen vergleichsweise sehr hochwertigen und vollständigen Eindruck. P4TG ist ebenfalls ein Open-Source-Projekt (Apache-2.0-Lizenz) und auf GitHub verfügbar [uni24].

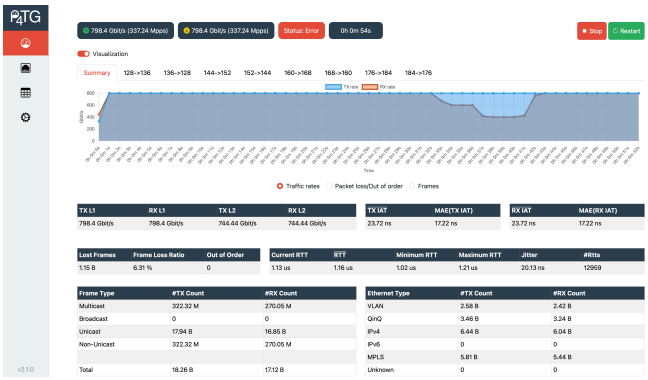

Abbildung 16: Screenshot: P4TG, Quelle: https://github.com/uni-tue-kn/P4TG

3.2 Simulation von DDoS-Angriffen

Traffic Generation ist nicht nur ein Werkzeug zur Analyse der allgemeinen Leistungsfähigkeit und Funktionalität eines Netzwerks, sondern auch ein entscheidendes Mittel, um gezielte Stress- und Sicherheitstests durchzuführen. Einer der wichtigsten Anwendungsfälle dabei ist die Simulation von Distributed Denial-of-Service (DDoS)-Angriffen [Ali23; Wan+11]. Diese Art der Simulation nutzt gezielt Traffic Generatoren, um realistische Angriffsvektoren nachzubilden und die Reaktionsfähigkeit eines Netzwerks auf extreme Lastspitzen zu testen. Durch die kontrollierte Erzeugung von großvolumigem, bösartigem Datenverkehr lassen sich

wertvolle Erkenntnisse über die Belastbarkeit und Sicherheit von Infrastrukturen gewinnen mit deren Hilfe effektive Gegenmaßnahmen entwickelt werden können. Dabei geht es nicht nur darum, Abwehrmechanismen auf ihre Wirksamkeit zu überprüfen, sondern auch die potenziellen Auswirkungen zu analysieren, falls diese Mechanismen versagen. Es ist daher durchaus sinnvoll, bestimmte Tests an Systemen durchzuführen, ohne dass beispielsweise Firewalls oder DDoS-Mitigation-Systeme aktiviert sind.

Diese Simulationen sollten unter kontrollierten Bedingungen und vorzugsweise in einer isolierten Testumgebung durchgeführt werden, um Schäden an Produktivsystemen zu vermeiden. Durch den Einsatz eines Testsystems lassen sich zudem Ausfallzeiten minimieren und die Auswirkungen auf legitime Benutzer erheblich reduzieren.

3.2.1 Entwicklung eines Testszenarios

Nachdem die Notwendigkeit und der Zweck einer DDoS-Simulation festgelegt wurden, beginnt die Planung und Entwicklung eines detaillierten Testszenarios. Der erste Schritt besteht darin, das Ziel der Angriffe zu definieren und zu entscheiden, welche Art von Angriff simuliert werden soll. Dabei können zuvor beschriebene Angriffsmethoden oder eine Kombination davon gewählt werden, abhängig von den Zielen des Tests und den spezifischen Anforderungen des Systems.

- Soll die Bandbreite des Ziels überlastet werden? **Bandbreitenerschöpfungsangriffe**.
- Sollen die Systemressourcen des Ziels überlastet werden? **Ressourcenerschöpfungsangriffe**.
- Soll eine bestimmte Anwendung oder ein Protokoll angegriffen werden? Auswahl einer bestimmten Angriffsmethode, die auf das Ziel zugeschnitten ist.

Insbesondere bei Angriffen auf Anwendungen ist es ratsam, zunächst Tests durchzuführen, um die am stärksten verwundbaren Bereiche der Anwendung zu identifizieren. Hierbei können Fehler, langsame Reaktionen oder ein erhöhter Speicherverbrauch, selbst wenn dieser nur geringfügig ist, als Indikatoren für Schwachstellen dienen. Ein typisches Beispiel für ein Testszenario wäre das Flooding eines Zielsystems mit spezifischen Nachrichten, wie etwa POST-Anfragen bei einer HTTP-Endpoint. Es ist wichtig, dass die Angriffe zielgerichtet erfolgen, um die Funktionalitäten der angegriffenen Anwendungen vollständig zu testen.

Ein weiterer wichtiger Punkt bei der Entwicklung eines Testszenarios ist die Auswahl des richtigen Tools [Gre20], welches als Angriffswerkzeug verwendet wird. Welche Anforderungen hier an ein solches Tool gestellt werden können, soll im nächsten Abschnitt erörtert werden. Im Zusammenhang damit sollten auch folgende Fragen beantwortet werden:

- Wie wird das Tool konfiguriert und verteilt? Hier können existierende Konfigurationsmanagement-Tools hilfreich sein.
- Wie wird der Angriff gestartet und gestoppt? Hierbei ist es wichtig, dass der Angriff kontrolliert und überwacht werden kann.
- Wie wird der Datenverkehr analysiert und die Auswirkungen des Angriffs gemessen? Hierbei können Monitoring-Tools oder spezielle Analyse-Tools zum Einsatz kommen [Gre20].

Schlussendlich sollte betont werden, dass beteiligte Personen und möglicherweise betroffene Systeme identifiziert werden müssen. Ingenieure bzw. Entwickler und weitere relevante Mitarbeiter müssen in den Testprozess einbezogen werden, insbesondere wenn Tests in einer Live-Umgebung durchgeführt werden. Nur so kann eine sorgfältige Planung des Tests gewährleistet werden.

3.2.2 Durchführung der DDoS-Simulationen

Wie bereits erwähnt, ist es entscheidend, das Zielsystem umfangreich zu überwachen, um den Angriff sinnvoll auswerten zu können. Hierbei können Monitoring-Tools, Log-Dateien oder spezielle Analyse-Tools zum Einsatz kommen. Folgende Metriken sind besonders relevant [Gre20; Lam24]:

- **Bandbreitennutzung**: Überwachung des Datenverkehrs, um sicherzustellen, dass keine unvorhergesehenen Überlastungen auftreten.
- **Anwendungen**: Beobachtung der Leistung der angegriffenen Anwendungen, um Anzeichen von Überlastung oder Fehlfunktionen zu erkennen.
- **Systemressourcen**: Überwachung der CPU-, Speicher- und Netzwerkauslastung der Zielsysteme.
- **Abhängigkeiten**: Überprüfung von Datenbanken und anderen kritischen Diensten, die für den Betrieb der Zielanwendung notwendig sind.

Je nach Szenario und Zielsystem ist die Echtzeit-Überwachung dieser Daten unerlässlich, um schnell auf unerwartete Ereignisse reagieren zu können und den Test im Notfall zu stoppen. Während der Simulation sollten die Ergebnisse kontinuierlich analysiert und dokumentiert werden, je nach Auswahl des Monitoring-Systems geschieht dies optimalerweise bereits automatisch, beispielsweise in Form von gespeicherten Zeitreihen der relevanten Messwerte [Gre20].

Nach Abschluss der Simulation stehen in der Regel verschiedene Befunde zur Verfügung. Diese müssen sorgfältig ausgewertet werden, um die zugrunde liegenden Ursachen zu identifizieren. Diese Root-Cause-Analyse sollte strukturiert durchgeführt werden, indem Anwendungen debuggt, Logs überprüft und gegebenenfalls Retests durchgeführt werden. Erst nach einer gründlichen Bewertung sollten Lösungen oder Gegenmaßnahmen entwickelt werden, welche die identifizierten Probleme gezielt adressieren.

3.2.3 Anforderungen an ein System für DDoS-Simulationen

Für die praktische Durchführung solcher Tests bietet sich die Verwendung von Traffic-Generatoren an, die in der Lage sind, DDoS-ähnlichen Verkehr zu erzeugen. Dabei ist es wichtig, dass die generierten Datenverkehrsmuster und -eigenschaften den tatsächlichen DDoS-Angriffen so nahe wie möglich kommen, um realistische Testergebnisse zu erhalten. In den vorherigen Abschnitten wurden bereits einige Traffic-Generatoren vorgestellt und es wurden verschiedene DDoS-Angriffsmethoden beschrieben. Zusammenfassend lassen sich auf Basis dieser Informationen folgende Anforderungen an einen Traffic-Generator für die Simulation von DDoS-Angriffen formulieren:

1. Die Erzeugung von hohem Datenverkehrsvolumen: Der Traffic-Generator sollte in der Lage sein, eine große Anzahl von Paketen pro Sekunde zu erzeugen, um die Bandbreite des Ziels zu überlasten.
2. Die Erzeugung von spezifischen Paketen: Der Traffic-Generator sollte in der Lage sein, spezifische Pakete zu erzeugen, die für typische DDoS-Angriffe charakteristisch sind, wie z. B. TCP-SYN-, UDP- oder ICMP-Pakete.
3. Die Erzeugung von variablen Paketgrößen: Der Traffic-Generator sollte in der Lage sein, Pakete mit variablen Größen zu erzeugen, um die Verarbeitung von Paketen mit unterschiedlichen Größen zu simulieren.
4. IP-Spoofing: Der Traffic-Generator sollte in der Lage sein, die Quell- und Ziel-IP-Adressen der Pakete zu fälschen, um die Herkunft des Angriffs zu verschleiern, die Amplifikation des Angriffs zu ermöglichen oder ein gesamtes Netzwerk anzugreifen.
5. Metriken und Statistiken: Der Traffic-Generator sollte in der Lage sein, Metriken und Statistiken über den erzeugten Datenverkehr zu sammeln, um die Auswirkungen des Angriffs zu analysieren und Grenzen des Zielsystems zu identifizieren.

Ebenfalls sind optimalerweise folgende nicht-funktionalen Anforderungen und Benutzeranforderungen zu berücksichtigen:

1. **Kosten und Verfügbarkeit**: Der Traffic-Generator sollte kostengünstig und einfach verfügbar sein, um eine breite Anwendung zu ermöglichen.
2. **Benutzerfreundlichkeit**: Der Traffic-Generator sollte auch von Netzwerkadministratoren und -entwicklern ohne spezielle Erfahrung mit der Implementierung von Traffic-Generatoren einfach zu bedienen sein.
3. **Flexibilität und Anpassbarkeit**: Der Traffic-Generator sollte flexibel und anpassbar sein, um verschiedene Arten von DDoS-Angriffen zu simulieren und unterschiedliche Testumgebungen zu unterstützen.

4 Design und Implementierung

Nachdem die notwendigen Grundlagen zur Umsetzung dieses Projekts erörtert und die Anforderungen an den Traffic-Generator für die Simulation von DDoS-Angriffen formuliert wurden, soll in diesem Abschnitt das Design und die Implementierung eines solchen Traffic-Generators vorgestellt werden. Dabei wird auf die Architektur, die Funktionalität und die Implementierungsdetails des Traffic-Generators eingegangen.

Bei dem entwickelten Prototypen handelt es sich um einen Traffic-Generator, der auf Basis der Tofino Barefoot-Architektur implementiert wurde. Die Tofino-Familie von programmierbaren Switches bietet integrierte Funktionen zur Generierung von Paketen mit sehr hohen Datenraten von bis zu 100 Gb/s pro physischem Port. Diese Geschwindigkeiten sind möglich, da die Erzeugung und das Forwarding von Datenpaketen in Hardware auf der Data Plane erfolgt. Wie bereits erwähnt, ist die Data Plane eines programmierbaren Switches außerdem *Software-defined*, also mit P4 programmierbar. Diese Eigenschaft bietet die notwendige Flexibilität, um die Funktionalität des Switches zu erweitern und spezifische Anforderungen zu erfüllen.

Für die Definition von Paket- und Trafficmustern wird ein Controller benötigt, der die P4-Programme auf den Switch lädt und die Traffic-Generierung steuert. Dieser Controller kann entweder lokal auf dem Switch laufen oder auf einem externen Server. Da der Controller quasi keine expliziten Performance-Anforderungen erfüllen muss, kann dieser in Form eines Python-Programms auf dem Managementsystem des Testers gestartet werden. Diese Architektur vereinfacht das Setup des Systems und ermöglicht eine einfache Integration in bestehende Testumgebungen. Eine einfache Remote-Steuerung, beispielsweise über gRPC-Calls, war eines der wichtigsten Designziele dieses Prototyps, da es im Grunde zwei große Vorteile bietet:

1. Tests können einfach ausgeführt und in entsprechende Automatisierungsprozesse integriert werden.
2. Die Entwicklung des Traffic-Generators wird vereinfacht und stark beschleunigt, wenn manuelle Konfigurations- und Kompilierungsprozesse auf dem Switch vermieden werden. Die Entwicklung kann zum Großteil direkt auf dem Managementsystem erfolgen, wo ggf. viele Entwicklungs-Tools und Bibliotheken zur Verfügung stehen. Diese solide Basis bietet die Möglichkeit, den Traffic-Generator in Zukunft zu erweitern und anzupassen.

Bei der Implementierung wurde sich an den beiden Projekten P4TG [uni24] und PIPO-TG [Fil24] orientiert, da diese Projekte bereits einige der Anforderungen an den Traffic-Generator erfüllen und eine gute Grundlage für die Entwicklung eines eigenen Prototyps bieten.

Beide Projekte beinhalten zudem viele Code-Beispiele für die Entwicklung einer Control Plane in Python, was bei der limitierten Dokumentation der Tofino-Entwicklungsumgebung sehr hilfreich ist. Falls Code dieses Projektes mit dem Code eines dieser Projekte übereinstimmen sollte, dann wurde er vermutlich von dort übernommen.

Der gesamte Code dieses Projekts ist zudem in einer Repository[1] des HTW-GitLab für Mitglieder der Hochschule verfügbar. [Pas24]

4.1 Technologien und Tools

Der Prototyp des Traffic-Generators verwendet für die Umsetzung der definierten Anforderungen eine Reihe unterschiedlicher Technologien und Konzepte. Aufgrund der damit inhärenten Komplexität des Gesamtsystems wurde auf das umfangreiche Ökosystem der Programmiersprache Python zurückgegriffen. Python bietet eine Vielzahl von Bibliotheken und Frameworks, die für die Entwicklung von Netzwerkanwendungen und -tools geeignet sind. Die folgenden Third-Party-Bibliotheken und Frameworks wurden für die Implementierung des Traffic-Generators verwendet:

- **bfrt-helper**: Dieses Python-Modul wurde von der Firma *APS Networks* entwickelt und bietet eine einfache Schnittstelle zur Steuerung von Tofino-Switches über das Barefoot Runtime Interface. Das Modul entspricht demnach einer Art gRPC-API-Wrapper, der die Komplexität des BRI abstrahiert und es dem Benutzer ermöglicht, die Data Plane des Switches einfach zu konfigurieren und zu steuern. Das Python-Modul ist Open-Source (GPL-3.0-Lizenz) und auf GitHub verfügbar [APS24].
- **grpc**: Die offizielle Implementierung des gRPC-Frameworks für Python. Die Bibliothek ist eine notwendige Abhängigkeit für die Verwendung von bfrt-helper, da die Kommunikation mit dem Switch über das gRPC-Protokoll erfolgt. Das Modul ist Open-Source (Apache-2.0-Lizenz) und auf GitHub verfügbar [grp24].
- **scapy**: Scapy ist ein mächtiges Python-Modul, das die Erzeugung und Manipulation von Netzwerkpaketen ermöglicht. Mit Scapy können komplexe Paketstrukturen definiert, Pakete generiert und analysiert sowie Netzwerkanalysen durchgeführt werden. Das Modul ist Open-Source (GPL-2.0-Lizenz) und auf GitHub verfügbar [sec24]. Scapy kann daher für die Definition von Packet-Templates verwendet werden und bietet die notwendige Flexibilität, um verschiedene Arten von DDoS-Angriffen zu simulieren.
- **paramiko**: Für die Interaktion mit dem Switch auf Betriebssystem-Ebene wird das SSH-Protokoll verwendet. Mit Paramiko können SSH-Verbindungen aufgebaut, Befehle auf entfernten Hosts ausgeführt und Dateien übertragen werden. Das Modul ist Open-Source (LGPL-2.1-Lizenz) und auf GitHub verfügbar [par24].
- **jinja**: Jinja ist eine Template-Engine für Python, mit der Konfigurationsdateien und Code für den Switch dynamisch generiert werden können. Das Modul ist Open-Source (BSD-3-Clause-Lizenz) und auf GitHub verfügbar.
- **tabulate**: Tabulate ist ein einfaches Python-Modul, das die Ausgabe von ordentlich formatierten Tabellen in der Konsole ermöglicht. Das Modul ist Open-Source (MIT-Lizenz) und auf GitHub verfügbar [Ast24].

[1] https://gitlab.rz.htw-berlin.de/s0558899/tofino-traffic-generator; Sollte es Probleme beim Zugriff geben, kann folgender Access Token verwendet werden: `8VoCMUARShr8D7bxwMTX`

Diese Bibliotheken und Frameworks sind besonders für die Implementierung der Control Plane des Traffic-Generators relevant. Die Data Plane des Traffic-Generators wird hingegen in der P4-Sprache entwickelt, wobei das Barefoot Software Development Environment eine wichtige Voraussetzung darstellt. Diese Entwicklungsumgebung bringt die notwendige Toolchain mit sich, die für die Entwicklung und Kompilierung von P4-Programmen für die Tofino-Architektur unerlässlich ist. Weitere wichtige Komponenten des Barefoot Software Development Environment sind:

- **tofino-model**: Das Tofino-Modell ist eine virtuelle Instanz des Tofino-Switches, die auf einem x86-System ausgeführt wird. Das Tofino-Modell ermöglicht die Entwicklung und das Testen von P4-Programmen auf einem virtuellen Switch, ohne dass ein physischer Tofino-Switch benötigt wird. Die BF-SDE umfasst zusätzlich Skripte für die einfache Ausführung des Modells und das Binden von virtuellen Netzwerkinterfaces an die virtuellen Ports des Switches.
- **bf_switchd**: Der eigentliche Switch-Daemon, der auf dem Tofino-Switch läuft und die Data Plane steuert sowie das Barefoot Runtime Interface bereitstellt. Der Service kann in einem interaktiven Modus gestartet werden, um den Switch mit Befehlen in einem Command-Line-Interface (CLI) zu konfigurieren.
- **bfshell**: Dieses interaktive CLI-Tool für die Interaktion mit dem Switch (bf_switchd) stellt ein Frontend für die Management-Plane des Switches dar. Neben der allgemeinen Konfiguration des Switches ist die bfshell auch für Debugging-Zwecke und die Überwachung der Data Plane essenziell.
- **bf-p4c**: Der P4-Compiler, der P4-Programme in ausführbaren Code für den Tofino-Switch kompiliert. Der Output des Compilers umfasst neben der ausführbaren Binary auch die Runtime-API-Definitionen.

4.2 Aufbau einer virtuellen Entwicklungsumgebung

Um einen effektiven Entwicklungsprozess zu gewährleisten, wurde eine virtuelle Testumgebung auf Basis des Tofino-Modells aufgebaut. Die Testumgebung ist reproduzierbar und bietet zudem eine Snapshot-Funktion, um den Zustand des Systems zu speichern und bei Bedarf wiederherzustellen. Das hat den Vorteil, dass Entwickler und Tester jederzeit auf eine saubere und konsistente Umgebung zurückgreifen können und nicht konstant auf physische Hardware angewiesen sind. Ebenfalls wird so das Risiko minimiert, die recht aufwendige Konfiguration eines Tofino-Switches zu verlieren oder zu beschädigen. Auch hier existiert praktischerweise bereits ein existierendes Projekt, welches diesen Schritt stark vereinfacht. Die Entwickler der bfrt-helper-Bibliothek bei APS Networks stellen Konfigurationsdateien bereit, die es ermöglichen, die Einrichtung einer virtuellen Testumgebung für das Tofino-Modell zu automatisieren. Diese Dateien können gemeinsam mit der Software Vagrant verwendet werden, um die Testumgebung zu provisionieren.

Bei Vagrant [Has24] handelt es sich um Software der Firma HashiCorp für die Erstellung und Verwaltung von virtuellen Entwicklungsumgebungen. Der Entwickler definiert die Entwicklungsumgebung textuell in einem sogenannten *Vagrantfile*, das die notwendigen Schritte zur Einrichtung beschreibt. Vagrant nutzt dabei existierende Virtualisierungsplattformen, wie VirtualBox oder VMware, um die Umgebung zu erstellen und zu verwalten. Folgende Schritte sind notwendig, um die virtuelle Testumgebung für das Tofino-Modell einzurichten:

1. Installation von Vagrant und einer Virtualisierungsplattform. Für dieses Projekt wird VirtualBox als Hypervisor verwendet.
2. Klonen des GitHub-Repositories von APS Networks, das das Vagrantfile für das Tofino-Modell enthält.
3. Anpassung des Vagrantfiles, um die gewünschten Konfigurationen für die virtuelle Umgebung festzulegen. Es wurde folgende virtuelle Maschine definiert:
   - OS: Ubuntu LTS 20.04 [Can24]
   - CPU: 8 vCPUs
   - RAM: 16 GB
   - Portweiterleitung zum Host: 2222 (SSH), 50052 (gRPC), 4000 (HTTP)
4. Die virtuelle Umgebung wird mit dem Befehl `vagrant up` gestartet. Vagrant lädt daraufhin automatisch das gewünschte Ubuntu-Image herunter, erstellt die virtuelle Maschine und führt das Setup-Skript aus.

Neben den oben genannten Grundeinstellungen der VM enthält das Vagrantfile noch weitere Konfigurationsschritte, die für die Einrichtung des Tofino-Modells notwendig sind. Dabei handelt es sich um die Installation und Einrichtung der Barefoot Software Development Environment (BF-SDE) sowie um die Installation diverser Pakete, Tools und Abhängigkeiten, die die Entwicklung vereinfachen (z.B. tcpdump [Tcp24]). Es sei zu erwähnen, dass einige der Skripte und das Vagrantfile selbst leicht angepasst werden mussten, um die Kompatibilität mit neueren Versionen verschiedener Software-Komponenten zu gewährleisten. Die auf GitHub verfügbaren Konfigurationsdateien und Skripte waren zum Zeitpunkt der Entwicklung etwas veraltet und mit der verfügbaren BF-SDE-Version nicht mehr kompatibel.

Der Zugriff auf die Testumgebung erfolgt nach Start der VM wie gewohnt über SSH, und es können weitere Ports und damit Services via Port Forwarding auf den Host weitergeleitet werden. Die Netzwerkkonfiguration der VM verwendet Network Address Translation (NAT), um die Verbindung zum Internet herzustellen und den Zugriff auf externe Ressourcen via Netzwerk-Ports der Host-Maschine zu ermöglichen. Da ein Traffic-Generator im Fehlerfall durchaus Ausfälle in einem Netzwerk bewirken kann, ist es wichtig, dass die virtuelle Testumgebung auf diese Art und Weise vom Netzwerk des Hostsystems isoliert wird. Optimalerweise sollte der erzeugte Traffic das interne Netzwerk der VM nicht verlassen, damit keine ungewollten Auswirkungen auf andere Systeme im lokalen Netzwerk oder sogar im Internet entstehen.

Um Pakete zum Tofino-Modell zu senden oder Pakete von diesem zu empfangen, werden virtuelle Netzwerkinterfaces benötigt. Diese Interfaces, auch *veth*-Interfaces genannt, verbinden die virtuellen Ports des Tofino-Modell-Switches mit Interfaces der virtuellen Maschine. Es ist auch möglich, virtuelle Switch-Ports direkt mit dem *externen Netzwerk* zu verbinden, falls das gewünscht ist. Dafür muss die VM mit Netzwerk-Interfaces im Bridge-Modus konfiguriert werden.

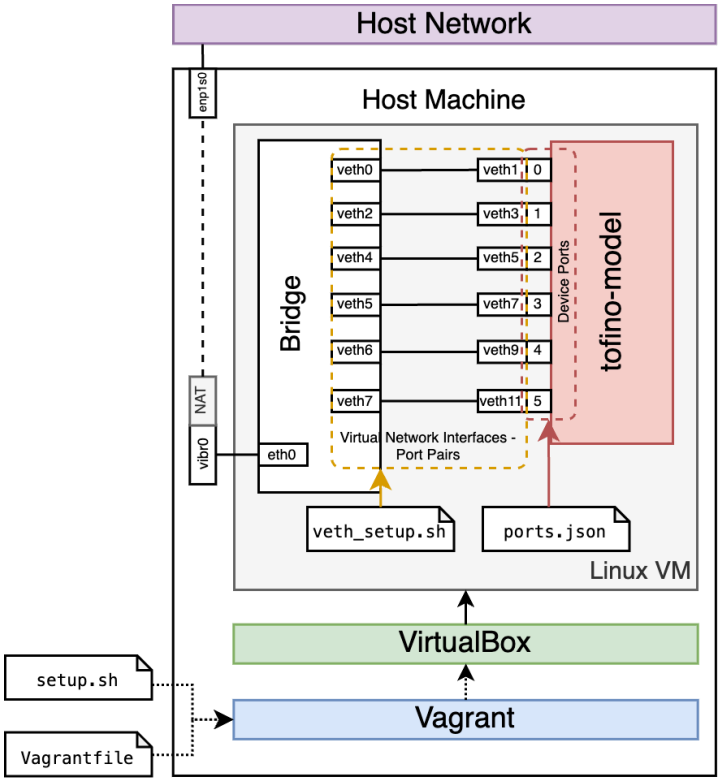

Abbildung 17: Gesamtbersicht: Virtuelle Entwicklungsumgebung

Praktischerweise stellen die BF-SDE sowie das Projekt Tofino-Model-Vagrant von APS Networks einige Skripte bereit, um eine beliebige Anzahl von virtuellen Ports zu erstellen und mit dem Tofino-Modell zu verbinden. Außerdem existieren auch Skripte, um diese Interfaces wieder zu entfernen. Damit das Tofino-Modell die virtuellen Ports verwenden kann, muss beim Start des Modells neben dem gewünschten P4-Programm auch eine Portkonfiguration im JSON-Format übergeben werden. Der Prototyp des Traffic-Generators übernimmt bei Angabe einer virtuellen Umgebung die Erstellung und Konfiguration sämtlicher virtueller Ports automatisch.

Da die Ports als virtuelle Netzwerkinterfaces auf dem Host angelegt werden, können diese auch von anderen Tools und Anwendungen verwendet werden. So ist es beispielsweise möglich, den generierten Traffic mit einem Netzwerkanalysetool wie Wireshark oder tcpdump zu überwachen und zu untersuchen.

4.3 Systemarchitektur

Folgendes Diagramm zeigt eine Gesamtübersicht über das Design des Traffic-Generators. Der Benutzer definiert den Testfall in einem Skript, welches auf Basis des Inputs P4-Code und einige Konfigurationsdateien generiert. Diese Dateien werden auf den Switch übertragen, wobei es sich entweder um einen physischen Tofino-Switch oder um eine virtuelle Instanz (Tofino-Modell) handeln kann. Anschließend wird der P4-Code auf dem Switch kompiliert und ausgeführt. Das hat den Vorteil, dass das Barefoot Software Development Environment (BF-SDE) nicht auf dem Managementsystem installiert werden muss.

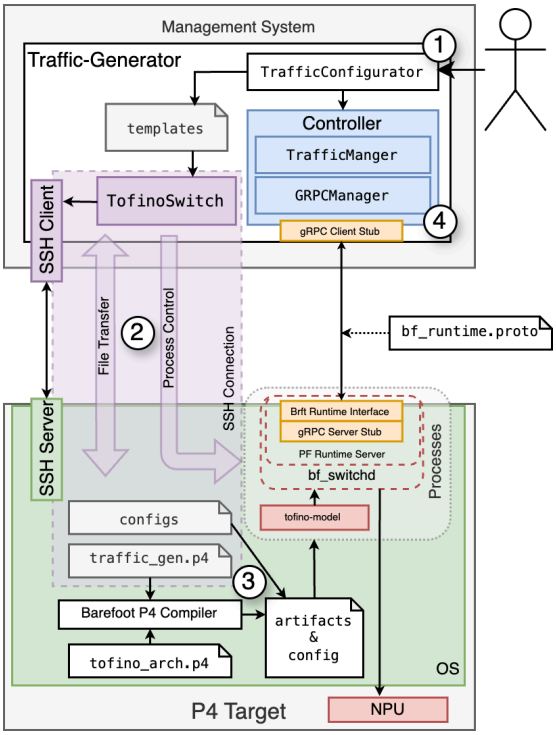

Abbildung 18: Gesamtbersicht: Traffic-Generator Prototyp

War der Kompilierungsprozess erfolgreich, erfolgt die weitere Steuerung der Data Plane mittels Barefoot Runtime Interface. Der Controller auf dem Managementsystem startet und konfiguriert nun den Traffic Generator, welcher die gewünschten Datenpakete generiert und an das Ziel sendet. Metriken und Statistiken werden vom Controller gesammelt und analysiert.

4.3.1 Data Plane

Die Tofino Native Architecture verfügt über einen integrierten Packet-Generator für jede der zwei oder vier verfügbaren Pipes. Diese Generatoren sind mit bestimmten internen Ports verbunden. Jeder Generator kann acht voneinander unabhängige Streams (auch als *Applications* bezeichnet) generieren, wobei jeder Stream separat konfiguriert werden kann [Int21]. Eine Application kann in vier verschiedenen Modi betrieben werden, die das Event bestimmen, durch das die Generierung gestartet wird.

1. **One-time Timer**: Die Application startet zu Zeitpunkt T und generiert B Batches von N Paketen.
2. **Periodic Timer**: Die Application generiert alle T Zeiteinheiten B Batches von N Paketen.
3. **Port Down**: Die Application generiert B Batches von N Paketen, wenn innerhalb derselben Pipe ein Port von Zustand *up* auf *down* wechselt.
4. **Packet Recirculation**: Die Application generiert B Batches von N Paketen, wenn ein Paket innerhalb des Generation-Ports rezirkuliert wird und ein bestimmtes Byte-Muster enthält, welches durch eine Bitmaske definiert wird.

Pakete, die von einem integrierten Packet-Generator erzeugt werden, enthalten einen 6-Byte Packet-Gen-Header, der je nach Modus verschiedene Informationen enthält. Allen Modi sind

jedoch die folgenden Felder gemein: die Pipe-ID, die Application-ID, die Stream-ID und die Packet-ID. Eine weitere wichtige Eigenschaft des Packet-Generators ist der 16-KByte große Paket-Buffer. Dieser Buffer wird von allen Applications geteilt und kann von der Control Plane konfiguriert werden.

Der hier vorgestellte Prototyp verwendet diesen Buffer zur Speicherung des initialen Template-Pakets, welches vom Benutzer mit Scapy definiert wird. Was die verfügbaren Modi angeht, so unterstützt der Prototyp lediglich den Periodic Timer-Modus, da dieser relativ vielfältig einsetzbar ist und die meisten Anforderungen an einen Traffic-Generator abdeckt.

Die Verarbeitungslogik der Datenpakete erfolgt auf der Data Plane des Switches, welche in P4 definiert wird. Für den Anwendungsfall eines Traffic-Generators sind dabei im Grunde nur wenige Verarbeitungsschritte notwendig, daher ist die Forwarding-Pipeline relativ simpel gehalten.

- **Ingress Parser**: Die Headerfelder der eingehenden Pakete werden extrahiert und gemeinsam mit den Metadaten an den folgenden Kontrollblock weitergeleitet.
- **Ingress**: Hier wird sichergestellt, dass nur Pakete mit einem validen Packet-Gen-Header weiterverarbeitet werden. Alle anderen Pakete werden verworfen.
- **Ingress Deparser**: Die Pakete werden wieder serialisiert und an den nächsten Block weitergeleitet.
- **Egress Parser**: Die Pakete werden erneut geparst und die Metadaten extrahiert. Dabei werden Quell- und Ziel-IP-Adressen aus den TCP- und UDP-Checksummen extrahiert und an den nächsten Kontrollblock weitergeleitet.
- **Egress**: Innerhalb dieses Kontrollblocks werden Quell- und Ziel-IP-Adressen anhand von Subnetzmasken randomisiert, welche vom Controller konfiguriert werden. Praktisch geschieht dies in zwei Schritten:
  1. Die Subnetzmasken werden negiert und bitweise mit jeweils einer Zufallszahl UND-verknüpft.
  2. Quell- und Ziel-IP-Adressen des Paket-Headers werden mit den beiden entstandenen Masken bitweise ODER-verknüpft.
- **Egress Deparser**: Die IP- und ggf. UDP- oder TCP-Checksummen werden aktualisiert. Die Pakete werden wieder serialisiert und weitergeleitet.

```
...
// Random variation of source and destination IP addresses
apply {
    if(egress_table.apply().hit) {
        // get random 32 bit number and make bitwise AND with negated network mask
        bit<32> s_tmp = src_rand.get() & ~src_mask;
        bit<32> d_tmp = dst_rand.get() & ~dst_mask;
        // apply random sub ip string to ip address
        hdr.ipv4.src_addr = hdr.ipv4.src_addr | s_tmp;
        hdr.ipv4.dst_addr = hdr.ipv4.dst_addr | d_tmp;
    }
}
...
```

Die Forwarding-Pipeline hat also zusammengefasst im Grunde zwei Hauptaufgaben: das Randomisieren der Adressen und die anschließende Aktualisierung der jeweiligen Checksummen. Die Checksummen sind wichtig, da sie die Integrität der generierten Pakete anzei-

gen. Die meisten Systeme werden beispielsweise TCP-Pakete mit ungültigen Checksummen einfach verwerfen, was den gewünschten Effekt des Traffic-Generators zunichte machen würde.

Neben der eigentlichen Verarbeitungslogik enthält der P4-Code auch die notwendigen Header-Definitionen für gängige Protokolle wie Ethernet, IPv4, UDP sowie für den Packet-Gen-Header. Diese Header-Definitionen sind notwendig, um die Pakete innerhalb der Forwarding-Pipeline korrekt zu parsen.

4.3.2 Control Plane

Die Control Plane des Switches ist für die Konfiguration und Steuerung der Data Plane zuständig und ist im Vergleich zur Data Plane in ihrer Funktionalität wesentlich umfangreicher. Neben einer Reihe von Funktionen zur Initialisierung und Steuerung des integrierten Packet-Generators sowie der Forwarding-Pipeline enthält der Controller eine Vielzahl von Management- und Monitoring-Funktionen, die für die Entwicklung und den Betrieb eines Switches notwendig sind. Der Controller ist in der Lage, die Konfiguration des Switches zu verwalten, P4-Programme zu laden und zu kompilieren sowie Metriken und Statistiken zu sammeln und auszuwerten. Dabei wird die Ansteuerung eines virtuellen Tofino-Modells und eines physischen Tofino-Switches weitestgehend vereinheitlicht, um die Entwicklung und das Testen zu vereinfachen. In den folgenden Abschnitten wird auf die wichtigsten Funktionen des Controllers eingegangen.

4.3.3 Switch-Management und Konfiguration über SSH

Der erste Schritt in der Entwicklung des Controllers war die Implementierung der Managementfunktionen. Diese entsprechen zum Großteil Befehlen, die über eine SSH-Verbindung auf dem Switch ausgeführt werden. Wie bereits erwähnt, kommt dafür die Python-Bibliothek Paramiko zum Einsatz. Warum die Entwicklung dieser Funktionen so wichtig ist, soll hier einmal verdeutlicht werden.

Um ein P4-Programm auf dem Switch auszuführen, müssen auf dem Switch folgende Schritte durchgeführt werden:

1. Die korrekten Verzeichnispfade der BF-SDE-Installation werden aus den Umgebungsvariablen des Zielsystems ausgelesen.
2. Der generierte Source-Code des P4-Programms sowie Konfigurationsdateien und Templates werden auf den Switch übertragen.
3. Der Source-Code wird mit dem P4-Compiler kompiliert und in ausführbaren Code umgewandelt. Die Binary sowie die API-Definitionen (`bf-rt.json`) werden in das korrekte Verzeichnis auf dem Switch kopiert.
4. Je nachdem, ob das Modell oder die reale Hardware angesteuert werden soll, werden folgende Schritte ausgeführt:
    - Virtuelles Modell: Die virtuellen Netzwerkinterfaces werden erstellt und konfiguriert, dementsprechend wird eine Portkonfiguration im JSON-Format erstellt. Das Tofino-Modell wird gestartet und das Programm sowie die Portkonfiguration übergeben. Anschließend wird der Switch-Daemon bf_switchd gestartet.

- Reale Hardware: Der Switch-Daemon bf_switchd wird direkt gestartet und das P4-Programm wird geladen. Anschließend wird die Portkonfiguration via bfshell vorgenommen.

5. Nachdem der Switch-Daemon gestartet wurde, steht das Barefoot Runtime Interface zur Verfügung und die Match-Action-Tables des Switches können konfiguriert werden.

Diese Schritte sind in der Praxis relativ umständlich, wenn sie jedes Mal händisch ausgeführt werden müssen. Es ist daher sinnvoll, diesen Ablauf so weit wie möglich zu automatisieren, um die weitere Entwicklung zu beschleunigen. Zudem wird die Benutzerfreundlichkeit des Programms dadurch erheblich gesteigert.

Für die Umsetzung wird ein Python-Modul entwickelt, das die Grundfunktionen für das Switch-Management bereitstellt. Das Modul bietet Funktionen für Datei- und Verzeichnis-Management sowie für das Handling der unterschiedlichen Prozesse auf dem Switch. Diese Funktionen können dann von einer neu definierten Klasse `TofinoSwitch` innerhalb des `switch_controller`-Moduls verwendet werden.

Diese Klasse übernimmt alle notwendigen Schritte für das Switch-Management und sorgt für die korrekte Durchführung der zuvor genannten Schritte. Es wurden zudem einige Optimierungen vorgenommen, um die Ausführungsgeschwindigkeit zu erhöhen und die Zuverlässigkeit der Funktionen zu verbessern. Beispielsweise werden die lokal generierten Dateien mit Dateien, die möglicherweise bereits auf dem Switch verfügbar sind, verglichen, um unnötige Übertragungen und Kompilierungsschritte zu vermeiden. Prozesse werden nur neu gestartet, wenn es aufgrund von bestimmten Änderungen notwendig ist.

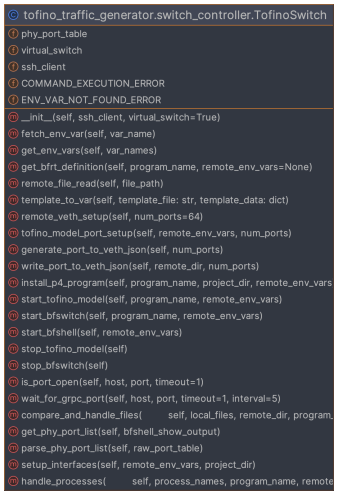

Abbildung 19: Gesamtbersicht: Virtuelle Entwicklungsumgebung

4.3.3.1 Konfiguration des Traffic-Generators

Der Traffic-Generator akzeptiert eine Reihe von Konfigurationsparametern, um die verschiedenen Eigenschaften des gewünschten Traffic-Musters zu definieren. Aufgrund des Umfangs dieser Konfiguration sind die meisten Parameter mit sinnvollen Standardwerten vorbelegt. Die folgenden Parameter können unter anderem vom Benutzer konfiguriert werden:

- **Physisch oder virtuell**: Der Traffic-Generator kann entweder auf einem physischen Tofino-Switch oder auf einer virtuellen Instanz des Tofino-Modells ausgeführt werden. Der Benutzer kann diese Option entsprechend auswählen.
- **Interner Packet-Generator**: Die Konfiguration des integrierten Packet-Generators des Tofino-Switches umfasst die Auswahl des internen Generator-Ports, die Einstellung des Packet-Buffers sowie die Dauer der Traffic-Generierung.
- **Output-Port**: Der Benutzer kann den Ausgangsport des Switches auswählen, an den der generierte Traffic gesendet werden soll.
- **Packet-Template**: Es kann zwischen verschiedenen vordefinierten Packet-Templates gewählt werden, die die Art des generierten Traffics definieren. Beispiele für Packet-Templates sind TCP-SYN-Flood, UDP-Flood oder ICMP-Flood. Es sind zudem detaillierte Konfigurationen der Packet-Header möglich. Außerdem kann die Länge des Pakets in Bytes definiert werden.
- **Traffic-Rate**: Der Benutzer kann eine konstante Datenrate des generierten Traffics in Bits pro Sekunde (bps) definieren.
- **IP-Ranges**: Der Benutzer kann die Quell- und Ziel-IP-Adressen in Form von CIDR-Notation definieren, um die Herkunft und das Ziel des generierten Traffics zu steuern. IP-Adressen innerhalb dieses Netzwerks werden zufällig generiert.

Aus der Gesamtkonfiguration des Traffic-Generators werden verschiedene Artefakte generiert, die für die weitere Verarbeitung notwendig sind. Dazu gehören der finale P4-Code und Port-Konfigurationen, die auf den Switch übertragen und dort kompiliert bzw. ausgeführt werden.

Die gesamte Konfigurationslogik wird in einer Klasse `TrafficConfigurator` gekapselt, die alle Funktionen für die Konfiguration des Traffic-Generators bereitstellt. Innerhalb des Programms kann damit ein Konfigurationsobjekt erstellt werden, das die gewünschten Parameter für die Traffic-Generierung als Attribute enthält. Hier ein Beispiel für die Konfiguration des Traffic-Generators mit einem simplen IP-Paket-Template:

```
# Create a new TrafficConfigurator object
traffic_configuration = TrafficConfigurator(virtual_switch=False)
traffic_configuration.configure_generator(port=68, generation_time_s=15)
traffic_configuration.add_physical_output_port(
    output_physical_port=1, port_speed="25G"
)
# Define the traffic rate in Mbps
traffic_configuration.add_throughput(throughput_mbps=1000, mode="port_shaping")
# Define the IP ranges for the source and destination, as well as the packet template
traffic_configuration.add_packet_data(
    source_cidr="192.168.178.1/24",
    destination_cidr="192.168.178.25/32",
    eth_src="e8:eb:d3:c1:56:e7",
    eth_dst="e8:eb:d3:c1:56:e5",
    pkt_len=500,
)
# Create the Packet-Template with scapy
traffic_configuration.craft_ipv4_packet()
# Generate the P4-Code and other configurations
traffic_configuration.generate()
# Further application logic ...
...
```

Die Flexibilität des Traffic-Generators basiert unter anderem auf dem dynamischen Rendering von Konfigurationsdateien, P4-Code und einem Build-Script. Für die Implementierung werden Jinja-Templates verwendet. Diese erlauben das Erstellen der fertigen Dateien zur Laufzeit des Programms, wobei die Templates mit den Werten des Konfigurationsobjekts gefüllt werden. Der Nutzer kann diese Templates beliebig anpassen und erweitern, ohne dass große Änderungen am Code notwendig sind. Folgende Dateien werden generiert:

- `src/build.sh` : Ein Bash-Script, welches die Kompilierung des P4-Programms auf dem Switch durchführt. Es nutzt dabei das bereits vorhandene CMake-Script des BF-SDE.
- `src/headers.p4` : Die Header-Definitionen des P4-Programms.
- `src/traffic_gen.p4` : Das eigentliche P4-Programm, welches die Data Plane des Switches konfiguriert.
- `src/port_config.txt` : Port-Konfigurationen für reale Hardware. Entspricht einer Liste von Befehlen, welche der bfshell übergeben werden.
- `veth-setup.sh` : Ein Bash-Script, welches die virtuellen Netzwerkinterfaces für das Tofino-Modell erstellt.

Diese Dateien können mit den zuvor beschriebenen Managementfunktionen auf den Switch übertragen und dort verwendet werden.

Eine weitere wichtige Aufgabe der `TrafficConfigurator`-Klasse ist die Erstellung eines Template-Pakets auf Basis benutzerdefinierter Parameter. Der Benutzer kann die Header-Felder des Template-Pakets nach Belieben anpassen und so die anschließend generierten Datenpakete definieren. Für die Erstellung des Template-Pakets existieren einige Helfer-Funktionen, welche die Definition herkömmlicher Paket-Typen (IP, TCP, UDP, ICMP usw.) vereinfachen. Diese Funktionen wurden mithilfe der sehr mächtigen Scapy-Bibliothek implementiert, die die Erzeugung und Manipulation von Netzwerkpaketen ermöglicht. Durch Scapy sind dem Benutzer bei der Erstellung des Template-Pakets im Prinzip keine Grenzen gesetzt und die Erweiterung des Traffic-Generators um neue Packet-Templates wird stark vereinfacht.

Das Template-Paket wird im Packet-Buffer des Packet-Generators gespeichert und dient als Vorlage für die generierten Datenpakete. Das Template-Paket wird mit Scapy erstellt und enthält die Header-Informationen, die für die Traffic-Generierung notwendig sind. Der Benutzer kann die Header-Felder des Template-Pakets nach Belieben anpassen und so die generierten Datenpakete steuern.

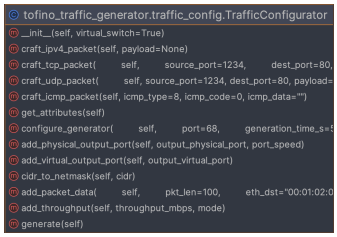

Abbildung 20: Diagramm: `TrafficConfigurator`-Klasse

4.3.3.2 Steuerung des Traffic-Generators

Die Remote-Funktionalität des Traffic-Generators ermöglicht dem Nutzer die einfache Konfiguration und Steuerung der Management- und Data Plane des Switches von einem entfernten Managementsystem. Umständliche Konfigurationsschritte auf dem Switch werden so

vermieden, was die Entwicklung und Ausführung von Tests stark vereinfacht. Zudem kann der Nutzer bestehende Entwicklungs- und Automatisierungstools verwenden, die möglicherweise bereits auf dem Managementsystem vorhanden sind.

Die Remote-Steuerung des Switches erfolgt grundsätzlich in zwei Phasen:

1. Die Initialisierung des Switches erfolgt über eine SSH-Verbindung. In dieser Phase werden notwendige Dateien übertragen, die Kompilierung angestoßen und einzelne Services gestartet.
2. Nachdem die Switch-Services erfolgreich gestartet wurden, steht auf dem Switch ein Endpoint des Barefoot Runtime Interface zur Verfügung, über den die weitere Steuerung erfolgt.

Im Laufe des Projekts wurde für die Umsetzung dieser Features ein relativ umfangreiches Framework von Kontroll- und Steuerungsfunktionen entwickelt, das es dem Benutzer oder zukünftigen Entwicklern erlaubt, den Traffic-Generator beliebig zu erweitern und anzupassen. Dabei war die Verwendung der Python-Bibliothek `bfrt-helper` von APS Networks von großer Hilfe, denn sie vereinfacht die Formulierung von gRPC-Nachrichten und die Verwaltung der gRPC-Verbindung immens. Die Bibliothek bietet im Kern zwei wichtige Klassen: Die `BfRtInfo`-Klasse, die Informationen über die verfügbaren Match-Action-Tables des P4-Programms bereitstellt, und die `BfRtHelper`-Klasse, die die Konfiguration dieser Tables ermöglicht. Hier wird beispielhaft die Konfiguration der Egress-Pipeline gezeigt:

```
request = self.bfrt_helper.create_table_write(
    program_name,
    egress_table.name,
    {"eg_intr_md.egress_port": Exact(PortId(output_port))},
    action_name="SwitchEgress.replace_ip_address",
    action_params={
        "s_mask": IntField(ip2int(source_mask)),
        "d_mask": IntField(ip2int(destination_mask)),
        "src_mac": MACField(mac2int(src_mac)),
        "dst_mac": MACField(mac2int(dst_mac)),
    },
    update_type=bfruntime_pb2.Update.Type.INSERT,
)
response = self.grpc_client.Write(request)
```

Selbst wenn für komplexere Anwendungsfälle keine direkten Funktionen bereitgestellt werden, kann die Bibliothek als Grundlage für die Entwicklung eigener Funktionen dienen, die die gewünschten gRPC-Nachrichten erzeugen und senden.

Das `switch_controller`-Modul des Traffic-Generators verfügt neben `TofinoSwitch` über zwei Klassen, die die gesamte Control-Plane-Funktionalität bereitstellen. Die `GRPCManager`-Klasse verwaltet die gRPC-Verbindung und beinhaltet Methoden zum Aufbau und Abbau der Verbindung sowie zur Initialisierung der `bfrt-helper`-Objekte.

Abbildung 21: Diagramm: `GRPCManager`-Klasse

Die zweite wichtige Klasse, `TrafficGenerator`, enthält die Methoden für die eigentliche Konfiguration der Data Plane und damit auch für die Steuerung des internen Packet-Generators.

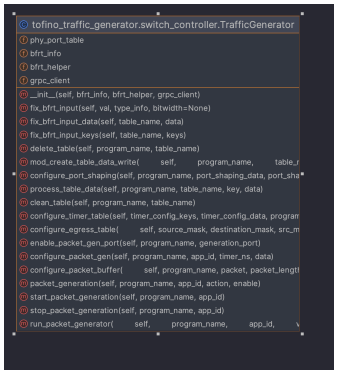

Abbildung 22: Diagramm: `GRPCManager`-Klasse

Auf die Klasse `TrafficGenerator` und ihre Aufgaben soll an dieser Stelle näher eingegangen werden, sie enthält unter anderem folgende besonders relevante Methoden:

- `configure_port_shaping()`: Konfiguriert die maximale Datenrate des Packet-Generators.
- `configure_timer_table()`: Notwendige Konfigurationen für die Timer-Modi des internen Packet-Generators.
- `configure_egress_table()`: Konfiguriert die Egress-Table des Switches mit den zuvor festgelegten Subnetzmasken.
- `configure_packet_gen()`: Konfiguriert den internen Packet-Generator mit den zuvor festgelegten Parametern.
- `configure_packet_buffer()`: Schreibt das Template-Paket in den Packet-Buffer des internen Packet-Generators.
- `packet_generation()`: Umfangreiche Methode zur Ausführung der Traffic-Generierung, die bei Verwendung realer Hardware auch Metriken erfasst.

4.3.4 Metriken des Traffic-Generators

Der entwickelte Prototyp bietet eine relativ rudimentäre Metrikerfassung, die für Auswertungen von Testergebnissen und die Analyse des generierten Traffics verwendet werden kann. Diese Metriken werden in Registern der Tofino Native Architecture für jeden Port au-

tomatisch erfasst und können vom Controller über das BRI (Barefoot Runtime Interface) abgefragt werden. Die Abfrage erfolgt in regelmäßigen Abständen (Polling) parallel zur Traffic-Generierung.

Die folgenden Metriken stehen dem Benutzer nach Abschluss eines Tests in Form einer Zeitreihe zur Verfügung:

- Gesamtanzahl der generierten Pakete
- Paketrate in Paketen pro Sekunde (Mpps)
- Durchsatz in Bits pro Sekunde (Mbps)

Um das Ganze zu veranschaulichen, hier ein Beispiel für die Ausgabe des Programms:

```
$ python main.py
[+] Generated src/traffic_gen.p4
[+] Generated src/headers.p4
[+] Generated src/port_config.txt
[*] Comparing generated source files with existing source files ...
  > traffic_gen.p4: Same
  > util.p4: Same
  > headers.p4: Same
[!] P4 source has not changed, skipped compilation step.
[+] Setting up physical ports
[OK] Port 50052 on 172.16.1.11 is now available!
[OK] gRPC Connection established
[+] Copied and loaded extended Barefoot Runtime definition for Tofino target
[+] Beginning traffic generator setup
  > Initialize traffic data rate to 25000 Mbps
  > Configure timer table
  > Configure source and destination mask
  > Configure traffic generator
  > Write initial packet into traffic generator buffer
[*] Press Enter to continue ...
[+] Running traffic generator for 30 seconds ...
  > Collecting port metrics from device port 56 ...
[OK] Run finished!
[+] Generate report:
  > Initial packet definition:
   ...
  > Traffic generator metrics for this run:

  n    time_s    total_frames_tx     Mpps     Mbps
---  --------  -----------------  -------  -------
  0   1.00094            5516129  5.51095  22043.8
  1   2.00403           11416165  5.69659  22786.4
  2   3.00705           17628625  5.86243  23449.7
  3   4.01013           23413055  5.83847  23353.9
  4   5.01322           29301617  5.84487  23379.5
  5   6.01619           35519581  5.904    23616
  6   7.01917           41304239  5.88449  23538
  7   8.02205           47193062  5.88292  23531.7
  8   9.02507           53410785  5.91804  23672.2
  9  10.028             59195232  5.90302  23612.1
 10  11.0309            65089893  5.90067  23602.7
  ...
```

Zusätzliche Daten können durch Tools auf dem Device-under-Test (DUT) erhoben werden, um die Auswirkungen des generierten Traffics auf das Zielsystem zu analysieren. Da in dieser Kategorie bereits viele Tools und Frameworks existieren, grenzt sich der Traffic-Generator bewusst von der Implementierung solcher Funktionen ab. Im weiteren Verlauf dieser

Dokumentation wird auf Beispiele für solche Tools eingegangen und deren Verwendung im Zusammenhang mit dem Traffic-Generator erläutert.

Andere, bereits vorgestellte Projekte wie P4TG gehen hier einen anderen Weg und bieten eine umfangreiche Metrikerfassung, die weit über die hier vorgestellten Funktionen hinausgeht. Die Entscheidung für eine einfache Metrikerfassung im Rahmen dieses Projekts wurde bewusst getroffen, um die Entwicklung dieses Prototyps zu vereinfachen und den Fokus auf die Kernfunktionalität zu legen. Zukünftige Weiterentwicklungen des Traffic-Generators könnten jedoch die Funktionen anderer Projekte integrieren und so die Metrikerfassung erweitern. Beispiele für solche Metriken sind Latenz, Jitter, Paketverlust oder Inter-Packet-Time.

5 Tests und Belastungssimulationen

Um die Funktionalität und Korrektheit des entwickelten Systems zu verifizieren, ist es notwendig, bestimmte Tests in einer geeigneten Testumgebung durchzuführen. Die in diesem Projekt verwendeten Technologien bieten umfangreiches Tooling für die Umsetzung entsprechender Testszenarien. Des Weiteren kann für die Simulation und Analyse von Netzwerken und Netzwerkverkehr auf eine Vielzahl von Open-Source-Tools zurückgegriffen werden. Das Gleiche gilt für die Sammlung und Auswertung von Daten bezüglich der Performance eines Zielsystems für generierten Traffic.

Im Kontext dieses Projekts lässt sich zwischen verschiedenen Arten von Tests unterscheiden, die wiederum unterschiedliche Anforderungen an eine bestimmte Testumgebung stellen. Die folgenden Tests sind für die Evaluation des Traffic-Generators besonders relevant:

- **Funktionale Tests**: Überprüfung der korrekten Implementierung der verschiedenen Funktionen, Konfigurationen und Steuerungsmöglichkeiten des Traffic-Generators.
- **Leistungstests**: Überprüfung der Leistungsfähigkeit des Traffic-Generators, insbesondere hinsichtlich der generierten Datenrate und des Durchsatzes.
- **Stabilitätstests**: Überprüfung der Stabilität des Traffic-Generators, insbesondere hinsichtlich der Dauer der Traffic-Generierung und der Zuverlässigkeit der Metrikerfassung.
- **Lasttests**: Überprüfung der Auswirkungen des generierten Traffics auf das Zielsystem, insbesondere hinsichtlich der Verarbeitungsgeschwindigkeit und der Ressourcennutzung.

Im weiteren Verlauf dieses Kapitels sollen verschiedene Testumgebungen vorgestellt werden, die für die Durchführung dieser Tests geeignet sind. Zusätzlich werden einige beispielhafte Testszenarien und deren Ergebnisse präsentiert. Im Zuge dessen wird auch auf die Verwendung spezieller Tools eingegangen, die für die Umsetzung bestimmter Tests erforderlich bzw. hilfreich sind. Zunächst sollen jedoch einige Anforderungen für eine adäquate Testumgebung definiert werden.

Die Tests der allgemeinen Funktionen des Traffic-Generators stellen insofern eine Herausforderung dar, als dass sie bereits einen Switch zur Ausführung benötigen, insbesondere dann, wenn es um Funktionen der Data Plane geht. Ähnliches gilt allerdings auch für Tests der Control Plane, da hierfür eine Data Plane API auf dem Switch zur Verfügung stehen muss. Praktischerweise bietet die im Kapitel zuvor vorgestellte virtuelle Entwicklungsumgebung mit dem Tofino-Modell eine geeignete Testumgebung für die Durchführung dieser Tests.

5.1 Tools und Anwendungen

Für die Durchführung der Tests kommen eine Reihe von Tools zum Einsatz. Diese sind nützlich, um Netzwerktraffic sowie allgemeine Performancecharakteristiken zu analysieren und für weitere Auswertungen zu speichern. Folgende Liste enthält einige der wichtigsten Anwendungen für diesen Zweck:

- **Wireshark**: Ein mächtiges Netzwerkanalysetool, das die Erfassung und Analyse von Netzwerkverkehr ermöglicht. Wireshark ist besonders nutzerfreundlich, da es eine grafische Benutzeroberfläche bietet. [Wir24]
- **tcpdump**: Das wahrscheinlich wichtigste Tool dieses Projekts. Ähnlich wie Wireshark ermöglicht tcpdump die Erfassung von Netzwerkverkehr, ist jedoch sehr simpel in der Anwendung und daher für Debugging-Zwecke besonders gut geeignet. Anders als Wireshark handelt es sich um ein reines Kommandozeilentool, was den Einsatz auf Servern und in Skripten erleichtert. [Tcp24]
- **iftop**: Kommandozeilentool zur Echtzeitüberwachung des Netzwerkverkehrs. Es zeigt die Datenrate und die Anzahl der Pakete pro Sekunde für jede Netzwerkverbindung an und bietet einen schnellen Überblick über die aktuelle Netzwerkauslastung. [ift24]
- **netdata**: Extrem leistungsstarkes Tool zur Echtzeitüberwachung von Systemen und Anwendungen, welches mit einer Vielzahl von Plugins erweiterbar ist. Netdata bietet ein grafisches Dashboard in Form einer Webanwendung, das sich nach Belieben modifizieren lässt. [Net24]

5.2 Virtuelle Testumgebung

Da die meisten Informationen zu dieser Umgebung bereits im vorherigen Kapitel vorgestellt wurden, soll an dieser Stelle nur noch einmal auf die wichtigsten Aspekte eingegangen werden. Die virtuelle Testumgebung besteht aus einer virtuellen Maschine, die auf einem x86-System ausgeführt wird und das vollständige Barefoot Software Development Environment sowie das Tofino-Modell enthält. Dieses System kann für Tests der grundlegenden Management- und Control-Plane-Funktionen des Traffic-Generators verwendet werden, da es neben einem SSH-Server auch ein Barefoot Runtime Interface zur Verfügung stellt.

Ein Vorteil des Tofino-Modells ist, dass es einen umfangreichen Debugging-Modus bietet, der es ermöglicht, die Data Plane des Switches aktiv zu überwachen. So werden die einzelnen Schritte der in P4 definierten Forwarding-Pipeline für den Entwickler sichtbar gemacht und können auf Korrektheit überprüft werden. Ein weiteres nützliches Feature der virtuellen Testumgebung ist die Möglichkeit, die generierten Datenpakete mit einem Netzwerkanalysetool zu überwachen.

Das ist möglich, da die Switch-Ports des Tofino-Modells einfach mit virtuellen Netzwerkinterfaces auf dem Hostsystem verbunden werden können. Anschließend kann der Traffic auf diesen Interfaces mit entsprechenden Tools wie Wireshark oder tcpdump überwacht und aufgezeichnet werden.

Folgendes Beispiel soll diesen Prozess verdeutlichen. Zunächst wird im Programm die entsprechende Konfiguration vorgenommen:

```
traffic_configuration = TrafficConfigurator(virtual_switch=True)
traffic_configuration.configure_generator(port=68, generation_time_s=15)
traffic_configuration.add_virtual_output_port(1)
traffic_configuration.add_packet_data(
    source_cidr="10.1.1.74/24",
    destination_cidr="10.1.1.5/32",
    eth_src="e8:eb:d3:c1:56:e7",
    eth_dst="e8:eb:d3:c1:56:e5",
    pkt_len=500,
)
traffic_configuration.craft_tcp_packet(with_checksum=True)
traffic_configuration.add_throughput(throughput_mbps=1000, mode="port_shaping")
traffic_configuration.generate()
...
```

Um Einblicke über das Verhalten der Data Plane zu erlangen, können im Voraus sowohl `tofino-model` als auch `bf_switchd` von der Kommandozeile gestartet werden. Um den generierten Traffic zu überwachen, können in weiteren Terminal-Fenstern die Tools `tcpdump` und `iftop` gestartet werden.

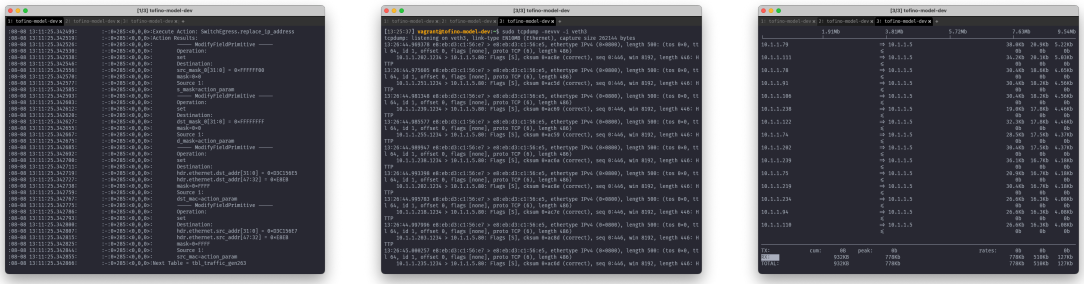

Abbildung 23: Testing-Tools (von links nach rechts): `tofino-model`, `tcpdump`, `iftop`

Zusammengefasst bietet demnach bereits eine vollständig virtuelle Testumgebung alle notwendigen Funktionen und Features, um die grundlegenden Funktionen des Traffic-Generators zu testen und zu evaluieren. Es gibt jedoch auch einige Einschränkungen, welche sie für weitere Testszenarien weniger geeignet machen. So ist die maximale Datenrate des internen Packet-Generators des Tofino-Modells stark begrenzt (siehe Abbildung) und kann nicht mit der Leistungsfähigkeit eines physischen Tofino-Switches mithalten. Zudem unterstützt das Modell kein *Port-Shaping*, also die Möglichkeit, die maximale Datenrate des Packet-Generators auf Port-Ebene zu konfigurieren. Es wird daher deutlich, dass für die Durchführung von Leistungstests, Stabilitätstests und Lasttests zusätzlich eine physische Testumgebung notwendig ist.

5.2.1 Physische Testumgebung im Labor

Im Netzwerklabor der HTW Berlin stehen verschiedene Systeme für den Aufbau einer Testumgebung zur Verfügung, wobei besonders der Netzwerkswitch *EdgeCore Wedge100BF-32X* (DCS800) hervorzuheben ist. Dieser Switch verfügt über den notwendigen Intel Tofino-Chip und bietet daher die Möglichkeit, P4-Programme direkt auf dem Switch auszuführen. Die wichtigsten Spezifikationen des Switches sind:

- 32xQSFP28-Ports mit Support für 1x25/40/50/100 GbE
- NPU: Intel Tofino BFN-T10-032D (Dual Pipeline)

- CPU: Intel x86 Broadwell-DE, Pentium D-1517
- RAM: 8GB SO-DIMM DDR4
- Performance: 6.4 Tb/s Switching Capacity
- OS: Ubuntu 20.04 LTS

Neben dem Switch steht ebenfalls ein Server mit zwei 25GbE-Netzwerkkarten zur Verfügung, der als Zielsystem für Lasttests und Performance-Messungen verwendet werden kann. Wie bei dem Switch ist auch hier ein herkömmliches Linux-Betriebssystem (Ubuntu 20.04 LTS) installiert, welches die Ausführung von Tools und Anwendungen zur Überwachung und Analyse des generierten Traffics ermöglicht. Dieses System hat folgende Eigenschaften:

- OS: Ubuntu 20.04 LTS
- Kernel: 5.15.0-117-generic
- CPU: Intel i7-8700 (12) @ 4.600GHz
- RAM: 16GB
- 2x25GbE-Netzwerkkarten

Für die Verbindung von Switch und Server wird ein Breakout-Kabel verwendet, das einen der 100GbE-Ports des Switches auf vier 25GbE-Ports aufteilt. In dem Testsetup wird einer der vier Ports mit dem Server verbunden, was einen 25GbE-Link ergibt. Außerdem werden zu Kontrollzwecken zwei der 25GbE-Ports des Switches direkt miteinander verbunden.

Für den Test von höheren Datenraten sind außerdem zwei 100GbE-Ports des Switches direkt miteinander verbunden, um den Traffic-Generator mit einer maximalen Datenrate von 100 Gbps zu testen. Es ergibt sich dadurch folgende Netzwerktopologie:

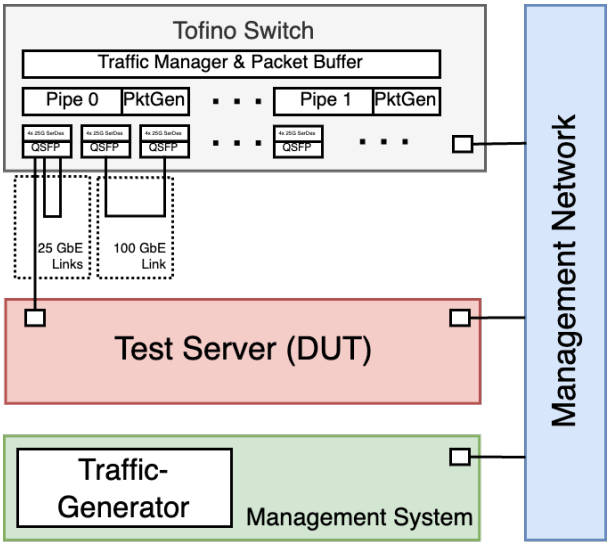

Abbildung 24: Diagramm: `GRPCManager` -Klasse

Diese Testumgebung ermöglicht aufgrund der höheren möglichen Datenraten und der verfügbaren Hardwarekomponenten erweiterte Testszenarien, die in der virtuellen Testumgebung nicht möglich waren. Innerhalb der Umgebung lassen sich dementsprechend einige Belastungssimulationen durchführen, um die Leistungsfähigkeit des Traffic-Generators zu überprüfen. Um zum Beispiel die maximale Datenrate des internen Packet-Generators zu testen, kann folgendermaßen vorgegangen werden.

Zunächst wird auf dem Switch ein Input- bzw. RX-Port konfiguriert, der die generierten Datenpakete empfängt. Die Konfiguration des Output-Ports wird von der Traffic-Generator-Software automatisch vorgenommen. Hier die Portkonfiguration:

```
bf-sde.pm> show
————+———+———+———+———————+———+——+——+———+———+———+———————+————————————————+————————+-
PORT |MAC |D_P|P/PT|SPEED  |FEC |AN|KR|RDY|ADM|OPR|LPBK   |FRAMES RX  |FRAMES TX  |E
————+———+———+———+———————+———+——+——+———+———+———+———————+————————————————+————————+-
7/0  |17/0|180|2/52|100G   |NONE|Ds|Au|YES|ENB|UP |  NONE  |          0|          0|
8/0  |16/0|188|2/60|100G   |NONE|Ds|Au|YES|ENB|UP |  NONE  |          0|          0|
```

Anschließend kann die Konfiguration in Traffic-Generator vorgenommen und der Test gestartet werden:

```
traffic_configuration = TrafficConfigurator(virtual_switch=False)
traffic_configuration.configure_generator(port=68, generation_time_s=15)
traffic_configuration.add_physical_output_port(
    output_physical_port=7, port_speed="100G"
)
traffic_configuration.add_packet_data(
    source_cidr="10.1.0.0/16",
    destination_cidr="10.1.1.5/32",
    eth_dst="e8:eb:d3:c1:56:e5",
    pkt_len=1024,
)
traffic_configuration.craft_tcp_packet()
traffic_configuration.add_throughput(throughput_mbps=100000, "port_shaping")
traffic_configuration.generate()
...
```

In der der `bfshell` können mit dem Befehl `rate-show` die aktuellen Metriken der Ports abgefragt werden. Hier ist die Ausgabe dargestellt. Es wird deutlich, dass die Zieldatenrate von 100 Gb/s nur knapp nicht erreicht wurde. Dennoch ist die generierte Datenrate von ca. 99785 Mb/s als Erfolg zu werten.

```
bf-sde.pm> rate-show
————+———+———+———+———————+———`———————+———————+———————————+———————————+———+———
PORT |MAC |D_P|P/PT|SPEED  |RDY|RX Mpps|TX Mpps|RX Mbps  |TX Mbps  |RX %|TX %
————+———+———+———+———————+———+———————+———————+———————————+———————————+———+———
7/0  |17/0|180|2/52|100G   |UP |   0.00|  11.90|     0.00| 99784.43|  0%| 99%
8/0  |16/0|188|2/60|100G   |UP |  11.90|   0.00| 99785.84|     0.00| 99%|  0%
————+———+———+———+———————+———+———————+———————+———————————+———————————+———+———
Pipe2| RX RATE   11.90Mpps   99785Mbps| TX RATE   11.90Mpps   99784Mbps
```

Die Werte aus dem Output des Traffic-Generators können ebenfalls zur Auswerten betrachtet werden. Hier sind die gemessenen Werte des Testlaufs dargestellt:

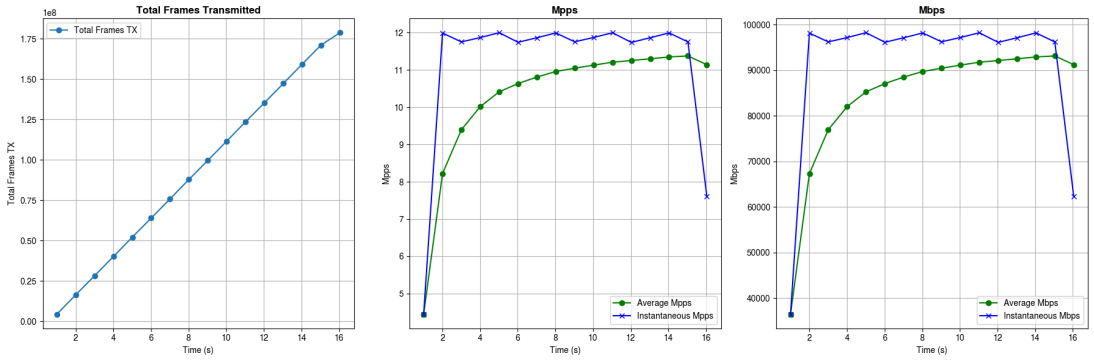

Abbildung 25: Traffic-Generator - Performance bei 100 Gb/s, 1024 Byte Frames, 15 Sekunden Dauer

Es wird deutlich, dass die Datenraten durchaus gewissen Schwankungen unterliegen. Zudem braucht es eine gewisse Zeit, bis die Datenrate den gewünschten Wert erreicht. Generell ist eine Testdauer von unter zwei Sekunden problematisch, da die Datenrate in dieser Zeit noch nicht stabil ist. Bei der Entwicklung war außerdem auffällig, dass die Metriken in den Port-Registern erst nach ca. drei Sekunden überhaupt Werte enthalten. Dies ist ein wichtiger Aspekt, der bei der Entwicklung von Testszenarien berücksichtigt werden sollte.

5.3 Szenario 1: TCP-SYN-Flood

Nachdem die allgemeine Funktionsfähigkeit des Traffic-Generators geprüft wurde, können nun komplexere Szenarien getestet werden. Ein interessantes Szenario ist der TCP-SYN-Flood-Angriff, welcher bereits zuvor beschrieben wurde. Als Ziel für den Angriff dient der Server, der im Netzwerklabor zur Verfügung steht. Um die Auswirkungen des Angriffs zu untersuchen, wird netdata auf dem Server installiert und gestartet. Über die anschließend verfügbare Weboberfläche können nun sehr umfassende Metriken in Echtzeit überwacht werden.

Als Kontrolle werden zuerst simple IPv4-Pakete ohne TCP-Header generiert und an den Server gesendet, wobei darauf geachtet wird, dass die Ziel-IP- sowie MAC-Adresse dem Netzwerkinterface des Servers entsprechen. Als Quell-Adressbereich wird 10.1.0.0/16 ausgewählt. Die Datenrate des Traffic-Generators wird auf die maximalen 25 Gb/s des Links eingestellt und die Paketlänge auf 1024 Bytes festgelegt. Die Testdauer beträgt 30 Sekunden. Das hat auf den Server folgende Auswirkungen:

Abbildung 26: TCP-SYN-Flood - Kontrolltest

Es ist sichtbar, dass der TCP/IP-Stack des Servers ca. 2,4 Millionen IPv4-Pakete pro Sekunde entgegennimmt, diese aber aufgrund des unbekannten Protokolls verwirft. Das hat durchaus Auswirkungen auf die CPU-Last, wie in den beiden oberen Grafiken zu sehen ist, auch wenn diese mit ca. 8 % noch recht gering ist. Aus der Grafik wird auch deutlich, was der genaue Grund für die erhöhte Auslastung ist: Die Anzahl der Software-Interrupts von Typ `NET_RX` ist stark erhöht, was auf die Verarbeitung der Pakete zurückzuführen ist.

Nun wird der eigentliche TCP-SYN-Flood-Angriff durchgeführt. Die oben genannten Parameter werden dabei nicht verändert, lediglich die Art der generierten Pakete wird angepasst, indem ein TCP-Header mit `SYN`-Flag hinzugefügt wird. In der folgenden Grafik sind die Auswirkungen des Angriffs auf den Server dargestellt:

5.3 Szenario 1: TCP-SYN-Flood

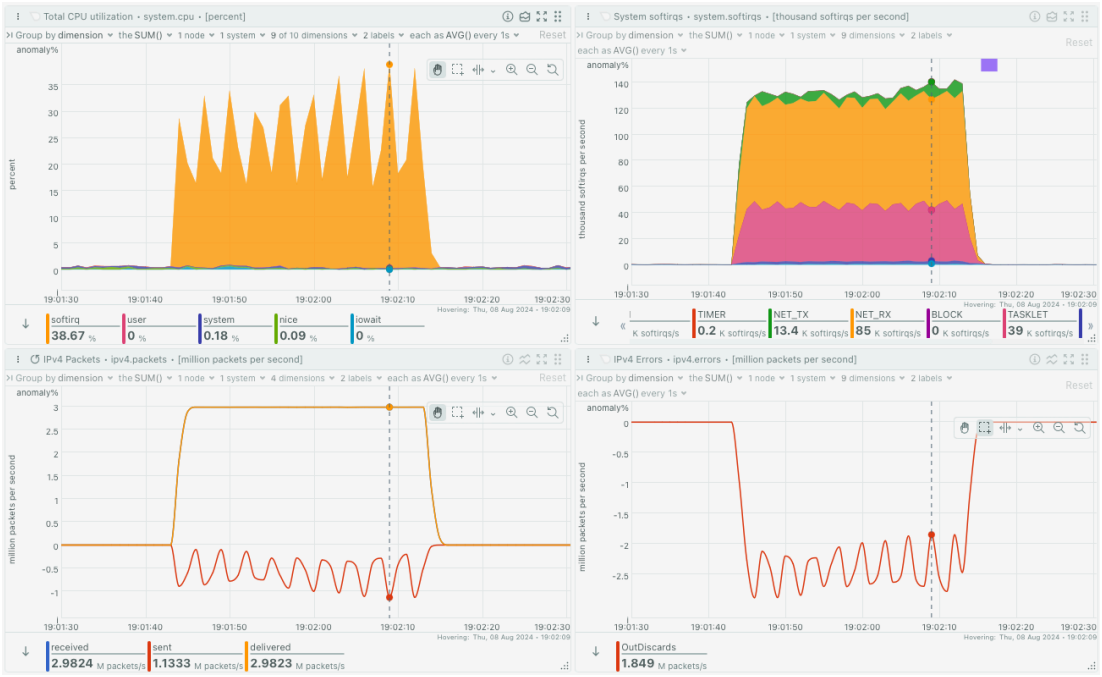

Abbildung 27: TCP-SYN-Flood - Test 1.0

Sofort wird deutlich, dass die CPU-Auslastung mit Spitzen von ca. 38 % wesentlich höher ist als beim Kontrolltest. Die Lastspitzen korrelieren interessanterweise mit dem nun ausgehenden Traffic des Servers, der nun gezwungen ist, die Pakete tatsächlich zu verarbeiten.

Eine weitere Analyse des Traffics zeigt eine große Menge von TCP-Reset-Paketen (TCP-RST), die vom Server an den Traffic-Generator gesendet werden.

```
10.1.1.5.80 > 10.1.1.150.1234: Flags [R.], cksum 0×9047 (correct), seq 0, ack 1, win 0,
length 0
10.1.1.5.80 > 10.1.1.106.1234: Flags [R.], cksum 0×9073 (correct), seq 0, ack 1, win 0,
length 0
...
```

Entspricht dies nun bereits der angekündigten TCP-SYN-Flood-Attacke? Nein, denn wie in der folgenden Grafik zu erkennen ist, bleibt die SYN-Queue des Servers völlig leer und der Grund dafür ist simpel: Auf dem Testserver ist keine Anwendung gestartet, die auf unserem Zielport 80 lauscht. Das erklärt auch, warum der Server eingehende Verbindungen sofort mit TCP-RST-Paketen terminiert.

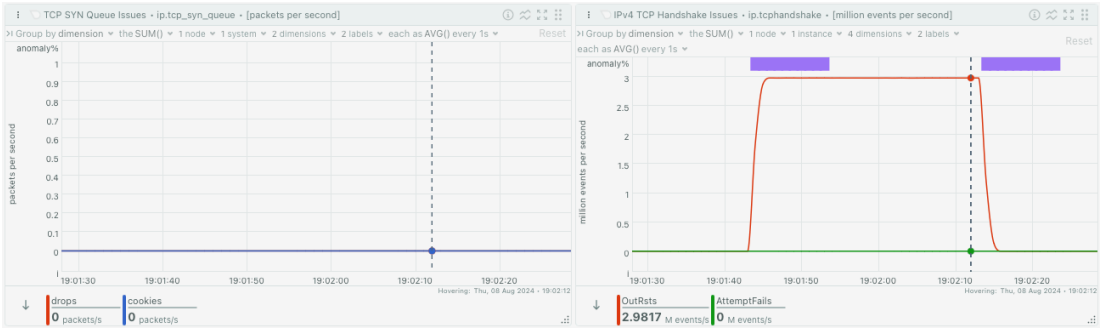

Abbildung 28: TCP-SYN-Flood - Test 1.1

Schließlich wird die Testumgebung um einen Apache-Webserver erweitert, der auf dem Zielsystem für eingehende Verbindungen auf Port 80 lauscht. Der Traffic-Generator wird erneut konfiguriert und der Angriff gestartet. Die Auswirkungen auf den Server sind in der folgenden Grafik dargestellt:

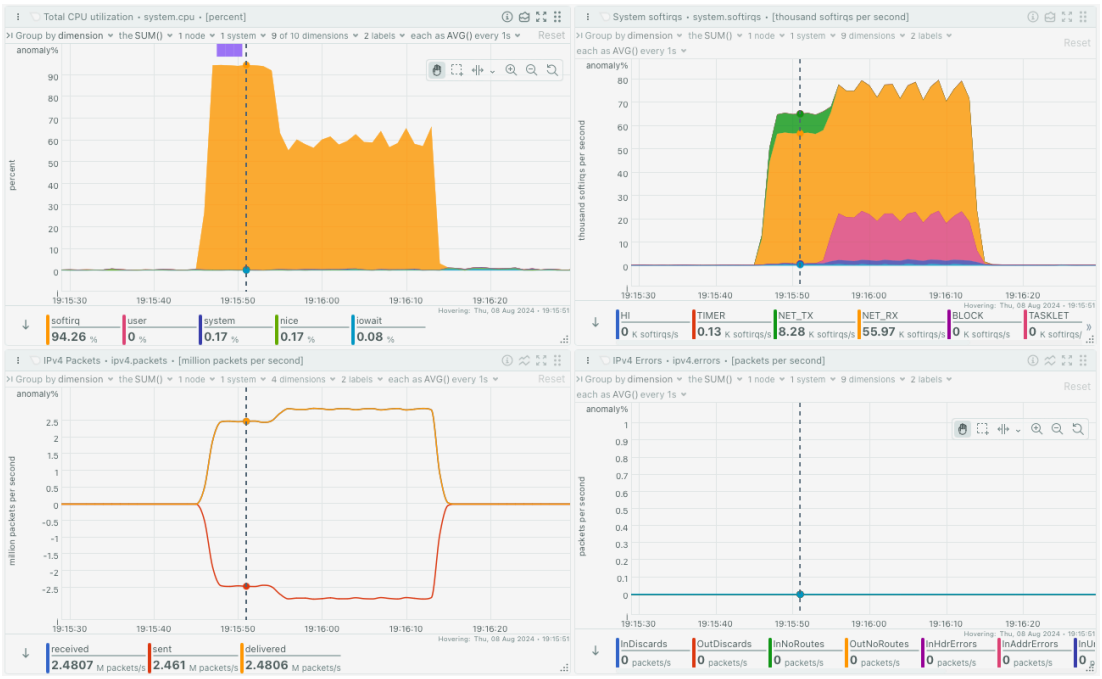

Abbildung 29: TCP-SYN-Flood - Test 2.0

Nun tritt tatsächlich der Fall ein, dass der eingehende Traffic den Server mit Lastspitzen von bis zu 100 % CPU-Auslastung überlastet. Weitere Tests zeigten, dass insbesondere längere Testdauern zu einer durchgängigen maximalen CPU-Auslastung führen. Neben der allgemeinen Auslastung der Bandbreite des Netzwerkinterfaces wird nun auch ein Problem in der TCP-SYN-Queue sichtbar, die nach Start des Webservers schnell anwächst. Zur Mitigation greift das System auf sogenannte SYN-Cookies zurück, wie aus der Grafik deutlich wird.

Abbildung 30: TCP-SYN-Flood - Test 2.1

SYN-Cookies sind eine Sicherheitsmaßnahme zum Schutz vor SYN-Flooding-Angriffen, die in RFC 4987 definiert ist. Anstatt die SYN-Queue zu füllen, werden dabei Verbindungsinformationen innerhalb der initialen Sequenznummer des SYN-ACK-Pakets codiert und an den Client gesendet. Wenn dieser legitim ist und mit einem ACK antwortet, kann die Verbindung aufgebaut werden. Zusammengefasst erlauben SYN-Cookies die zustandslose Erzeugung von SYN-ACKs, um eingehende SYNs zu verarbeiten.

Dieser Mechanismus verhindert zwar das Überlaufen der SYN-Queue, führt aber auch zu einer erhöhten CPU-Last, da die Codierung und Decodierung der Sequenznummern zusätzliche Rechenleistung erfordert. Außerdem sind SYN-Cookies auf die 32-Bit Länge des TCP-Sequenznummerfeldes beschränkt, was wiederum die Verwendung optionaler TCP-Parameter wie Explicit Congestion Notification (ECN), Selective ACK oder Windows-Scaling verhindert. Zwar ist Linux in der Lage, diese Werte als eine Art Workaround im 32-Bit TCP-Timestamp-Feld zu speichern, jedoch ist dieses Verhalten nicht gerade standardkonform und führt zu einer höheren Bandbreitenauslastung von 12 Byte pro Paket. Wenn die Größe eines normalen TCP-Headers von 20 Byte angenommen wird, entspricht das einem Overhead von 66 %.

Aus der obigen Grafik ist zudem zu entnehmen, dass der Übergang zu und von SYN-Cookies nicht ganz fließend funktioniert. So kommt es nach dem Abbau der Cookies kurzzeitig zu Drops in der SYN-Queue, wie im Diagramm unten rechts zu erkennen ist.

5.4 Szenario 2: ICMP-Ping-Flood

Als zweites Szenario wird ein ICMP-Flood-Angriff durchgeführt. Hierbei wird der Traffic-Generator so konfiguriert, dass er ICMP-Pakete an den Server sendet. Die Paketlänge wird auf 100 Bytes festgelegt und die Datenrate auf 25 Gbps eingestellt. Da das Internet Control Message Protocol (ICMP) dazu entwickelt wurde, um Probleme im Netzwerk zu diagnostizie-

ren, indem es Nachrichten über den Status von Netzwerkeigenschaften sendet, ist es auf den meisten Systemen standardmäßig aktiviert. ICMP-Pakete werden daher standardmäßig von den meisten Systemen verarbeitet, was sie zu einem beliebten Angriffsvektor macht. Daher ist für diesen Test, anders als beim TCP-SYN-Flood-Angriff, keine spezielle Anwendung auf dem Zielsystem notwendig.

Der ICMP-Typ wird als *Ping Request* festgelegt, indem der ICMP-Typ 8 und ICMP-Code 0 gewählt werden. Die Testdauer beträgt erneut 30 Sekunden.

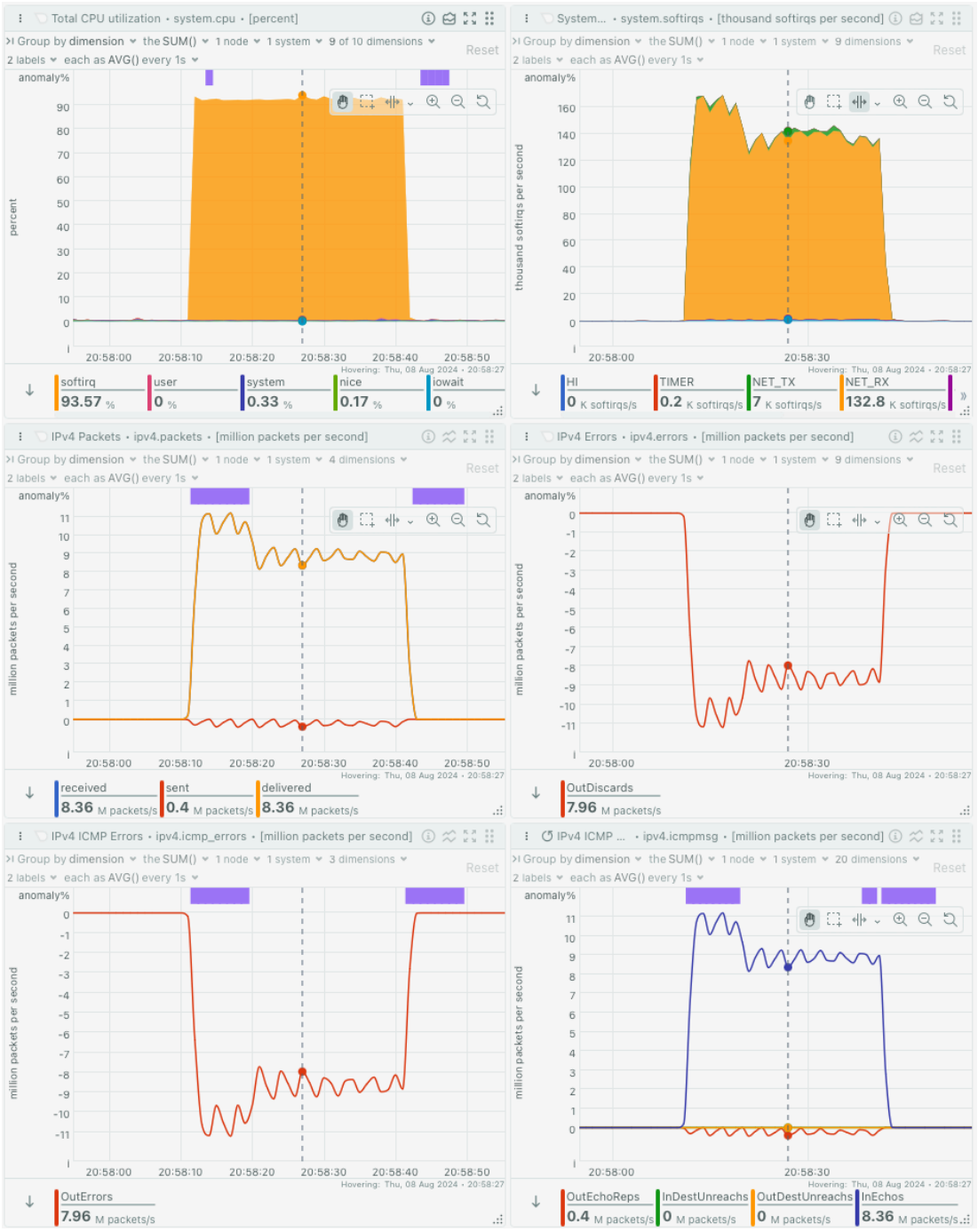

Abbildung 31: ICMP-Flood - Test 1

Wie in der Grafik zu sehen ist, erzeugt auch dieser Angriff eine sehr hohe CPU-Last, hervorgerufen durch ca. 132.800 Software-Interrupts pro Sekunde. Die meisten Antworten auf die Ping-Requests werden zudem von Linux verworfen, was in dem Diagramm ganz unten links zu sehen ist.

Der Test zeigte, dass selbst vergleichsweise simple Angriffsmethoden eine erstaunlich starke Wirkung auf die Systemperformance haben und im schlimmsten Fall zur Beeinträchtigung anderer Prozesse führen können.

5.5 Szenario 3: UDP-Flood

Dieses dritte und letzte Szenario beschäftigt sich mit der Simulation eines UDP-Flood-Angriffs. Hierbei wird der Traffic-Generator so konfiguriert, dass er UDP-Pakete an einen bestimmten Ziel-Port des Servers sendet. Die Paketlänge wird dabei erneut auf 1024 Bytes festgelegt und die Datenrate auf 25 Gbps eingestellt. Die Testdauer beträgt wie zuvor 30 Sekunden.

Das UDP-Protokoll ist ein verbindungsloses Protokoll, das keine Bestätigung der Pakete erfordert. Stattdessen geht der Server bei der Verarbeitung eines eingehenden UDP-Pakets wie folgt vor:

1. Der Server empfängt das UDP-Paket und prüft, ob eine Anwendung auf dem gewünschten Ziel-Port lauscht.
2. Wenn keine Anwendung auf dem Port lauscht, sendet der Server ein ICMP-Paket, um den Sender darüber zu informieren, dass der Port nicht erreichbar ist (*Destination Unreachable*).

Dieser Prozess der Fehlerbehandlung kann ressourcenintensiv werden, wenn der Server eine große Anzahl von UDP-Paketen erhält. Zu Testzwecken wird in diesem Szenario der Zielport 5201 und der Quellport 1234 gewählt, wobei die genaue Portnummer keine Rolle spielt, solange sie nicht von einer Anwendung verwendet wird.

Anders als bei den Tests zuvor, ist die Auswirkung des Angriffs auf den Server weniger dramatisch. Die CPU-Last bleibt relativ konstant bei ca. 28 %, obwohl die Anzahl der Software-Interrupts mit ca. 150.000 pro Sekunde vergleichsweise sehr hoch ist. Aus dem Diagramm ist zudem das zuvor beschriebene Verhalten des Servers zu erkennen, der ICMP-Pakete an den Traffic-Generator sendet, um ihn über die Nichterreichbarkeit des Ziel-Ports zu informieren.

Abbildung 32: UDP-Flood - Test 1

Doch was passiert, wenn nun die Paketgröße verringert wird? Bei der gleichen Datenrate von 25 Gbps und einer Paketlänge von 100 Bytes zeigt sich ein anderes Bild. Die CPU-Last steigt auf sehr konstante 100 % und die Anzahl der Software-Interrupts `NET_RX` auf ca. 250.000 pro Sekunde. Die Menge der gesendeten ICMP-Pakete bleibt jedoch gleich und scheint daher auf ca. 1.000 Pakete pro Sekunde begrenzt zu sein.

5.5 Szenario 3: UDP-Flood

Abbildung 33: UDP-Flood - Test 2

Dieser Test demonstrierte, wie bei vielen Angriffen tatsächlich die Paketrate eine sehr entscheidende Rolle spielt. Schließlich geht es bei DDoS-Angriffen weniger um die effiziente Nutzung der Bandbreite, sondern vielmehr um die Überlastung der Ressourcen des Zielsystems.

Mehr Pakete bedeuten in der Regel mehr Arbeit für die CPU und den Netzwerkstack des Systems, der auf jedes valide eingehende Paket reagieren muss. Insbesondere dann, wenn Adressdaten wie im Falle des getesteten Traffic-Generators für jedes einzelne Paket zufällig generiert werden. Der Server muss dann ggf. eine sehr große Anzahl von Zuständen für jeden Verbindungsversuch managen und eine Fehlerbehandlung durchführen.

6 Fazit und Ausblick

Zum Abschluss dieser Arbeit sollen noch einmal die gewonnen Kenntnisse zusammengefasst und im Kontext der Aufgabenstellung ausgewertet werden. Neben der Formulierung eines Fazits wird ein Ausblick auf mögliche zukünftige Projekte und Erweiterungen des Systems gegeben.

6.1 Fazit

Die Aufgabenstellung dieser Arbeit befasste sich mit automatisierten Netzwerk-Belastungssimulationen mithilfe der Verwendung eines programmierbaren Switches. Dazu wurden zunächst die fachlichen Grundlagen für die Entwicklung eines entsprechenden Systems erarbeitet, um die technischen Voraussetzungen für die Umsetzung der Aufgabenstellung zu definieren. Es wurde hergeleitet, wie das Konzept des modernen Software-Defined Networking (SDN) funktioniert und programmierbare Switches hervorbrachte, deren Data Plane in der Programmiersprache P4 flexibel implementiert und konfiguriert werden kann. Ebenfalls wurde gezeigt, dass die Control Plane dabei weiterhin umfangreiche Möglichkeiten zur Steuerung mittels offener Programmierschnittstellen beibehält.

Um herauszufinden, wie ein Netzwerk zu Testzwecken belastet werden kann, wurde außerdem das Konzept von Network Traffic Generators erläutert und verschiedene Anwendungsfälle für solche Systeme vorgestellt. Dabei wurden die unterschiedlichen Arten von Traffic-Generatoren besprochen und die Vor- und Nachteile existierender hardware- und softwarebasierter Lösungen diskutiert.

Um ein praktisches Anwendungsgebiet für die entwickelte Software zu finden, wurde sich auf die Simulation von DDoS-Angriffen konzentriert, die besonders im Bereich der IT-Sicherheit eine große Rolle spielen. Um diese Form von Angriffen zu simulieren, wurden dabei besonders die technischen Aspekte verschiedener Angriffsstrategien beleuchtet.

Auf Basis all dieser gewonnenen Erkenntnisse wurde ein hardwarebasierter Prototyp mit der einfachen Bezeichnung Traffic-Generator entwickelt, der auf der Intel Tofino-Plattform basiert. Dabei wurde gezeigt, wie genau die Entwicklung eines solchen Systems abläuft und welche technischen Herausforderungen dabei zu bewältigen sind. Der Traffic-Generator ermöglicht die Generierung von Netzwerkverkehr mit einer Vielzahl von Konfigurationsmöglichkeiten und kann so für die Simulation verschiedener Angriffsszenarien verwendet werden. Die Implementierung des Traffic-Generators erfolgte dabei in Python und nutzt die BRI-Schnittstelle des Tofino-Chips zur Steuerung der Data Plane. Für die Implementierung wurde auf die Erkenntnisse und Funktionen einer Reihe existierender Open-Source-Projekte

zurückgegriffen, die die Entwicklung des Prototyps erheblich vereinfacht bzw. erst möglich gemacht haben.

Das Ergebnis ist der Prototyp eines skriptgesteuerten Systems, welches die Generierung von High-Speed-Netzwerkverkehr mit einer Vielzahl von Konfigurationsmöglichkeiten ermöglicht und so für die Simulation verschiedener Angriffsszenarien verwendet werden kann. Die Fähigkeiten des Traffic-Generators wurden anhand von drei verschiedenen Testszenarien demonstriert, die die Auswirkungen von TCP-SYN-Flood-, ICMP-Ping-Flood- und UDP-Flood-Angriffen auf ein Zielsystem untersuchten.

Zusammenfassend lässt sich sagen, dass dieses Projekt keine grundlegende Neuentwicklung darstellt. Vielmehr wurde auf bereits existierende Konzepte und Technologien zurückgegriffen, um ein System zu entwickeln, welches dem Nutzer die Durchführung solcher Tests erleichtern soll. Dahingehend wurde versucht, die Probleme und gewisse Einschränkungen existierender Systeme zu lösen und die Gesamtimplementierung im Bezug auf den Use Case von DDoS-Simulationen zu optimieren. Das Ergebnis ist in erster Linie ein sehr flexibles und einfach erweiterbares System. Es stellt in dem Sinne eine gute Basis für zukünftige Projekte dar, die auf dem entwickelten Prototypen aufbauen können. Die implementierten Funktionen können daher als Framework für die Entwicklung weiterer Anwendungen und Tools dienen, die sich mit der Simulation von Netzwerkverkehr und der Analyse von Netzwerkperformance beschäftigen. Außerdem wurde Wert auf die einfache Ausführung und Konfiguration des Systems gelegt, um es auch für weniger erfahrene Nutzer zugänglich zu machen. So kommt der Traffic-Generator mit einer Vielzahl von Standardkonfigurationen und umfangreichen Remote-Funktionen daher, welche es dem Benutzer ermöglichen, Tests bequem auf dem eigenen System anzustoßen und Konfigurationen und Code auf den Ziel-Switch zu übertragen. Abschließend zeichnet sich der Traffic-Generator neben seiner Flexibilität und Benutzerfreundlichkeit vor allem durch einen gewissen Mangel an Komplexität aus, welcher bei existierenden Systemen oft vermehrt zu finden ist und vermutlich die Hemmschwelle für die tatsächliche Nutzung solcher Systeme stark erhöht.

Rückblickend sind jedoch auch einige Aspekte des Projekts kritisch zu betrachten. So ist die Implementierung des Traffic-Generators zwar funktional, jedoch noch nicht vollständig ausgereift und in vielerlei Hinsicht noch verbesserungswürdig. So sind viele Funktionen, welche sich mit der Konfiguration des Zielsystems auseinandersetzen, eher umständlich umgesetzt. Zusätzlich fehlen dem Traffic-Generator eine Reihe von Möglichkeiten zur komplexeren Variation der Traffic-Eigenschaften, gerade was die Datenrate über Zeit betrifft. Auch die Auswertung der Testergebnisse könnte durchaus noch verbessert werden. Am Ende waren es hauptsächlich zwei Faktoren, welche die Entwicklung des Projekts maßgeblich beeinflusst haben. Zum einen war es die begrenzte Zeit, die für die Entwicklung zur Verfügung stand, gerade wenn die umfangreiche Komplexität des gesamten Technologie-Stacks betrachtet wird. Zum anderen gibt es aufgrund von Intels Kommunikationspolitik quasi keine frei verfügbare, vollumfängliche Dokumentation für die verwendete Entwicklungsumgebung und deren Softwarebibliotheken. Dies führte dazu, dass ein erhebliches Zeitbudget der Arbeit für aufwendiges Reverse Engineering bestehender Projekte und Trial-and-Error-Tests aufgewendet werden musste, um einen gewissen Basisstand an Funktionalität zu erreichen.

Meine persönliche Hoffnung besteht darin, dass mehr Aspekte der P4-Programmierung und der damit verbundenen Schnittstellen und Softwarebibliotheken weiter standardisiert werden, um so ein offeneres Ökosystem für Entwickler zu schaffen.

6.2 Ausblick

In diesem letzten Abschnitt der Arbeit wird auf mögliche zukünftige Projekte und Erweiterungen des Systems eingegangen. Dabei wird aufgezeigt, wie der Traffic-Generator weiterentwickelt und verbessert werden kann, um die Leistungsfähigkeit und Funktionalität des Systems weiter zu steigern.

Andere Projekte und aktuelle Forschungsergebnisse haben gezeigt, dass die Tofino-Plattform im Bezug auf die Generierung und vor allem die Auswertung von Netzwerkverkehr noch sehr viel, in dieser Arbeit nicht behandeltes, Potenzial bietet. Projekte wie P4TG zeigen, wie durch den Einsatz von Monitoring-Frames und cleveren Registerabfragen sehr detaillierte Informationen über den generierten Traffic gewonnen werden können. Diese Eigenschaft ist besonders für die Reproduzierbarkeit bestimmter Testergebnisse und die Fehlersuche in komplexen Netzwerktopologien von Vorteil.

Ein weiterer Ansatz zur Verbesserung des Systems wäre die Verwendung von sogenannten *Multicast-Groups*. Dieses Feature eines P4-Switches erlaubt es, Traffic von einem Eingangsport einfach auf mehrere Ausgangsports zu replizieren. Je nach Hardware-Modell des Switches ist dadurch eine erhebliche Steigerung der Gesamtdatenrate des generierten Traffics möglich, da mehrere Ports gleichzeitig mit Daten versorgt werden können. Die Umsetzung dieser Funktion ist tatsächlich vergleichsweise simpel, war allerdings aufgrund des begrenzten Zeitrahmens einer Abschlussarbeit nicht mehr umsetzbar.

Ein anderes interessantes Projekt wäre die Integration des Traffic-Generators in eine bestehende Netzwerküberwachungslösung, also ein weiterer Schritt in Richtung umfassender Automatisierung des Simulationsprozesses. So könnten die Ergebnisse einer Simulation nach Vorbild des OpenTelemetry-Projektes [Ope24] exportierbar und für andere Anwendungen nutzbar gemacht werden. Jedoch müssten dafür vermutlich sehr umfangreiche Management- und Control-Plane-Funktionen implementiert werden, welche ebenfalls das Deployment von Monitoring-Sensor-Software auf den Zielsystemen erfordern.

Aus Entwicklersicht wäre es vermutlich sinnvoll, das System auf Basis des Projekts *Open Traffic Generator* (OTG) [Ope24] weiterzuentwickeln. OTG ist ein Open-Source-Projekt, welches einen API-Standard für die Steuerung von Traffic-Generatoren definiert und so eine bessere Interoperabilität verschiedener Systeme ermöglicht. Beispielsweise wird diese API heute bereits von kommerziellen Traffic-Generatoren wie Keysights *Elastic Network Generator* [Key24] und dessen offener Community-Version *Ixia-c* [ope24] unterstützt.

An dieser Stelle könnten vermutlich noch viele weitere Ideen und Vorschläge für die Verbesserung des Systems genannt werden. Es bleibt daher zu hoffen, dass die in dieser Arbeit entwickelten Konzepte und Ideen als Grundlage für zukünftige Projekte dienen können.

Abkürzungsverzeichnis

***ACK* – Acknowledgment**: „ACK ist ein TCP-Flag, das den Empfang eines Pakets bestätigt."

***ALU* – Arithmetic Logic Unit**: „Eine ALU ist ein Bestandteil der CPU, der arithmetische und logische Operationen ausführt."

***AN* – Auto Negotiation**: „Auto Negotiation ist ein Ethernet-Mechanismus zur Aushandlung von Verbindungsparametern zwischen Geräten."

***API* – Application Programming Interface**: „Eine API ist eine Schnittstelle, die es Softwarekomponenten ermöglicht, miteinander zu kommunizieren."

***ASIC* – Application-specific Integrated Circuit**: „Ein ASIC ist ein integrierter Schaltkreis, der speziell für eine bestimmte Anwendung entwickelt wurde."

***BF-SDE* – Barefoot Software Development Environment**: „BF-SDE ist eine Entwicklungsumgebung für das Programmieren von Intel-Barefoot-Netzwerkchips."

***BIOS* – Basic Input/Output System**: „Das BIOS ist die Firmware eines Computers, die die Hardware initialisiert und das Laden des Betriebssystems steuert."

***BRI* – Barefoot Runtime Interface**: „BRI ist eine Schnittstelle zur Verwaltung von Barefoot-Netzwerkgeräten."

***C2* – Command-and-Control**: „C2 ist ein Begriff, der in der Cybersecurity verwendet wird, um die Infrastruktur zu beschreiben, die Angreifer zur Steuerung von Malware oder infizierten Systemen nutzen."

***CLI* – Command Line Interface**: „CLI ist eine textbasierte Benutzerschnittstelle zur Eingabe von Befehlen."

***CPU* – Central Processing Unit**: „Die CPU ist die zentrale Recheneinheit eines Computers, die Befehle ausführt und Daten verarbeitet."

***DDoS* – Distributed Denial of Service**: „Ein DDoS-Angriff ist eine Art von Cyberangriff, bei dem mehrere Systeme verwendet werden, um einen Server mit Anfragen zu überlasten und ihn dadurch außer Betrieb zu setzen."

***DoS* – Denial of Service**: „Ein DoS-Angriff zielt darauf ab, die Verfügbarkeit eines Netzwerks oder Dienstes zu stören, indem eine Flut von Anfragen gesendet wird, um die Ressourcen zu überlasten."

***DPDK* – Data Plane Development Kit**: „DPDK ist eine Sammlung von Bibliotheken und Treibern, die eine schnelle Paketverarbeitung auf Netzwerkgeräten ermöglicht."

***DPI* – Deep Packet Inspection**: „DPI ist eine Technik, die den Inhalt von Datenpaketen analysiert, um detaillierte Einblicke in den Netzwerkverkehr zu gewinnen."

***ECN* – Explicit Congestion Notification**: „ECN ist eine Erweiterung von TCP/IP, die Netzwerkknoten ermöglicht, Staus zu signalisieren, ohne Pakete zu verwerfen."

***FEC* – Forward Error Correction**: „FEC ist eine Methode zur Fehlerkorrektur bei der Datenübertragung durch Hinzufügen von Redundanz."

***FPGA* – Field-programmable Gate Array**: „Ein FPGA ist ein integrierter Schaltkreis, der nach der Herstellung vom Nutzer programmiert werden kann."

***gRPC* – gRPC Remote Procedure Calls**: „gRPC ist ein leistungsstarkes Remote-Prozeduraufruf-Framework, das es Clients und Servern ermöglicht, direkt miteinander zu kommunizieren."

***HTTP* – Hypertext Transfer Protocol**: „HTTP ist das grundlegende Protokoll für die Übertragung von Webseiten über das Internet."

***HTTPS* – Hypertext Transfer Protocol Secure**: „HTTPS ist ein Protokoll zur sicheren Übertragung von Daten im World Wide Web."

***ICMP* – Internet Control Message Protocol**: „ICMP wird verwendet, um Fehlermeldungen und operationale Informationen in IP-Netzwerken zu senden."

***ICMP* – Internet Control Message Protocol**: „ICMP ist ein Netzwerkprotokoll, das zur Diagnose von Netzwerkverbindungen verwendet wird."

***IoT* – Internet of Things**: „Das IoT beschreibt das Netzwerk von physischen Geräten, Fahrzeugen, Gebäuden und anderen Objekten, die mit Sensoren, Software und Netzwerkkonnektivität ausgestattet sind."

***IP* – Internet Protocol**: „IP ist das grundlegende Protokoll für die Adressierung und den Versand von Datenpaketen über das Internet."

***IPv6* – Internet Protocol Version 6**: „IPv6 ist die Nachfolgeversion von IPv4 und bietet eine größere Anzahl an IP-Adressen."

***JSON* – JavaScript Object Notation**: „JSON ist ein leichtes Datenformat, das zum Austausch von Daten zwischen einem Server und einem Web-Client verwendet wird."

***LTS* – Long Term Support**: „LTS bezieht sich auf Softwareversionen, die über einen längeren Zeitraum unterstützt werden."

***LTS* – Long Term Support**: „LTS bezieht sich auf Softwareversionen, die über einen längeren Zeitraum unterstützt werden."

***MAC* – Media Access Control**: „MAC-Adressen sind eindeutige Kennungen, die Netzwerkschnittstellen zur Kommunikation zugewiesen werden."

***MAT* – Match-Action-Tables**: „Match-Action-Tables werden in Netzwerkswitches verwendet, um Entscheidungen auf Basis von eingehenden Datenpaketen zu treffen."

***NAT* – Network Address Translation**: „NAT ist eine Methode zur Änderung von IP-Adressen, um private Netzwerke mit dem Internet zu verbinden."

***NIC* – Network Interface Card**: „Eine NIC ist eine Hardwarekomponente, die einen Computer mit einem Netzwerk verbindet."

***OCP* – Open Compute Project**: „Das Open Compute Project ist eine Gemeinschaft, die auf die Entwicklung offener Standards für Rechenzentrums- und Netzwerkinfrastrukturen abzielt."

***ONIE* – Open Network Install Environment**: „ONIE ist eine Open-Source-Umgebung für die Installation von Netzwerkbetriebssystemen auf Netzwerkswitches."

***OS* – Operating System**: „Ein Betriebssystem (OS) ist die grundlegende Software, die die Hardware eines Computers steuert."

***OSI* – Open Systems Interconnection**: „Das OSI-Modell ist ein konzeptionelles Rahmenwerk, das die Netzwerkkommunikation in sieben Schichten unterteilt."

***PISA* – Protocol Independent Switch Architecture**: „PISA ist eine flexible Netzwerkarchitektur, die unabhängige Verarbeitung von Netzwerkprotokollen ermöglicht."

***PUSH* – Push**: „PUSH ist ein TCP-Flag, das angibt, dass die empfangenen Daten sofort verarbeitet werden sollen."

***QSFP* – Quad Small Form-factor Pluggable**: „QSFP ist ein Transceiver-Standard, der vier Kanäle mit hoher Datenrate unterstützt."

***RAM* – Random Access Memory**: „RAM ist ein flüchtiger Speicher, der Daten temporär speichert, auf die schnell zugegriffen werden kann."

***RST* – Reset**: „RST ist ein TCP-Flag, das zur sofortigen Beendigung einer Verbindung verwendet wird."

***SAI* – Switch Abstraction Interface**: „SAI ist eine API, die die Abstraktion von Switch-Hardware ermöglicht, um die Interoperabilität zwischen verschiedenen Switches und Netzwerksystemen zu fördern."

***SDN* – Software-Defined Networking**: „SDN ist ein Ansatz für das Netzwerkmanagement, der die Steuerung der Netzwerkinfrastruktur von der Hardware trennt."

***SFP* – Small Form-factor Pluggable**: „SFP ist ein kompakter, austauschbarer Transceiver, der in Netzwerkschnittstellen verwendet wird."

***SRAM* – Static Random-Access Memory**: „SRAM ist eine Art flüchtiger Speicher, der schneller und zuverlässiger als DRAM ist, aber mehr Platz benötigt."

***SSH* – Secure Shell**: „SSH ist ein Protokoll zur sicheren Verwaltung und Übertragung von Daten über unsichere Netzwerke."

***SYN* – Synchronize**: „SYN ist ein Flag in TCP-Paketen, das zur Initialisierung einer Verbindung zwischen zwei Hosts verwendet wird."

***TCAM* – Ternary Content-Addressable Memory**: „TCAM ist ein spezieller Speicher, der für schnelle Suchvorgänge in Routing- und Switching-Tabellen verwendet wird."

***TCP* – Transmission Control Protocol**: „TCP ist ein zuverlässiges, verbindungsorientiertes Protokoll, das den zuverlässigen Austausch von Daten zwischen Computern ermöglicht."

***TNA* – Tofino Native Architecture**: „TNA ist die native Architektur von Tofino-Netzwerkchips."

***TTL* – Time to Live**: „TTL ist ein Feld in IP-Paketen, das die maximale Lebensdauer eines Pakets im Netzwerk angibt."

***UDP* – User Datagram Protocol**: „UDP ist ein verbindungsloses Protokoll, das eine schnelle Übertragung von Daten ohne Gewährleistung der Zuverlässigkeit ermöglicht."

***VM* – Virtuelle Maschine**: „Eine Virtuelle Maschine (VM) ist eine Software, die eine vollständige Computerumgebung emuliert und es ermöglicht, ein Betriebssystem und Anwendungen unabhängig von der Hardware auszuführen."

Literaturverzeichnis

[ABG23] Adeleke, Oluwamayowa Ade ; Bastin, Nicholas ; Gurkan, Deniz: Network Traffic Generation: A Survey and Methodology. In: *ACM Computing Surveys* Bd. 55 (2023), Nr. 2, S. 1–23

[APS24] *APS-Networks/bfrt-helper*. URL https://github.com/APS-Networks/bfrt-helper. - abgerufen am 2024-08-12

[Abd24] AbdullahBell: *Tutorial: Azure DDoS Protection simulation testing*. URL https://learn.microsoft.com/en-us/azure/ddos-protection/test-through-simulations. - abgerufen am 2024-08-11

[Ali23] Aliyev, Rashad: DDoS Simulation: Empowering Targets through Simulated Attacks.. In: *2023 IEEE 17th International Conference on Application of Information and Communication Technologies (AICT)*. Baku, Azerbaijan : IEEE, 2023 — ISBN 9798350303568, S. 1–4

[Ama24] Amazon Web Services, Inc.: *Öffentliche DDoS-Testrichtlinie – Amazon Web Services (AWS)*. URL https://aws.amazon.com/de/security/ddos-simulation-testing/. - abgerufen am 2024-08-11

[Ari+03] Ari, I. ; Hong, B. ; Miller, E.L. ; Brandt, S.A. ; Long, D.D.E.: Managing flash crowds on the Internet.. In: *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*. Orlando, FL, USA : IEEE Comput. Soc, 2003 — ISBN 978-0-7695-2039-1, S. 246–249

[Ast24] Astanin, Sergey: *astanin/python-tabulate*. URL https://github.com/astanin/python-tabulate. - abgerufen am 2024-08-12

[BDP10] Botta, Alessio ; Dainotti, Alberto ; Pescape, Antonio: Do you trust your software-based traffic generator?. In: *IEEE Communications Magazine* Bd. 48 (2010), Nr. 9, S. 158–165

[Bri23] Brian Krebs: *Thinking of Hiring or Running a Booter Service? Think Again. – Krebs on Security*. URL https://krebsonsecurity.com/2023/01/thinking-of-hiring-or-running-a-booter-service-think-again/. - abgerufen am 2024-08-11

[CM23] Chaudhary, Shubhankar ; Mishra, Pramod Kumar: DDoS attacks in Industrial IoT: A survey. In: *Computer Networks* Bd. 236 (2023), S. 110015–110016

[Can24] Canonical: *Enterprise Open Source and Linux*. URL https://ubuntu.com/. - abgerufen am 2024-08-12

[Che+] Chen, Yanqing ; Tian, Bingchuan ; Tian, Chen ; Dai, Li ; Zhou, Yu ; Ma, Mengjing ; Tang, Ming ; Zheng, Hao ; u. a.: Norma: Towards Practical Network Load Testing

[Cis24] Cisco Systems, Inc.: *Solutions - Cisco Silicon One G200 Data Sheet*. URL https://www.cisco.com/c/en/us/solutions/collateral/silicon-one/silicon-one-g200-ds.html. - abgerufen am 2024-08-11

[Clo22] Cloudflare, Inc.: *Five Best Practices for Mitigating DDoS Attacks: How to Defend Against Rapidly Evolving Distributed Denial-of-Service Threats and Address Vulnerabilities at Every Layer*. URL https://www.cloudflare.com/static/d442dfee7ea56f899d8df461bb7a077f/BDES-2587-Design-Wrap-Refreshed-DDoS-White-Paper-Letter.pdf

[Clo23] Cloudflare, Inc.: *HTTP/2 Zero-Day vulnerability results in record-breaking DDoS attacks*. URL https://blog.cloudflare.com/zero-day-rapid-reset-http2-record-breaking-ddos-attack. - abgerufen am 2024-08-11

[Clo24] Cloudflare, Inc.: *What is an IP stresser? | DDoS booters*. URL https://www.cloudflare.com/learning/ddos/ddos-attack-tools/ddos-booter-ip-stresser/. - abgerufen am 2024-08-11

[Clo24] Cloudflare, Inc.: *Simulating test DDoS attacks · Cloudflare DDoS Protection docs*. URL https://developers.cloudflare.com/ddos-protection/reference/simulate-ddos-attack/. - abgerufen am 2024-08-11

[Clo24] Cloudflare, Inc.: *HTTP/2 Rapid Reset Attack Protection*. URL https://www.cloudflare.com/h2/. - abgerufen am 2024-08-11

[Com24] *Comparing TRex Advanced Stateful performance to Linux NGINX*. URL https://trex-tgn.cisco.com/trex/doc/trex_astf_vs_nginx.html. - abgerufen am 2024-08-11

[Cov+09] Covington, G. Adam ; Gibb, Glenn ; Lockwood, John W. ; Mckeown, Nick: A Packet Generator on the NetFPGA Platform.. In: *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. Napa, CA, USA : IEEE, 2009 — ISBN 978-0-7695-3716-0, S. 235–238

[DM04] Douligeris, Christos ; Mitrokotsa, Aikaterini: DDoS attacks and defense mechanisms: classification and state-of-the-art. In: *Computer Networks* Bd. 44 (2004), Nr. 5, S. 643–666

[DPD24] DPDK Project: *About*. URL https://www.dpdk.org/about/. - abgerufen am 2024-08-11

[Dut19] Dutt, Dinesh G.: *Cloud native data center networking: architecture, protocols, and tools*. First edition.. Beijing Boston Farnham : O'Reilly, 2019 — ISBN 978-1-4920-4560-1

[Emm+15] Emmerich, Paul ; Gallenmüller, Sebastian ; Raumer, Daniel ; Wohlfart, Florian ; Carle, Georg: MoonGen: A Scriptable High-Speed Packet Generator.. In: *Proceedings of the 2015 Internet Measurement Conference*, 2015, S. 275–287

[Emm+17] Emmerich, Paul ; Gallenmuller, Sebastian ; Antichi, Gianni ; Moore, Andrew W. ; Carle, Georg: Mind the Gap - A Comparison of Software Packet Generators.. In: *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. Beijing, China : IEEE, 2017 — ISBN 978-1-5090-6386-4, S. 191–203

[Emm24] Emmerich, Paul: *emmericp/MoonGen*. URL https://github.com/emmericp/MoonGen. - abgerufen am 2024-08-11

[Eur23] European Union Agency for Cybersecurity.: *ENISA threat landscape 2023: July 2022 to June 2023.*. LU : Publications Office, 2023

[Eur23] European Union Agency for Cybersecurity.: *ENISA threat landscape for DoS attack: January 2022 to August 2023.*. LU : Publications Office, 2023

[Fil24] FilipoGC: *FilipoGC/PIPO-TG*. URL https://github.com/FilipoGC/PIPO-TG. - abgerufen am 2024-06-04

[Gai20] Gai, Silvano: *Building a future-proof cloud infrastructure: a unified architecture for network, security and storage services*. 1. Aufl.. Hoboken : Pearson Education, Inc, 2020 — ISBN 978-0-13-662409-7

[Goo24] Google LLC: *Protocol Buffers*. URL https://protobuf.dev/. - abgerufen am 2024-08-11

[Goo24] Google LLC: *How it works: The novel HTTP/2 'Rapid Reset' DDoS attack.* URL https://cloud.google.com/blog/products/identity-security/how-it-works-the-novel-http2-rapid-reset-ddos-attack. - abgerufen am 2024-08-11

[Gre20] Gregg, Brendan: *Systems performance: enterprise and the cloud, Addison-wesley professional computing series*. Second.. Boston : Addison-Wesley, 2020 — ISBN 978-0-13-682015-4

[Gup+23] Gupta, Sahil ; Gosain, Devashish ; Kwon, Minseok ; Acharya, Hrishikesh B: DeeP4R: Deep Packet Inspection in P4 using Packet Recirculation.. In: *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023, S. 1–10

[Has24] HashiCorp: *Vagrant by HashiCorp.* URL https://www.vagrantup.com/. - abgerufen am 2024-08-12

[Hau+23] Hauser, Frederik ; Häberle, Marco ; Merling, Daniel ; Lindner, Steffen ; Gurevich, Vladimir ; Zeiger, Florian ; Frank, Reinhard ; Menth, Michael: A Survey on Data Plane Programming with P4: Fundamentals, Advances, and Applied Research. In: *Journal of Network and Computer Applications* Bd. 212 (2023), S. 103561–103562

[Int21] Intel Corporation: *P416 Intel® Tofino™ Native Architecture – Public Version.* URL https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf

[Int24] Intel Corporation: *Intel® P4 Studio.* URL https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html. - abgerufen am 2024-08-11

[KR21] Kurose, James F. ; Ross, Keith W.: *Computer networking: a top-down approach*. Eighth edition.. Hoboken : Pearson, 2021 — ISBN 978-0-13-668155-7

[KSK19] Kundel, Ralf ; Siegmund, Fridolin ; Koldehofe, Boris: How to measure the speed of light with programmable data plane hardware?.. In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. Cambridge, United Kingdom : IEEE, 2019 — ISBN 978-1-72814-387-3, S. 1–2

[Ken22] Kenny Lei: Cisco UADP & Silicon One ASIC Architecture & Innovations, 2022

[Key24] Keysight Technologies: *Network Test Hardware | Keysight.* URL https://www.keysight.com/us/en/products/network-test/network-test-hardware.html. - abgerufen am 2024-08-11

[Key24] *Keysight Elastic Network Generator | Keysight.* URL https://www.keysight.com/us/en/products/network-test/protocol-load-test/keysight-elastic-network-generator.html. - abgerufen am 2024-08-12

[Kun+20] Kundel, Ralf ; Siegmund, Fridolin ; Blendin, Jeremias ; Rizk, Amr ; Koldehofe, Boris: P4STA: High Performance Packet Timestamping with Programmable Packet Processors (2020)

[Kun+22] Kundel, Ralf ; Siegmund, Fridolin ; Hark, Rhaban ; Rizk, Amr ; Koldehofe, Boris: Network Testing Utilizing Programmable Network Hardware. In: *IEEE Communications Magazine* Bd. 60 (2022), Nr. 2, S. 12–17

[Kun24] Kundel, Ralf: *ralfkundel/P4STA*. URL https://github.com/ralfkundel/P4STA. - abgerufen am 2024-08-12

[LHM23] Lindner, Steffen ; Häberle, Marco ; Menth, Michael: P4TG: 1 Tb/s Traffic Generation for Ethernet/IP Networks. In: *IEEE Access* Bd. 11 (2023), S. 17525–17535

[Lam24] LambdaTest: *What is Load Testing? Complete Tutorial With Best Practices.* URL https://www.lambdatest.com/learning-hub/load-testing. - abgerufen am 2024-08-12

[Loo21] *Looking back on 10 years of building world-class data centers.* URL https://tech.facebook.com/engineering/2021/11/10-years-world-class-data-centers/. - abgerufen am 2024-08-11

[MMS13] Molnar, Sandor ; Megyesi, Peter ; Szabo, Geza: How to validate traffic generators?.. In: *2013 IEEE International Conference on Communications Workshops (ICC)*. Budapest, Hungary : IEEE, 2013 — ISBN 978-1-4673-5753-1, S. 1340–1344

[Mah+17] Mahjabin, Tasnuva ; Xiao, Yang ; Sun, Guang ; Jiang, Wangdong: A survey of distributed denial-of-service attack, prevention, and mitigation techniques. In: *International Journal of Distributed Sensor Networks* Bd. 13 (2017), Nr. 12, S. 155014771774146

[Mem24] Memarzanjany, Fereydoun: *Thraetaona/Blitzping.* URL https://github.com/Thraetaona/Blitzping. - abgerufen am 2024-08-11

[NVI24] *NVIDIA Cumulus Linux Architecture.* URL https://www.nvidia.com/en-us/networking/ethernet-switching/cumulus-linux/. - abgerufen am 2024-08-11

[Net24] Netdata Inc.: *Netdata: The open source observability platform everyone needs.* URL https://www.netdata.cloud/. - abgerufen am 2024-08-12

[Noa18] Noa Zilberman: P4 Tutorial, 2018

[Npi24] *Nping — Network packet generation tool & ping utility.* URL https://nmap.org/nping/. - abgerufen am 2024-08-11

[ONF23] ONF: *ONF Merges Market Leading Portfolio of Open Source Networking Projects into the Linux Foundation.* URL https://opennetworking.org/news-and-events/press-releases/onf-merges-market-leading-portfolio-of-open-source-networking-projects-into-the-linux-foundation/. - abgerufen am 2024-08-11

[ONF24] *ONF Programmable Networks Projects.* URL https://opennetworking.org/onf-sdn-projects/. - abgerufen am 2024-08-11

[Odo16] Odom, Wendell: *CCENT/CCNA ICND1 100-105 official cert guide.* Academic edition.. Indianapolis, IN : Cisco Press, 2016 — ISBN 978-1-58720-597-2

[Ope24] *Open Compute Project.* URL https://www.opencompute.org/projects/networking. - abgerufen am 2024-08-11

[Ope24] OpenTelemetry Authors: *OpenTelemetry.* URL https://opentelemetry.io/. - abgerufen am 2024-08-12

[Ope24] *Open Traffic Generator APIs & Data Models.* URL https://otg.dev/. - abgerufen am 2024-08-12

[Ove24] *Overview — Open Network Install Environment documentation.* URL https://opencomputeproject.github.io/onie/overview/index.html. - abgerufen am 2024-08-11

[P4 24] *P4 – Language Consortium.* URL https://p4.org/. - abgerufen am 2024-08-11

[PBZ22] Parsonson, Christopher W.F. ; Benjamin, Joshua L. ; Zervas, Georgios: Traffic generation for benchmarking data centre networks. In: *Optical Switching and Networking* Bd. 46 (2022), S. 100695–100696

[PD22] Peterson, Larry L. ; Davie, Bruce S.: *Computer networks: a systems approach.* Sixth edition.. Cambridge, Massachusetts : Morgan Kaufmann Publishers, an imprint of Elsevier, 2022 — ISBN 978-0-12-818200-0

[Pas24] Pascal Bast: *Pascal Bast / Tofino Traffic Generator · GitLab.* URL https://gitlab.rz.htw-berlin.de/s0558899/tofino-traffic-generator. - abgerufen am 2024-08-12

[Pau23] Paul Krzyzanowski: CS 417 Week 2: – DISTRIBUTED SYSTEMS - Point-to-point communication: Remote Procedure Calls, 2023

[Pet+21] Peterson, Larry L. ; Cascone, Carmelo ; O´Connor, Brian ; Vachuska, Thomas ; Davie, Bruce: *Software-Defined Networks: A Systems Approach*. Wroclaw : Systems Approach LLC, 2021 — ISBN 978-1-73647-210-1

[Pin24] *Ping of Death*. URL https://insecure.org/sploits/ping-o-death.html. - abgerufen am 2024-08-11

[SOG07] Sicker, Douglas C. ; Ohm, Paul ; Grunwald, Dirk: Legal issues surrounding monitoring during network research.. In: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. San Diego California USA : ACM, 2007 — ISBN 978-1-59593-908-1, S. 141–148

[San24] Sanfilippo, Salvatore: *antirez/hping*. URL https://github.com/antirez/hping. - abgerufen am 2024-08-11

[Son24] *Sonic Foundation – Linux Foundation Project*. URL https://sonicfoundation.dev/. - abgerufen am 2024-08-11

[Spi24] Spirent Communications: *Ethernet Testing - TestCenter Appliances and Modules - Spirent*. URL https://www.spirent.com/products/testcenter-hardware. - abgerufen am 2024-08-11

[Spi24] Spirent Communications: *Measure, Validate, Deploy -TestCenter - Spirent*. URL https://www.spirent.com/products/testcenter-ethernet-ip-cloud-test. - abgerufen am 2024-08-12

[Swa+20] Swann, Matthew ; Rose, Joseph ; Bendiab, Gueltoum ; Shiaeles, Stavros ; Savage, Nick: A Comparative Study of Traffic Generators: Applicability for Malware Detection Testbeds. In: *Journal of Internet Technology and Secured Transactions* Bd. 8 (2020), Nr. 1, S. 705–713

[TB22] Thomson, Martin ; Benfield, Cory: *HTTP/2*, 2022

[TFW21] Tanenbaum, Andrew S. ; Feamster, Nick ; Wetherall, David: *Computer networks*. Sixth edition, global edition.. Harlow, United Kingdom : Pearson, 2021 — ISBN 978-1-292-37406-2

[Tcp24] Tcpdump Group: *tcpdump(1) man page | TCPDUMP & LIBPCAP*. URL https://www.tcpdump.org/manpages/tcpdump.1.html. - abgerufen am 2024-08-12

[The21] The P4.org Architecture Working Group: *P416 Portable Switch Architecture (PSA)*. URL https://p4.org/p4-spec/docs/PSA.pdf. - abgerufen am 2024-08-11

[The24] The P4 Language Consortium: *P4~16~ Language Specification*. URL https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. - abgerufen am 2024-08-11

[The24] The P4.org API Working Group: *P4Runtime Specification*. URL https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html. - abgerufen am 2024-08-11

[The24] *The LAND attack (IP DOS)*. URL https://insecure.org/sploits/land.ip.DOS.html. - abgerufen am 2024-08-11

[Wan+11] Wang, Jie ; Phan, Raphael C.-W. ; Whitley, John N. ; Parish, David J.: DDoS attacks traffic and Flash Crowds traffic simulation with a hardware test center platform.. In: *2011 World Congress on Internet Security (WorldCIS-2011)*. London : IEEE, 2011 — ISBN 978-1-4244-8879-7 978-0-9564263-7-6, S. 15–20

[Wir24] Wireshark Foundation: *Wireshark · Go Deep*. URL http://localhost:3000/. - abgerufen am 2024-08-12

[Zho+19] Zhou, Yu ; Xi, Zhaowei ; Zhang, Dai ; Wang, Yangyang ; Wang, Jinqiu ; Xu, Mingwei ; Wu, Jianping: HyperTester: high-performance network testing driven by programmable swit-

ches.. In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. Orlando Florida : ACM, 2019 — ISBN 978-1-4503-6998-5, S. 30–43

[cis24] *cisco-system-traffic-generator/trex-core.* URL https://github.com/cisco-system-traffic-generator/trex-core. - abgerufen am 2024-08-11

[esn24] *esnet/iperf.* URL https://github.com/esnet/iperf. - abgerufen am 2024-08-11

[gRP24] gRPC Authors: *gRPC.* URL https://grpc.io/. - abgerufen am 2024-08-11

[grp24] *grpc/grpc.* URL https://github.com/grpc/grpc. - abgerufen am 2024-08-12

[hyp23] *hypertester/hypertester.* URL https://github.com/hypertester/hypertester. - abgerufen am 2024-08-12

[ift24] *iftop(8): bandwidth usage on interface by host - Linux man page.* URL https://linux.die.net/man/8/iftop. - abgerufen am 2024-08-12

[ope24] *opencomputeproject/OpenNetworkLinux.* URL https://github.com/opencomputeproject/OpenNetworkLinux. - abgerufen am 2024-08-11

[ope24] *open-traffic-generator/ixia-c.* URL https://github.com/open-traffic-generator/ixia-c. - abgerufen am 2024-08-12

[par24] *paramiko/paramiko.* URL https://github.com/paramiko/paramiko. - abgerufen am 2024-08-12

[sec24] *secdev/scapy.* URL https://github.com/secdev/scapy. - abgerufen am 2024-08-12

[uni24] *uni-tue-kn/P4TG.* URL https://github.com/uni-tue-kn/P4TG. - abgerufen am 2024-08-12

[z/O23] *z/OS Basic Skills.* URL https://www.ibm.com/docs/en/zos-basic-skills?topic=vmt-who-uses-mainframes-why-do-they-do-it. - abgerufen am 2024-08-11

Abbildungsverzeichnis

Tabellenverzeichnis