Technische Universität Ilmenau
Fakultät Informatik und Automatisierung
Fachgebiet Telematik/Rechnernetze

TECHNISCHE UNIVERSITÄT
ILMENAU

Erste Review zum Thema

Abwehr von Denial-of-Service-Angriffen durch effiziente User-Space Paketverarbeitung

AEGIS

Autoren:

Fabienne Göpfert
Felix Husslein
Johannes Lang
Jakob Lerch
Tim Häußler
Robert Jeutter
Leon Leisten
Tobias Scholz

Betreuer: Martin Backhaus

Entwurf 26.Mai 2021

Inhaltsverzeichnis

Kapitel 1

Problemstellung

Denial-of-Service-Angriffe stellen eine ernstzunehmende Bedrohung dar.
Im digitalen Zeitalter sind viele Systeme über das Internet miteinander verknüpft. Viele Unternehmen, Krankenhäusern und Behörden sind dadurch zu beliebten Angriffszielen geworden [1]. Motive für solche Angriffe sind finanzielle oder auch politische Gründe.
Bei DoS[1]- und DDoS[2]-Attacken werden Server und Infrastrukturen mit einer Flut sinnloser Anfragen so stark überlastet, dass sie von ihrem normalen Betrieb abgebracht werden. Daraus kann resultieren, dass Nutzer die angebotenen Dienste nicht mehr erreichen und Daten bei dem Angriff verloren gehen können.
Hierbei können schon schwache Rechner großen Schaden bei deutlich leistungsfähigeren Empfängern auslösen. In Botnetzen können die Angriffe von mehreren Computern gleichzeitig, koordiniert und aus verschiedensten Netzwerken stammen [2].
Das Ungleichgewicht zwischen Einfachheit bei der Erzeugung von Angriffsverkehr gegenüber komplexer und ressourcenintensiver DoS-Abwehr verschärft das Problem zusätzlich. Obwohl gelegentlich Erfolge im Kampf gegen DoS-Angriffe erzielt werden (z.B. Stilllegung einiger großer „DoS-for-Hire" Webseiten), vergrößert sich das Datenvolumen durch DoS-Angriffe stetig weiter. Allein zwischen 2014 und 2017 hat sich die Frequenz von DoS-Angriffen um den Faktor 2,5 vergrößert und das Angriffsvolumen verdoppelt sich fast jährlich [3]. Die Schäden werden weltweit zwischen 20.000 und 40.000 US-Dollar pro Stunde geschätzt [4].
Im Bereich kommerzieller DoS-Abwehr haben sich einige Ansätze hervorgetan (z.B. Project Shield [5], Cloudflare [6], AWS Shield [7]). Der Einsatz kommerzieller Lösungen birgt einige Probleme, etwa mitunter erhebliche Kosten oder das Problem des notwendigen Vertrauens, welches dem Betreiber einer DoS-Abwehr entgegengebracht werden muss. Folglich ist eine effiziente Abwehr von DoS-Angriffen mit eigens errichteten und gewarteten Mechanismen ein verfolgenswertes Ziel - insbesondere wenn sich dadurch mehrere Systeme zugleich schützen lassen.
Ziel des Softwareprojekts ist es, ein System zwischen Internet-Uplink und internem Netzwerk zu schaffen, das bei einer hohen Bandbreite und im Dauerbetrieb effektiv (D)DoS Angriffe abwehren kann, während Nutzer weiterhin ohne Einschränkungen auf ihre Dienste zugreifen können. Die entstehende Anwendung implementiert einen (D)DoS-Traffic-Analysator und einen intelligenten Regelgenerator, wodurch interne Netzwerke vor externen Bedrohungen, die zu einer Überlastung des Systems führen würden, geschützt sind. Es enthält Algorithmen zur Verkehrsanalyse, die bösartigen Verkehr erkennen und ausfiltern können, ohne die Benutzererfahrung zu beeinträchtigen und ohne zu Ausfallzeiten zu führen.

[1] Denial of Service, dt.: Verweigerung des Dienstes, Nichtverfügbarkeit des Dienstes
[2] Distributed Denial of Service

Kapitel 2

Anforderungen

Im folgenden Kapitel werden die funktionalen und die nicht-funktionalen Anforderungen an das zu entwickelnde System beschrieben. Diese Anforderungen grenzen den Projektumfang eindeutig ein und legen fest, welche Eigenschaften das zu entwickelnde System haben soll. Dafür wird die **MuSCoW**-Methode verwendet, welche einführend kurz erläutert wird.

2.1 Priorisierung der Anforderungen

Um Anforderungen zu strukturieren und nach Wichtigkeit zu priorisieren, wird in der Regel ein System zur Klassifizierung der Eigenschaften verwendet. Hier wurde eine Priorisierung nach der **MuSCoW**-Methode vorgenommen:

Must: Diese Anforderungen sind unbedingt erforderlich und nicht verhandelbar. Sie sind erfolgskritisch für das Projekt.

Should: Diese Anforderungen sollten umgesetzt werden, wenn alle Must-Anforderungen trotzdem erfüllt werden können.

Could: Diese Anforderungen können umgesetzt werden, wenn die Must- und Should-Anforderungen nicht beeinträchtigt werden. Sie haben geringe Relevanz und sind eher ein „Nice to have“.

Won't: Diese Anforderungen werden im Projekt nicht explizit umgesetzt, werden aber eventuell für die Zukunft vorgemerkt.

2.2 Funktionale Anforderungen

Funktionale Anforderungen legen konkret fest, was das System können soll. Hier wird unter anderem beschrieben, welche Funktionen das System bieten soll. Die folgende Tabelle zeigt diese funktionalen Anforderungen.

ID	Name	Beschreibung	MuSCoW
F01	Lokale Administration	Das System muss lokal per Command-Line-Interface administriert werden können.	Must

ID	Name	Beschreibung	MuSCoW
F02	Angriffsarten	Das System muss die Folgen der aufgelisteten (D)DoS-Angriffe abmildern können: • SYN-Flood • SYN-FIN Attack • SYN-FIN-ACK Attack • TCP-Small-Window Attack • TCP-Zero-Window Attack • UDP-Flood Dabei ist vorausgesetzt, dass das Ziel eines Angriffes eine einzelne Station in einem Netzwerk ist und kein Netzwerk von Stationen. Es sind also direkte Angriffe auf einzelne Server, Router, PC, etc. gemeint.	Must
F03	Keine zusätzliche Angriffsfläche	Besonders darf das System den unter „Angriffsarten" spezifizierten Angriffen keine zusätzliche Angriffsfläche bieten, d.h. es darf es auch nicht durch Kenntnis der Implementierungsdetails möglich sein, das System mit diesen Angriffen zu umgehen.	Must
F04	L3/ L4 Protokolle	Das System muss mit gängigen L3/ L4 Protokollen klarkommen.	Must
F05	Modi	Passend zum festgestellten Angriffsmuster muss das System eine passende Abwehrstrategie auswählen und ausführen.	Must
F06	Position	Das System soll zwischen dem Internet-Uplink und dem zu schützenden System oder einer Menge von Systemen platziert werden.	Must
F07	Weiterleiten von Paketen	Das System muss legitime Pakete vom externen Netz zum Zielsystem weiterleiten können.	Must
F08	Installation und Deinstallation	Das System muss durch Befehle in der Kommandozeile zu installieren und zu deinstallieren sein. Hilfsmittel hierzu sind: Installationsanleitung, Installationsskript, Meson und Ninja.	Must
F09	Mehrere Angriffe nacheinander und zeitgleich	Das System muss mehreren Angriffen nacheinander und zeitgleich standhalten, hierbei muss berücksichtigt werden, dass auch verschiedene Angriffsarten und Muster zur gleichen Zeit erkannt und abgewehrt werden müssen.	Must
F10	IPv4	Das System muss mit IPv4-Verkehr zurechtkommen.	Must
F11	Hardware	Das System soll nicht Geräte- bzw. Rechnerspezifisch sein.	Should
F12	Zugriff	Der Zugriff auf das lokale System soll per SSH oder Ähnlichem erfolgen, um eine Konfiguration ohne Monitor zu ermöglichen.	Should
F13	Betrieb	Das System soll auf Dauerbetrieb ohne Neustart ausgelegt sein.	Should

ID	Name	Beschreibung	MuSCoW
F14	Privacy	Das System soll keine Informationen aus der Nutzlast der ihm übergebenen Pakete lesen oder verändern.	Should
F15	Konfiguration	Der Administrator soll die Konfiguration mittels Konfigurationsdateien ändern können.	Can
F16	Abrufen der Statistik	Der Administrator soll Statistiken über das Verhalten des Systems abrufen können.	Can
F17	Starten und Stoppen des Systems	Der Administrator soll das System starten und stoppen können.	Can
F18	Informieren des Anwenders	Der Anwender soll über Angriffe informiert werden.	Can
F19	Administration über graphische Oberfläche	Das System soll über eine graphische Oberfläche administriert werden können.	Can
F20	IPv6	Das System soll mit IPv6-Verkehr zurechtkommen können	Can
F21	Weitere Angriffsarten	Das schützt weder vor anderen außer den genannten DoS-Angriffen (siehe F02 „Angriffsarten“)-insbesondere nicht vor denjenigen, welche auf Anwendungsebene agieren-, noch vor anderen Arten von Cyber-Attacken, die nicht mit DoS in Verbindung stehen. So bleibt ein Intrusion Detection System weiterhin unerlässlich.	Won’t
F22	Anzahl der zu schützenden Systeme	Das System wird nicht mehr als einen Server, Router, PC, etc. vor Angriffen schützen.	Won’t
F23	Fehler des Benutzers	Das System soll nicht vor Fehlern geschützt sein, da es durch eine nutzungsberechtigte Person am System ausgeführt wird. So sollen beispielsweise Gefährdungen, welche aus fahrlässigem Umgang des Administrators mit sicherheitsrelevanten Softwareupdates resultieren, durch das zu entwickelnde System nicht abgewehrt werden.	Won’t
F24	Softwareupdates	Das System soll keine Softwareupdates erhalten und soll nicht gewartet werden.	Won’t
F25	Router-/Firewall-Ersatz	Das System soll nicht als Router oder als Firewall-Ersatz verwendet werden.	Won’t
F26	Hardware-Ausfälle	Das System soll keine Hardwareausfälle (zum Beispiel auf den Links) beheben.	Won’t
F27	Fehler in Fremdsoftware	Das System kann nicht den Schutz des Servers bei Fehlern in Fremdsoftware garantieren.	Won’t

2.3 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen gehen über die funktionalen Anforderungen hinaus und beschreiben, wie gut das System eine Funktion erfüllt. Hier sind zum Beispiel Messgrößen enthalten, die das System einhalten soll. Im folgenden werden diese nichtfunktionalen Anforderungen beschrie-

ben.

ID	Name	Beschreibung	MuSCoW
NF01	Betriebssystem	Die entwickelte Software muss auf einer Ubuntu 20.04 LTS Installation laufen. DPDK muss in Version 20.11.1 vorliegen und alle Abhängigkeiten erfüllt sein.	Must
NF02	Verfügbarkeit	Die Verfügbarkeit des Systems soll bei mindestens 98% liegen. Verfügbarkeit heißt hier, dass das System in der Lage ist, auf legitime Verbindungsanfragen innerhalb von 10 ms zu reagieren.	Must
NF03	Datenrate	Die anvisierte Datenrate, welche vom externen Netz durch das zu entwickelnde System fließt, muss bei mindestens 20 Gbit/s liegen.	Must
NF04	Paketrate	Die anvisierte Paketrate, welche vom zu entwickelnden System verarbeitet werden muss, muss bei mindestens 30 Mpps liegen.	Must
NF05	Transparenz	Der Anwender soll das Gefühl haben, dass die Middlebox nicht vorhanden ist.	Should
NF06	Abwehrrate SYN-Flood	Die für die Angriffe anvisierten Abwehrraten sind für die SYN-Flood, SYN-FIN und SYN-FIN-ACK jeweils 100%.	Should
NF07	False Positive	Der maximale Anteil an fälschlicherweise nicht herausgefiltertem und nicht verworfenem illegitimen Traffic, bezogen auf das Aufkommen an legitimem Traffic, soll 10% im Angriffsfall und 5% im Nicht-Angriffsfall nicht überschreiten.	Should
NF08	False Negative	Der maximale Anteil an fälschlicherweise nicht verworfenem bösartigem Traffic, bezogen auf das Gesamtaufkommen an bösartigem Traffic, soll 5% nicht überschreiten.	Should
NF09	Round Trip Time	Die Software soll die Round-Trip-Time eines Pakets um nicht mehr als 10 ms erhöhen.	Should

Kapitel 3

Anforderungsanalyse

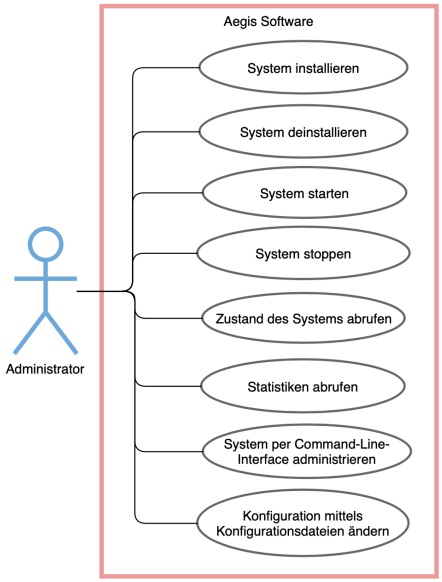

Abbildung 3.1: UML-konformes Use-Case-Diagramm

Es wurde sich dafür entschieden, auch Akteure in das unten stehende Use-Case-Diagramm (siehe Abb. 3.2) aufzunehmen, die nach den UML-Standards nicht in einem Use-Case-Diagramm (siehe Abbi. 3.1) vorkommen.
Dadurch können auch Use-Cases modellieren werden, die über das Installieren und die grundlegenden administrativen Aufgaben hinausgehen.
So agiert beispielsweise der Angreifer in Abb. 3.1 nicht mit dem System. Jedoch ist er für das Verständnis, welche Schutzmechanismen ergriffen werden müssen, unerlässlich. Deshalb ist er in Abb. 3.2 hinzugekommen.
Der normale Nutzer soll möglichst wenig bis gar nichts vom Abwehrsystem mitbekommen, weswegen er nur mit wenigen Anwendungsfällen in Beziehung steht.
Somit ist dieses Anwendungsfalldiagramm ein Beispiel dafür, dass die Diagramme in diesem Projekt eine Sonderrolle einnehmen.

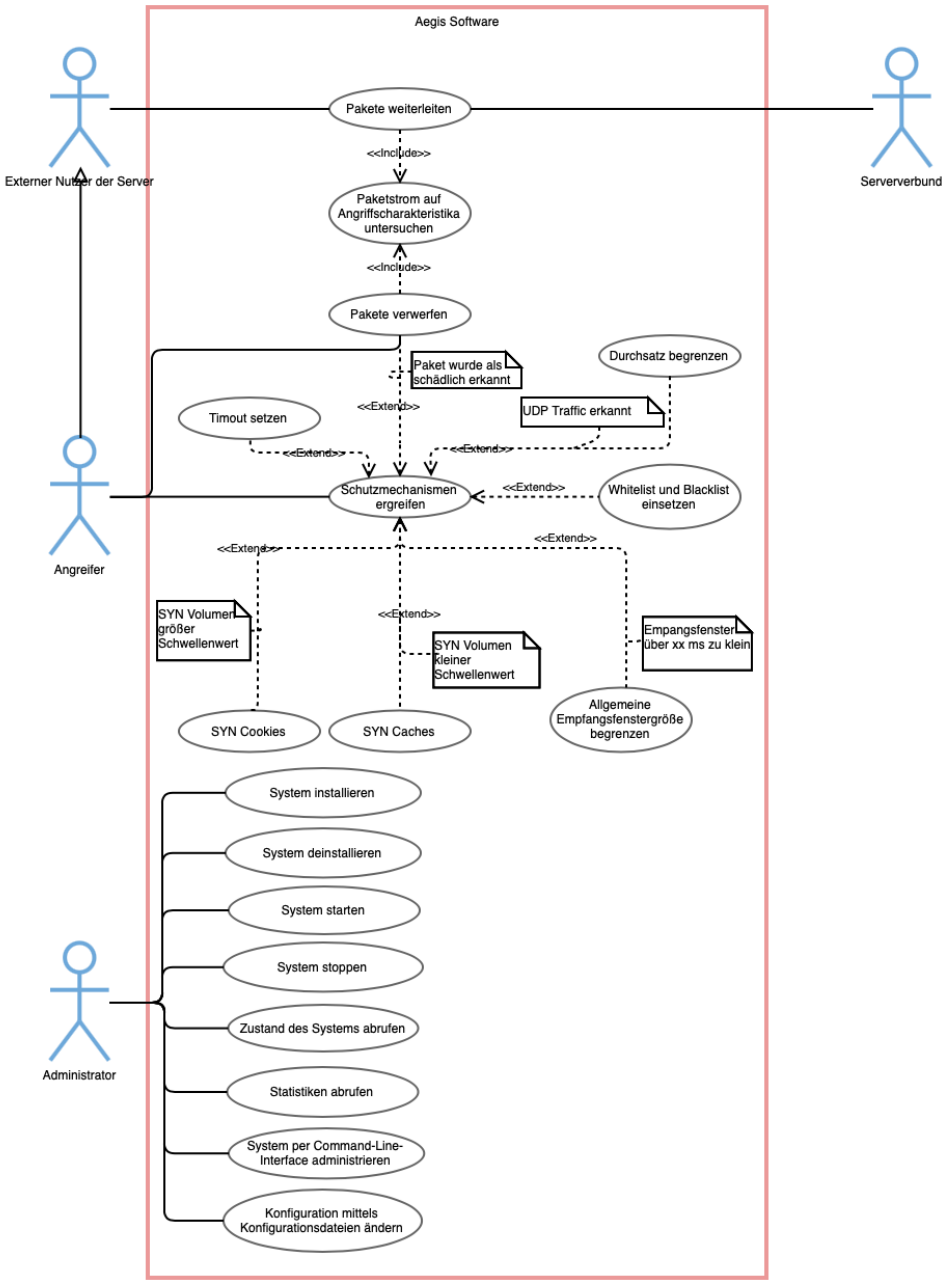

Abbildung 3.2: nicht UML-konformes Use-Case-Diagramm

Kapitel 4

Aufwandsschätzung

Zur Schätzung des Aufwands in einem IT-Projekt liegen verschiedene Verfahren vor. Methoden, wie beispielsweise das Analogieverfahren, bei welchem der Aufwand durch Analogieschlüsse aus anderen, bereits abgeschlossenen Entwicklungsprojekten geschätzt wird, sind für das vorliegende Projekt jedoch wenig geeignet. Denn für diese Art der Aufwandsschätzung müssen Aufwandsdaten bereits abgeschlossener Entwicklungsprojekte vorliegen. Diese Projekte sollen zudem in Punkten wie inhaltliches Ziel, Produktumfang, personelle und zeitliche Ressourcen oder Vorgehensweise sehr ähnlich sein. Solche Daten liegen zu diesem Projekt nicht vor.
Auch das Function Point Verfahren ist hier nicht anwendbar, da bei diesem nur mithilfe einer auf Erfahrung beruhenden Tabelle Functions Points in Aufwand umgerechnet werden können. Eine solche Tabelle liegt hier ebenfalls nicht vor. Zudem lassen sich viele der Anforderungen nicht eindeutig in die fünf Kategorien „Eingabedaten“, „Ausgabedaten“, „Abfragen“, „Datenbestände“ und „Referenzdateien“ einordnen. Allgemein werden im Function Point Verfahren nicht (oder nur indirekt) die Komplexität der Algorithmen und der Aufwand für unterstützende Aktivitäten wie Projektmanagement, Qualitätsprüfung oder Dokumentation berücksichtigt. Somit ist dieses Verfahren lediglich für betriebliche und kaufmännische Systeme gut geeignet, weniger aber für technische Anwendungen wie hier.
Anmerkung: Nachfolgende Aufwandsschätzungen werden während des Projektverlaufs weiter aktualisiert (zum Beispiel bei Änderungen von Anforderungen und/oder Meilensteinen).

4.1 Parkinson’s law

Eine sehr bekannte, jedoch ironisierende Methode zur Aufwandsschätzung ist das Parkinsonsche Gesetz (engl.: „Parkinson’s law“). Dieses lautet wie folgt: „Work expands so as to fill the time available for its completion“. Laut diesem Gesetz nimmt somit die Dauer der Arbeit genau die Zeit an, die ihr zur Verfügung steht.
Da für die Bearbeitung des Softwareprojekts drei Monate zur Verfügung stehen und das Projektteam aus acht Personen besteht, beträgt der Aufwand nach diesem Gesetz 24 Personenmonate.

Interpretation des Ergebnisses:

Das Ergebnis dieser Schätzung sollte nicht zur Planung und Kontrolle des Projektfortschritts verwendet werden, da es zum Aufschieben von Arbeit veranlasst und so die Effizienz reduziert.

Dieses Gesetz spiegelt vielmehr den britischen Humor wieder, als es einen sinnvollen Beitrag zur Projektplanung leistet.
Um dem Aufschieben von Aufgaben entgegenzuwirken, zeigt jedoch das Parkinsonsche Gesetz indirekt eine einfache Lösung auf: Es müssen knappe Deadlines gesetzt werden. Das erhöht die Disziplin, die Aktivität und die Produktivität im Team. Der Goal-Gradient-Effekt bestärkt dies zusätzlich. So besagt dieses Gesetz, dass der Aufwand, den man in eine Sache investiert, umgekehrt proportional zur verbleibenden Zeit steigt.

4.2 COCOMO II

Bei COCOMO II (COnstrucitve COst MOdel), welches bereits 1981 durch den Softwareingenieur Barry W. Boehm entwickelt wurde, handelt sich es um ein algorithmisches Modell zur Aufwandsschätzung von Software. In diesem Modell werden zahlreiche Einflussfaktoren wie Quantität, Qualität oder Produktivität berücksichtigt. Zudem besteht COCOMO II aus drei Teilmodellen, welche sich unter anderem in den Skalenfaktoren oder den Modellkonstanten unterscheiden. Im Folgenden wird sich auf *„The Early Design Model“* (Die frühe Entwicklungsstufe) bezogen, da diese Stufe stark zum derzeitigen Projektstand passt. Auf dieser Stufe liegen schon sowohl die Anforderungen als auch ein erster Grobentwurf vor.
Das Modell baut im Wesentlichen auf folgender Formel auf:

$$PM = A \cdot \text{Größe}^{E} \cdot M \tag{4.1}$$

Für den Koeffizienten A wird der Erfahrungswert 2,5 angenommen.
Die Größe wird in KLSLOC (kilo source lines of code) angegeben. Sie beträgt hier 3,6. Als Referenzprojekt dient POSEIDON [8]. In diesem Projekt wurde eine DDoS-Abwehr durch ungefähr 3600 C/C++-Codezeilen implementiert. Im Unterschied zu der hier zu entwickelnden Software wurde jedoch ein komplettes Endprodukt entwickelt und kein Prototyp, wie es in dem vorliegenden Projekt der Fall ist.
Um den steigenden Aufwand bei wachsender Projektgröße zu berücksichtigen, wird der Exponent E, dessen Wert zwischen 1,01 und 1,26 liegt, verwendet. Durch die Bewertung unterschiedlicher Skalierungsfaktoren wird E berechnet.

Faktor	**Punkte**	**Bemerkungen**
Neuartigkeit	2	Bei einzelnen Teammitgliedern liegt noch geringe Erfahrungmit dieser Art von Projekten vor. Dies ist oftmals begründet durch das junge Alter. Jedoch gibt es online schon Lösungen zu ähnlichen Problemen vor, an denen man sich orientieren kann.
Entwicklungsflexibilität	2	Zwar liegen einige Vorgaben zum Ablauf des SW-Projektes vor (z.B. die Einteilung in drei Hauptphasen), jedoch erlaubt das Vorgehensmodell Unified Process noch eine gewisse Flexibilität im Entwicklungsprozess.
Architektur/ Risikoauflösung	3	Die Risiken wurden rechtzeitig identifiziert und Maßnahmen überlegt. Jedoch könnten aufgrund der geringen Erfahrung noch Risiken unendeckt geblieben sein.
Teamzusammenhalt	1	Die Vertrautheit und Zusammenarbeit im Team ist optimal.
Ausgereiftheit des Prozesses	5	Der Prozess ist noch wenig ausgereift.

E lässt sich nun berechnen, indem man auf den fixen Wert 1,01 ein Hundertstel von der Summe der Punkte addiert.

$$E = 1,01 + \frac{2+2+3+1+5}{100} = 1,01 + 0,13 = 1,14$$

M ergibt sich durch die Multiplikation folgender Projekt- und Prozessfaktoren:

- RCPX: Product Reliability and Complexity
- RUSE: Developed for Resuability
- PDIF: Platform Difficulty
- PERS: Personnel Capability
- PREX: Personnel Experience
- FCIL: Facilities
- SCED: Required Developement Schedule

Mithilfe der unteren Skala werden die einzelnen Projekt- und Prozessfaktoren bewertet. In der Farbe Rot sind diejenigen Faktoren gekennzeichnet, die auf das vorliegende Projekt am besten zutreffen.

	- - -	- -	-	~	+	++	+++
RCPX	0,49	0,60	0,83	1	1,33	1,91	2,72
RUSE			0,95	1	1,07	1,15	1,24
PDIF			0,87	1	1,29	1,81	2,61
PERS	2,12	1,62	1,26	1	0,83	0,63	0,50
PREX	1,59	1,33	1,22	1	0,87	0,74	0,63
FCIL	1,43	1,30	1,10	1	0,87	0,73	0,62
SCED		1,43	1,14	1	1	1	n/a

Nun lässt sich der Multiplikator M berechnen:

$$M = PERS \cdot RCPX \cdot RUSE \cdot PDIF \cdot PREX \cdot FCIL \cdot SCED = 1,33 \cdot 0,95 \cdot 0,87 \cdot 1 \cdot 1,33 \cdot 1 \cdot 1 \approx 1,22$$

Jetzt sind alle Werte gegeben, um mithilfe der Formel 4.1 den Aufwand in Personenmonate zu schätzen.

$$PM = A \cdot \text{Größe}^{E} \cdot M = 2,5 \cdot 3,6^{1,14} \cdot 1,22 \approx 13,17$$

Nun kann der Aufwand in Personenmonat in Personenstunden umgerechnet werden die Folgende:

$$PS = 13,17 \cdot 160h = 2107h$$

Das Team besteht aus zwei Wirtschaftsinformatiker, die jede Woche 15h für das Softwareprojekt aufwenden sollen, und sechs Informatiker und Ingenieurinformatiker, deren Wochenstundenanzahl bei 20h liegt. Das Softwareprojekt soll innerhalb von 12 Wochen beendet werden.
Somit ist die zur Verfügung stehende Zeit:

$$(2 \cdot 15h + 6 \cdot 20h) \cdot 12 = 1800h$$

Interpretation der Ergebnisse:

Der Wert von 2107h liegt ca. 300h über dem angestrebten Wert von 1800h. Es kann unterschiedliche Gründe für diese Abweichung geben.
Ein Grund dafür könnte sein, dass im zu entwickelnden System kein schlüsselfertiges Produkt entwickeln wird, sondern vielmehr ein Prototyp. Somit könnte der Schätzwert von 3600 Codezeilen zu hoch gegriffen sein.
Durch die Komplexität des Projektes kann es zudem vorkommen, dass die 15 beziehungsweise 20 Wochenstunden nicht immer ausreichend sind und somit mehr Arbeitszeit aufgewendet werden muss.
Zudem muss keine Zeit zur Berücksichtigung von Support vorgesehen werden, da es nach den 12 Wochen als abgeschlossen angesehen werden kann.
Bereits kleine Änderungen an den Projekt- und Prozessfaktoren haben große Auswirkungen. Falls bereits einer dieser oben aufgeführten Faktoren ungenau geschätzt wurde, kann das Ergebnis verfälscht sein.
Abschließend kann noch angemerkt werden, dass es sich um eine Schätzung handelt. Schätzungen sind (fast) immer mit Ungenauigkeiten verbunden.

Kapitel 5

Risikoanalyse

5.1 Risikoidentifikation

Durch Befragung des gesamten Projektteams wurden sowohl fachliche, kaufmännische und planerische Risiken identifiziert. Indem Teammitglieder mit unterschiedlichen Erfahrungen und Fachkompetenzen miteinbezogen wurden, konnten Risiken aus verschiedenen Blickwinkeln identifiziert werden. In der folgenden Tabelle werden sowohl die verschiedenen Risiken als auch deren Eintrittswahrscheinlichkeit[1], Auswirkung und Maßnahmen kurz beschrieben.
Die dort beschriebenen Maßnahmen verfolgen das Ziel, die Eintrittswahrscheinlichkeit zu verringern und bei Eintritt die Auswirkungen abzuschwächen.
Hinweis: Die unten stehenden Tabelle wird im Laufe des Projektes noch weiter ergänzt, da die Risikoanalyse keine einmalige, sondern fortlaufende Aktivität im Projekt darstellt. So können beispielsweise neue Risiken auftauchen, die zu Beginn des Projektes noch nicht vorhanden waren oder übersehen wurden. Auch Eintrittswahrscheinlichkeiten können sich noch ändern. Außerdem ist es möglich, dass Auswirkungen und Maßnahmen hinzukommen oder verschwinden.

ID	Risiko	Eintrittswahrscheinlichkeit	Auswirkung	Maßnahmen
R01	Software wird unzureichend dokumentiert.	30%	Sicherheitslücken, falsch aufgefasste Anforderungen, Softwareanomalien, schwierige Wartung, Software kann nicht richtig getestet werden u. v. m.	Motivation der Teammitglieder dazu, dass diese gewissenhaft Dokumentation führen; Erklären der Wichtigkeit der Dokumentation; Einführen von Konventionen zur Dokumentation; Verwendung automatischer Dokumentationswerkzeuge

[1] Wert zwischen 0% und 100%. 0% entspricht dem unmöglichen Ereignis, 100% dem sicheren Ereignis. Ereignisse mit Werten nahe 0% sind unwahrscheinlich, Ereignisse mit Werten nahe 100% wahrscheinlich.

ID	Risiko	Eintrittswahrscheinlichkeit	Auswirkung	Maßnahmen
R02	Unzureichende Erfahrung des Projektteams bezüglich neuer Tools (z. B. DPDK, Ninja, Meson) und der Programmiersprache C++	80%	Zeitverzögerung (v. a. längere Dauer der Implementierung durch umfassende Einarbeitungsphase)	Gute Einarbeitung; Erstellen und Halten von Präsentationen zu schwierigen Themen; Gegenseitige Hilfe; Festlegen von Ansprechpartner für die einzelnen Themenbereiche
R03	Weniger Austausch und weniger effiziente Zusammenarbeit durch Online-Lehre	70%	Probleme werden später oder nicht sichtbar; schwieriger Überblick über den Arbeitsstand bei anderen Teammitgliedern; Statusmeldungen fehlen; Geringes Teamgefühl	regelmäßige Treffen ohne Zeitdruck; möglichst organisierte Kommunikation (z.B. über Zulip oder Webex)
R04	Hardware-Probleme (z. B. Ausfall oder andere Defekte)	20%	Zeitverzögerung, finanzielle Kosten	sorgfältiger Umgang mit der Hardware
R05	Testbed nicht optimal konfigurierbar (aufgrund der verringerten Geräteanzahl und beschränkter Optionen ist nicht jede vorteilhafte Konstellation möglich)	60%	erschwerte Implementierung, Zeitverzögerung, Nichterfüllen einzelner Anforderungen, finanzielle Kosten beim Kauf zusätzlicher Hardware	möglichst effiziente Nutzung der vorhandenen Hardware; Kauf zusätzlicher Hardware; Entwicklung eines Netzwerkplans

5.2 Risikomatrix

Die folgende Risikomatrix (auch Risikoportfolio, Risikodiagramm oder Risiko-Map, siehe Abbildung 5.1) visualisiert die Projektrisiken R01 bis R06, indem sie die Eintrittswahrscheinlichkeiten und die dazugehörigen Schadensausmaße ins Verhältnis setzt. Je weiter man sich im Diagramm nach rechts bewegt, desto höher ist die Eintrittswahrscheinlichkeit. Während diese Wahrscheinlichkeit ganz links bei 0% liegt, beträgt sie am rechten Rand 100%. Je weiter oben sich das Risiko im Diagramm befindet, desto größer ist das Schadensausmaß.
Bei den Risiken, die sich im grünen Bereich befinden, sind keine zusätzlichen Maßnahmen zur Risikominimierung notwendig, wohingegen die Risiken im gelben Bereich so weit wie möglich

abgemindert werden sollen. Die gefährlichsten Risiken befinden sich im roten Bereich. Für diese Risiken müssen geeignete Präventionsmaßnahmen getroffen werden, um sie in den gelben Bereich zu bewegen.
Ebenso kann man aus der Position der Risiken ableiten, wie dringend Präventionsmaßnahmen eingeführt werden müssen.

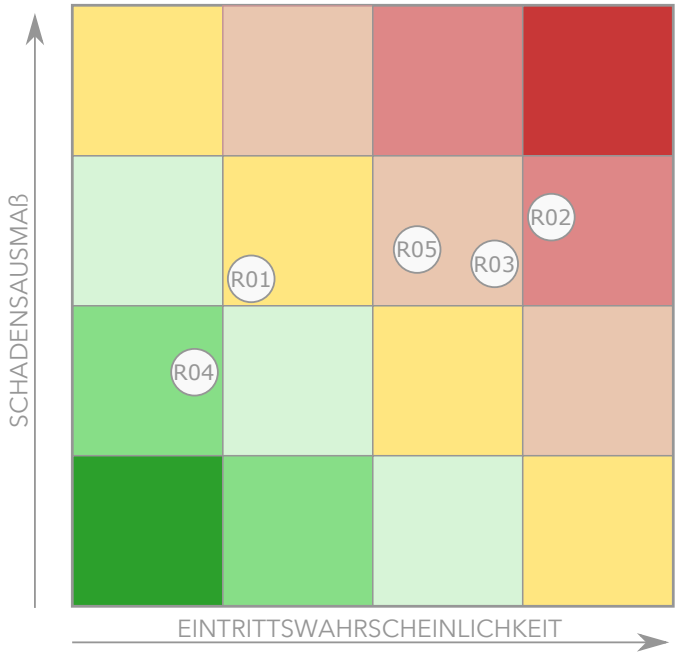

Abbildung 5.1: Risikomatrix mit den Risiken R01 bis R06

Aus der Grafik lässt sich erkennen, dass vor allem für die Risiken R02, R03 und R05 geeignete Maßnahmen zur Minimierung der Risiken eingeführt werden müssen.

5.3 Verbindung zum Vorgehensmodell

Indem sich für den Unified Process als Vorgehensmodell entschieden wurde, werden in dem Projekt Risiken schon frühzeitig adressiert. Zudem werden zu Beginn jeder Phase jeweils die Punkte mit den größten Risiken zuerst bearbeitet.

Kapitel 6

Vorgehen

6.1 Vorgehensmodell und die Anpassung des Vorgehens

Ein Vorgehensmodell legt einen bestimmten Ansatz für die Art der Durchführung und die Reihenfolge der Teilaufgaben der Systementwicklung maßgeblich fest. Erst mit einem Vorgehensmodell wird ein komplexer Softwareentwicklungsprozess übersichtlich, plan- und strukturierbar. Die Wahl des Vorgehensmodells ist deshalb von enormer Bedeutung.
Im **Wasserfallmodell** sind zwar Planung, Kontrolle und Steuerung vergleichsweise einfach, jedoch erfordern Rücksprünge einen hohen Änderungsaufwand in allen Dokumenten. Somit zeigt sich dieses sequentielle Modell unflexibel gegenüber Projekten, bei denen sich die Anforderungen an das zu entwickelnde Projekt oft verändern. Dies ist jedoch hier, da durch die Komplexität des Themas eventuell erst einige Anforderungen später identifiziert werden.
Es wurde gegen ein **agiles Vorgehen** gestimmt, da unter anderem das Projektmanagement in diesem Vorgehensmodell durch das chaotische Vorgehen sehr schwierig ist. Außerdem hat die Planbarkeit des Ergebnisse hohe Priorität.
Letztlich wurde sich für den **Unified Process** (siehe Abb. 6.1) entschieden. Dieses Vorgehensmodell nutzt die UML als Notationssprache.

Grundsätzlich besteht dieses iterative und inkrementelle Vorgehensmodell aus vier Phasen. Zu Beginn des Projekts werden in der **Konzeptionsphase** (inception phase) die zentralen Anforderungen ermittelt, der Projektumfang definiert und möglichst viele Projektrisiken entdeckt. Diese Phase stellt die Kürzeste dar. Sie endet mit dem Milestone lifecycle objective.
Auf die Konzeptionsphase folgt die **Ausarbeitungsphase** (elaboration phase), in der unter anderem die Systemanforderungen vervollständigt und die Entwurfsspezifikation entwickelt werden. Hier wird auch ein erster Protoptyp erarbeitet. In diesem Projekt umfasst diese Phase sowohl den Entwurf, als auch große Teile der Planung. Diese Phase besteht aus drei Iteration, wobei die Iterationen unterschiedlich lange sind. In jeder Iteration wird das vorherige Ergebnis verfeinert.
In der **Konstruktionsphase** (construction phase) findet ein Großteil der Implementierung, aber auch des Testens statt.
Die letzte Phase ist die **Inbetriebnahme** (transition phase). Die Auslieferung der Software an den Kunden wird im vorliegenden Projekt sehr kurz ausfallen. Jedoch wird in dieser Phase zusätzlich verstärkt getestet.
In den Phasen laufen verschiedene Kernprozesse zu unterschiedlichen Anteilen parallel ab. Man kann die einzelnen Phasen wie folgt ins Deutsche übersetzen: Business Modelling $\widehat{=}$ Geschäftspro-

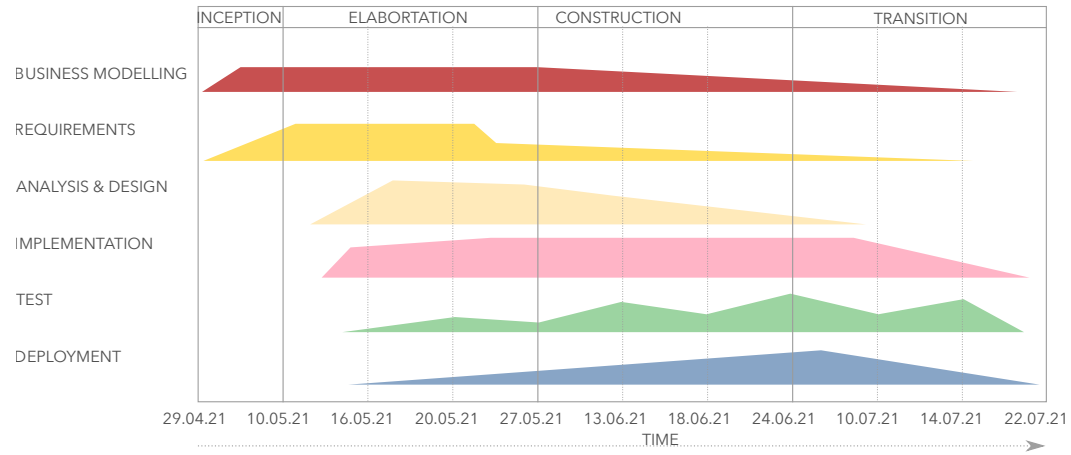

Abbildung 6.1: Angepasstes Vorgehensmodell (Unified Process)

zessmodellierung, Requirements ≙Anforderungsanalyse, Analysis and Design ≙ Analyse und Design, Implementation ≙ Implementierung, Test ≙ Test, Developmenent ≙ Auslieferung.
Aus dem agilen Vorgehen wurden für das Projekt einige Prinzipien übernommen: Zum einen die **häufigen Meetings** und **kurzen Statusmeldungen**, damit jedes Teammitglied genau weiß, wo die anderen Mitglieder stehen und was deren derzeitigen Probleme sind.
Zudem ist durch Gitlab ein **Kanbanboard** (siehe Abb. 6.2) verfügbar. Mit diesem agilen Projektmanagement-Tool werden die Aufgaben visualisiert. Es hilft dem Projektteam, die Arbeit zu strukturieren und so deren Effizienz zu steigern.
Auch wurde sich innerhalb des Projektteams auf (agile) **Werte** geeinigt, um eine bestmögliche Zusammenarbeit zu gewährleisten (siehe Abb. 6.3).

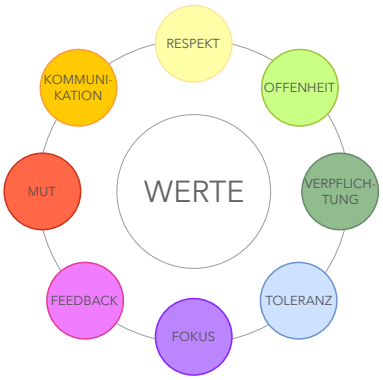

Abbildung 6.3: Werte

Jedes Teammitglied zeigt **Respekt** gegenüber die anderen und schätzt diese. Außerdem zeigen es jedem Verständnis, wenn beispielsweise jemand ein Problem hat. Zudem wird auch auf die Schwächeren Rücksicht genommen.
Offenheit bedeutet hier, dass neue Informationen und Erfahrungen bewusst aufgenommen wer-

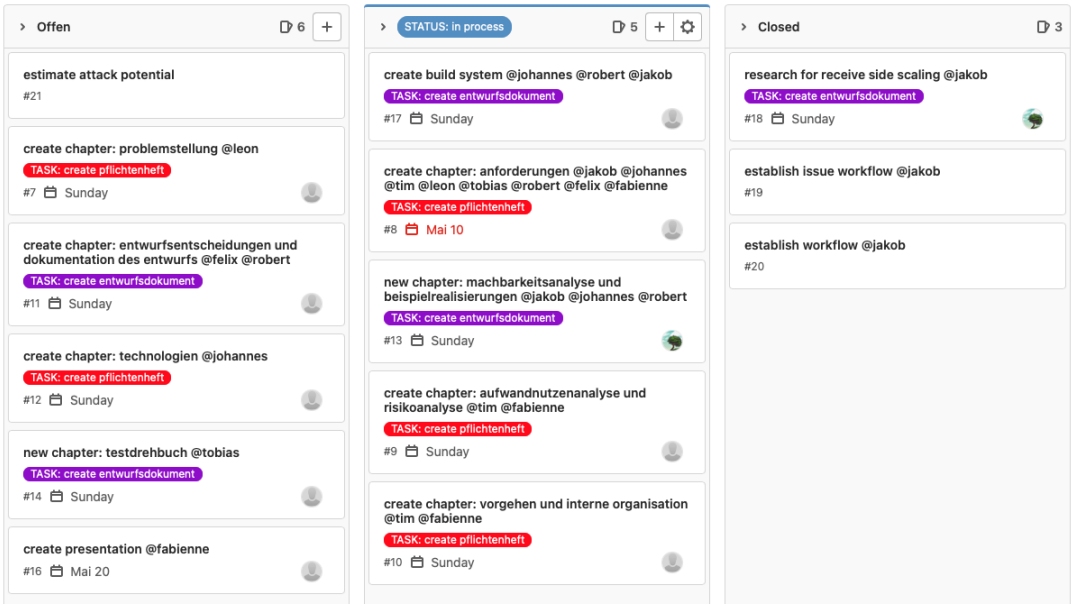

Abbildung 6.2: Board auf Gitlab (Datum des Screenshots: 15.05.2021)

den und nicht vorschnell als unwichtig bewertet werden. Jeder darf seine Meinung frei äußern, um Missverständnissen und Auseinandersetzungen vorzubeugen. Ebenso soll Transparenz herrschen und Problemen sollen sofort auf den Grund gegangen werden.
Der Wert **Verpflichtung** kann so interpretiert werden, dass alle Teammitglieder sich der Projektaufgabe verbunden fühlen sollen.
Außerdem sind **Toleranz** und Vielfalt zentrale Werte. Personen mit anderen Sichtweisen werden nicht etwa als Konkurrenten gesehen, sondern vielmehr als Bereicherung für das Team.
Der **Fokus** liegt im Projekt vor allem auf den Aufgaben und auf den gemeinsamen Zielen. Verschwenden von Zeit und Kapazitäten und Ablenkungen sollen vermieden werden. Zudem hat die Fertigstellung einer bereits begonnen Aufgabe Vorrang gegenüber den Start einer neuen Aufgabe.
Indem in den Meetings regelmäßig der derzeitige Stand aufgezeigt wird, kann sich jedes Teammitglied individuelles **Feedback** von den anderen Beteiligten holen.
Zusätzlich soll jedes Teammitglied **Mut** aufweisen, indem es zum Beispiel neue Aufgaben übernimmt, die vielleicht zuerst als schwer machbar und komplex erscheinen. Außerdem erfordert es Mut zu sagen, wenn man Hilfe benötigt, aber auch mitzuteilen, dass man hinter dem Zeitplan liegt. Auch das Ausprobieren neuer Lösungswege fällt unter den Wert Mut.
Zudem ist die tägliche **Kommunikation** mit den Teammitgliedern für eine gute Zusammenarbeit erforderlich. So sollte jeder beispielsweise mehrmals täglich nach neuen Zulip-Nachrichten schauen und dort auf Fragen antworten. Außerdem wurde gemeinsam beschlossen, dass man bei Änderungen an den Dokumenten andere benachrichtigt.

6.2 Projektplan

Der Projektplan besteht aus verschiedenen Objekten: Dem Projektstrukturplan und einem Ablaufplan (hier Gantt-Diagramm).
Der **Projektstrukturplan**, welcher in Abb. 6.4 zu sehen ist, gliedert und strukturiert das Pro-

jekt hierarchisch. Er ist die Grundlage für die Ablaufplanung.

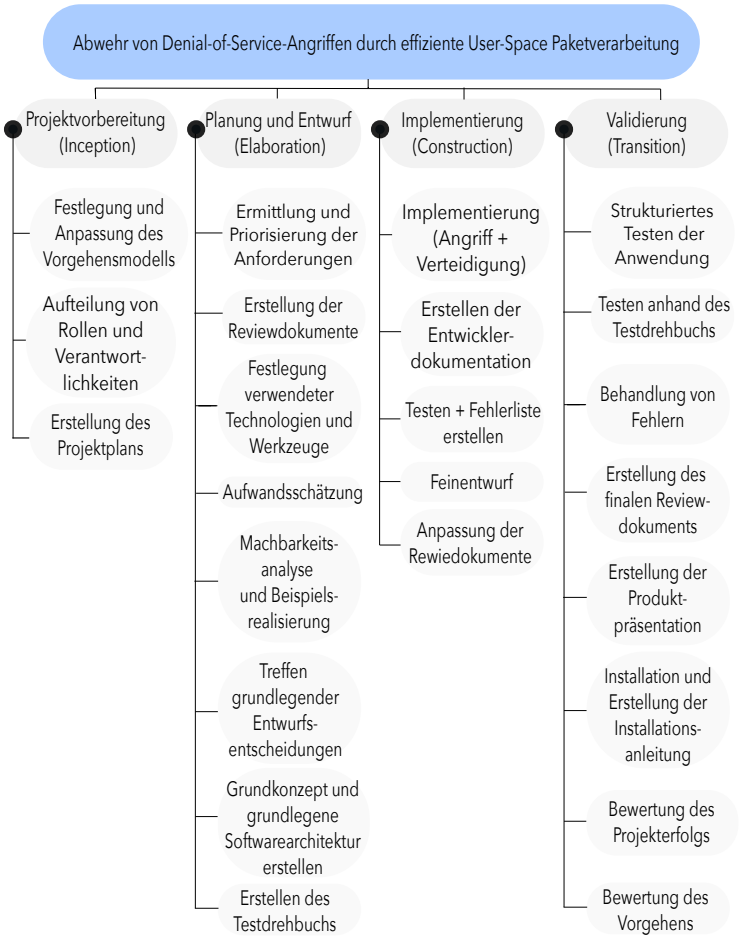

Abbildung 6.4: Projektstrukturplan

Der **Ablaufplan** dient zur zeitlichen Darstellung des Projektablaufes. Die hier gewählte Darstellung ist das Gantt-Diagramm. Mittels Microsoft Project Professional wird so nicht nur das Projekt geplant, gesteuert und überwacht, sondern auch das Gantt-Diagramm erstellt.
In den Meetings wird regelmäßig der aktuelle Stand der einzelnen Teammitglieder abgefragt. Somit wird die Werte des Ganntt-Diagramms kontinuierlich aktualisiert. Die Gantt-Diagramme zu den verschiedenen Phasen sind in den Abbildungen 6.5, 6.6, 6.7 und 6.8 zu finden. Dort sind zudem die Abhängigkeiten zwischen den einzelnen Aufgaben zu sehen (zum Beispiel Anfang-zu-Anfang- oder Ende-zu-Anfang-Beziehungen)

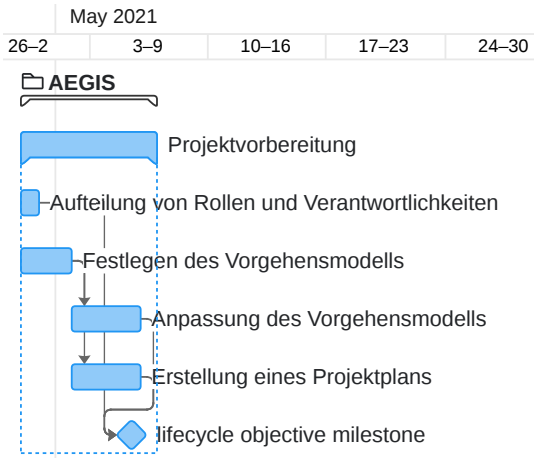

Abbildung 6.5: Ausschnitt des Gantt-Diagramms für die Projektvorbereitungsphase (Stand: 14.05.2021)

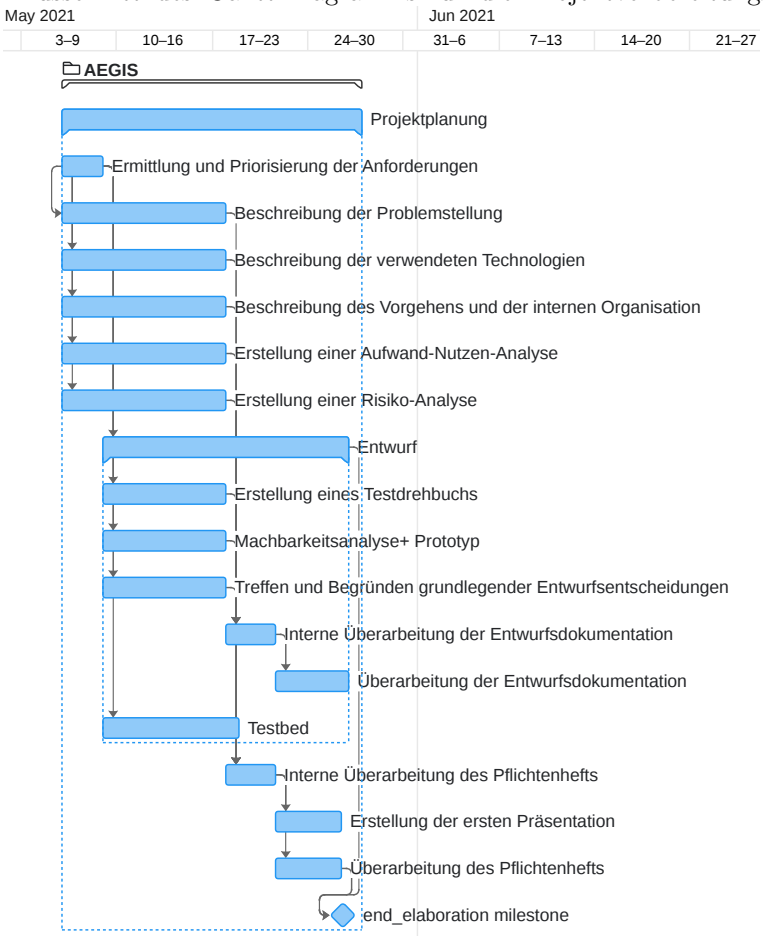

Abbildung 6.6: Ausschnitt des Gantt-Diagramms für die Planungs- und Entwurfsphase (Stand: 14.05.2021)

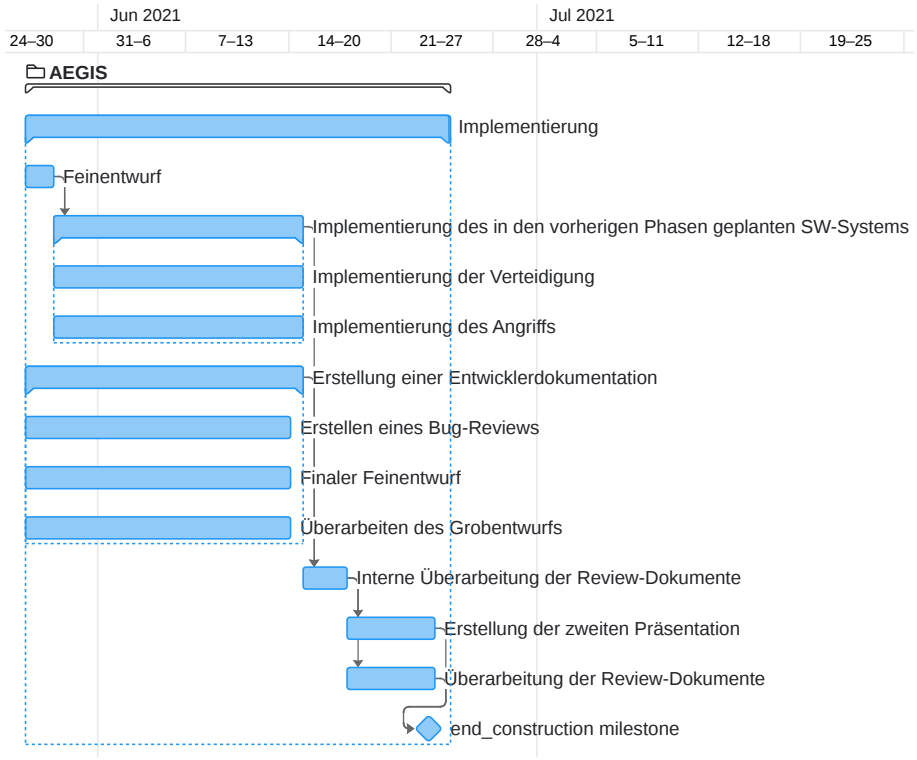

Abbildung 6.7: Ausschnitt des Gantt-Diagramms für die Implementierungsphase (Stand: 14.05.2021)

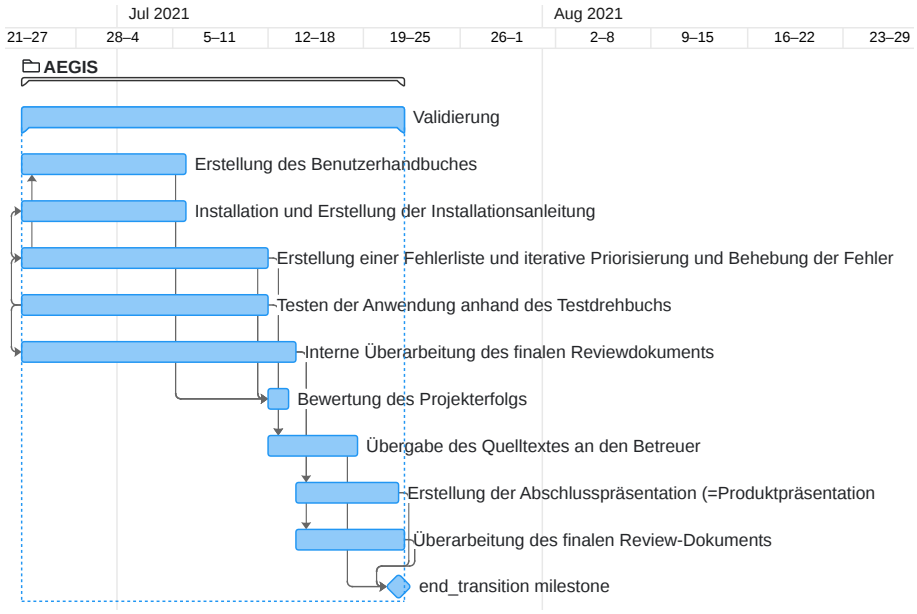

Abbildung 6.8: Ausschnitt des Gantt-Diagramms für die Validierungssphase (Stand: 14.05.2021)

6.3 Meilensteine

An dieser Stelle kommt zunächst die Frage auf, was ein Meilenstein eigentlich genau ist. Dabei handelt es sich um einen besonders wichtigen Punkt im Projektverlauf, an dem etwas überprüft wird. Ein Meilenstein kann ein Ereignis sein, an dem etwas abgeschlossen ist, etwas begonnen wird oder über die weitere Vorgehensweise entschieden wird.
Meilensteine umfassen nie eine Zeitdauer, sondern sind Zeitpunkte und werden meist am Ende von Projektphasen definiert. Es kann allerdings auch innerhalb einzelner Phasen zusätzliche Meilensteine geben.
Am 27.05.2021, am 24.06.2021 und am 21.07.2021 finden Reviews für das Softwareprojekt statt, bei denen die Ergebnisse der letzten Phase mit Unterstützung einer Präsentation durch ein oder zwei Studierende vorgestellt werden. Außerdem müssen bis zu diesen drei Tagen die jeweiligen Review-Dokumente fertiggestellt und eingereicht worden sein. Nach der Präsentation und Verteidigung der Ergebnisse werden diese bewertet. Da es sich dabei um Prüfpunkte handelt, stellt ein erfolgreich absolviertes Review die Erreichung eines Meilensteins dar. Außerdem wurde sich passend zum Unified Process für einen Meilenstein am Ende der ersten Konzeptionsphase entschieden.
Die vier Meilensteine des hier durchgeführten Projekts sind also die folgenden:

1. lifecycle objective
2. end_elaboration
3. end_construction
4. end_transition

Diese sind auch im GitLab erstellt und dein einzelnen Issues zugeordnet. Es bleibt noch anzumerken, dass das Erreichen des letzten Meilensteins gleichzeitig den Projektabschluss darstellt.
Außerdem können den drei Projektphasen Elaboration, Construction und Transmission noch jeweils drei Meilensteine mit Bezug zu den Review-Dokumenten zugeordnet werden. Diese sind:

1. end_first_version
2. end_internal_revision
3. end_last_revision

Damit ist der Abschluss verschiedener Phasen der Erstellung der Review-Dokumente gemeint. Das Ziel lautet bei allen drei Phasen, circa eineinhalb Wochen vor der Abgabe eine erste Fassung erstellt zu haben. Wenn das geschafft wurde, ist der Meilenstein end_first_version erreicht. In den darauffolgenden Tagen wird diese Fassung intern, also durch alle Teammitglieder, überarbeitet und verbessert. Dieses überarbeitete Dokument wird dann eine Woche vor der endgültigen Abgabe bei Martin Backhaus eingereicht und der Meilenstein end_internal_revision ist erreicht. Schließlich wird in der Woche vor der Abgabe Feedback von Martin Backhaus eingearbeitet und letzte Verbesserungen werden vorgenommen. Mit der Abgabe des Dokuments ist auch der letzte Meilenstein end_last_revision erreicht.

Kapitel 7

Interne Organisation

7.1 Rollen

Abhängig von individuellen Fähigkeiten und Interessen der einzelnen Teammitglieder wurden zu Beginn folgende Rollen zugeteilt:

Rolle	**Name**	**Studiengang**
Projektleiterin	Fabienne Göpfert	Wirtschaftsinformatik
Systemarchitekt	Robert Jeutter	Informatik
DPDK-Chief	Jakob Lerch	Ingenieurinformatik
Redakteur	Tim Häußler	Wirtschaftsinformatik
Dokumentation	Leon Leisten	Informatik
Build Engineer	Johannes Lang	Informatik
Qualitätsmanager bzw. Tester	Tobias Scholz	Ingenieurinformatik
Angreifer	Felix Hußlein	Informatik
Entwickler	alle Teammitglieder	-

Im Folgenden werden die Rollen und die dazugehörigen Aufgaben näher beschrieben.
Die **Projektleitung** übernimmt im Projekt die operative organisatorische Leitung. Dabei ist sie nicht nur verantwortlich für die Koordination der Mitglieder, der Organisation des Ablaufs oder der Kontrolle und Bewertung des Projektergebnisses, sondern muss auch „jedem Teammitglied das Gefühl [geben], es habe selbst entschieden“(Daniel Goeudevert). Zudem stellt die Projektleitung die Schnittstelle zur Umgebung des Projekts dar, was sich zum Beispiel in der Kommunikation mit Stakeholdern widerspiegelt. Um diese Rolle bestmöglich erfüllen zu können, ist Kommunikationsfähigkeit, große Zuverlässigkeit und hohes Verantwortungsbewusstsein erforderlich. Auch sind Erfahrung bei der Organisation, Planung und Steuerung hilfreich, die zumindest durch theoretische Veranstaltungen wie „IT-Projektmanagement“ oder „Geschäftsprozessmanagement“ Wirtschaftsinformatiker und -informatikerinnen vorweisen können.
Das Überblicken aller Komponenten und der Zusammenarbeit derer ist die Aufgabe des **Systemarchitekten**. Zudem ist er für die Entwicklung der Architektur des Systems und die Erweiterung vorhandener Architekturen zuständig. Vor allem im Systementwurf und der Systemspezifikation ist diese Rolle stark von Bedeutung. Um all diese Aufgaben zu erfüllen, sind Fähigkeiten wie ein tiefes Verständnis im technischen Bereich, Kenntnis in Methodenentwicklung und Kom-

munikationsfähigkeit erforderlich.
Die Rolle des **DPDK-Chiefs** ist stark projektspezifisch. DPDK steht dabei für Data Plane Development Kit, wobei es sich um ein schnelles Paketverarbeitungsframework auf Benutzerseite speziell für leistungsintensive Anwendungen handelt. Die Rolle des DPDK-Chiefs muss DPDK tiefer verstehen und dieses gewonnene Verständnis auch mit seinen Teammitgliedern kommunizieren. Zudem steht er diesen als ständiger Ansprechpartner zu diesem Thema zur Verfügung.
Zu den Aufgaben eines **Redakteur** gehört das Konzipieren, Erstellen und Überprüfen von Dokumentationen. Solche Dokumente können zum Beispiel die Installationsanleitung, das Pflichtenheft oder die Entwicklerdokumentation sein. Um diese Aufgaben erfüllen zu können, muss der Redakteur im stetigen Austausch mit den Entwicklern stehen.
Zur **Dokumentation**: Diese Rolle ist dafür zuständig, ein Softwaredokumentationswerkzeug auszuwählen und festzulegen, was genau auf welche Weise dokumentiert werden soll. Weiterhin ist er der Ansprechpartner für alle Fragen zur Dokumentation und kümmert sich um die Vollständigkeit der Code-Dokumentation.
Der **Build Engineer** plant das Build System und die Unit Tests. Außerdem ist er für die Versionierung zuständig.
Die Rolle des **Qualitätsmanagers und Testers** hat zahlreiche Aufgaben. Der Tester definiert verschiedene Testfälle. Er leitet diese Testfälle sowohl aus der Spezifikation (funktionale Testverfahren), als auch aus der Struktur des Quellprogramms ab (strukturelle Testverfahren). Daraufhin wählt der Tester verschiedene Testdatenkombinationen aus und definiert das erwartete Softwareverhalten. Nachdem er das Testobjekt mit einer Testdatenkombination ausgeführt hat, vergleicht er das erwartete mit dem tatsächlichen Verhalten. Das Testergebnis wird anschließend dokumentiert. Falls Fehler erkannt wurden, müssen diese an die Entwickler zur Fehlerbehebung weitergegeben werden. Nachdem diese Rolle auch das Qualitätsmanagement umfasst, kommen noch folgende Aufgaben hinzu: die Qualitätsplanung, d.h. die Planung der Anforderungen an ein Produkt (bzw. an das Projekt); die Qualitätsplanung, d.h. das steuernde Eingreifen bei Abweichungen von den Anforderungen; die Qualitätssicherung, d.h. Vertrauen schaffen, dass die Qualitätsanforderungen eingehalten werden; Qualitätsverbesserung, d.h. die kontinuierliche Verbesserung des Ausmaßes der Erfüllung von Anforderungen.
Aufgaben wie das Vorbereiten von Angriffen und das Ausführen von Tests gehören zu den Aufgaben des **Angreifers**. Der Angreifer versucht, das Programm durch gezielte Angriffe zu einem Zustand der Nichtfunktionalität zu bringen. Dabei baut der Angreifer nicht nur eine einzelne Verbindung vom Angriffsrechner zum Server auf, wie es bei DoS-Attacken der Fall ist, sondern er kann auch eine Vielzahl (dezentraler) Quellen für den Angriffstraffic nutzen. Dies erschwert es, den Angriff abzudämmen. Eine solche Attacke, welche in der Praxis von Bot-Netzen ausgeht, nennt man DDoS-Attacke (DDoS steht hierbei für „Distributed Denial of Service“). Zur weiteren Aufgabe des Angreifers zählt es, die Software auf neue Angriffspotentiale zu untersuchen. Außerdem soll er Informationen bereitstellen, um diese Angriffe zu verhindern.

7.2 Kommunikationswege

Das meist genutzte Kommunikationstool ist das Open Source-Tool **Zulip**. Dabei handelt es sich um ein Gruppen-Chatsystem, das vom Fachgebiet Telematik betrieben wird und zur Kommunikation aller Teammitglieder untereinander und mit dem Betreuer Martin Backhaus verwendet wird.
Wesentliche Vorteile von Zulip sind im Folgenden kurz beschrieben: Clients sind für alle gängigen Plattformen verfügbar und man kann sowohl Gruppen- als auch Direktnachrichten versenden. Weiterhin braucht man sich nicht mit fragwürdigen Datenschutzbestimmungen beschäftigen.

Außerdem unterstützt die Software Markdown und Syntaxhervorhebung und einzelne Personen lassen sich mit „@“ markieren, wodurch eine direkte und effiziente Kommunikation möglich wird. Bezüglich der Organisation gibt es mehrere Streams, also themenspezifische Gruppenchats. Es gibt einen Haupt-Stream und einige weniger genutze Streams. Weiterhin wird ein Stream in mehrere Topics (Themen) unterteilt, z. B. für die Agenda eines bestimmten Treffens, konkrete Fragen zum Installieren von DPDK oder zum Git-Workflow. Es ist möglich, sich entweder alle Nachrichten oder nur die Nachrichten einer bestimmten Topic anzeigen zu lassen und somit mehrere Diskussionen gleichzeitig zu führen.
Für die Meetings wird auf **Cisco Webex Meetings** zurückgegriffen. Das ist ein Tool für Videokonferenzen, das von der Universität bereitgestellt wird. Über einen Link, den alle Teammitglieder vor dem Meeting erhalten, kann sich jeder einwählen. Die Software steht für alle gängigen Betriebssysteme zur Verfügung und bietet das Feature, den Bildschirm zu teilen, um Präsentationen zu zeigen oder gemeinsam an einem Dokument zu arbeiten. Bei technischen Schwierigkeiten mit Webex oder für interne Treffen wird auch teilweise auf **Jitsi** zurückgegriffen.
Eine weitere Form der Kommunikation stellt **GitLab** dar. Dabei handelt es ich um ein Open Source-Versionsverwaltungssystem, das für das Projekt benutzt wird. Issues, Labels und Branches sollen aussagekräftig benannt werden. Außerdem besteht für das Projekt ein Wiki, über das verschiedenste Informationen ausgetauscht werden, z.B. Protokolle für jedes Meeting, nützliche Links, der Git-Workflow, verwendete Software u.v.m.
Die gemeinsam beschlossenen Bennenungskonventionen besagen, dass z.B. Branch-, Issue-, Ordner- und Dateinamen klein geschrieben werden. Des weiteren soll nur ASCII und nur die englische Sprache benutzt werden, mit Ausnahme des Wikis, der Präsentationen und der Review-Dokumente.

7.3 Weitere organisatorische Festlegungen

Das komplette Projektteam trifft sich zweimal wöchentlich für circa 90 Minuten mit seinem Betreuer Martin Backhaus. Zu diesen Treffen ist pünktliches Erscheinen erfordert. Diese Treffen werden durch zweimal wöchentliche interne Team-Meetings ergänzt. Der zeitliche Umfang dieser Meetings variiert je nach Bedarf. Zu jedem Meeting wird im Vorhinein eine Agenda erstellt. Regelmäßig wird in diesen Meetings der derzeitige Stand abgefragt, Zudem werden wichtige Entscheidungen protokolliert und in das Gitlab-Wiki übertragen. In diesem Wiki werden außerdem nützliche Links geteilt und Wissen weitergegeben.
Um den Einstieg in das Projekt zu erleichtern, wurde am 06.05.2021 von je zwei Teammitgliedern eine Präsentation halten. Die Themen der Präsentationen umfassten DPDK, Denial-of-Service-Angriffe, effizienten Datenstrukturen, Git und Latex.

Kapitel 8

Dokumentation des geplanten Entwurfs

In folgendem Kapitel werden die grundlegenden Entscheidungen des Entwurfs erklärt und durch die Rahmenbedingungen begründet. Ein intuitiver Einstieg soll durch das Schrittweise heranführen an das System über Erklärung des Netzwerkaufbaus, der Angriffsarten sowie Abwehrmechanismen, hin zu den UML-Diagrammen geboten werden.

8.1 Netzwerkaufbau

Die Abbildung 8.1 zeigen den typischen, zu erwartenden Netzwerkaufbau, welcher in dieser Form im Internet und in der Produktivumgebung vorkommen würde. Das System untergliedert sich grob in drei Teile. Links im Bild ist jeweils das Internet zu erkennen, in diesem sind verschiedene Netzwerke mit jeweils verschiedenen Computern miteinander verbunden. Unter den vielen Computern im Internet, welche für Serversysteme teilweise harmlos sind, befinden sich allerdings auch einige Angreifer. Hier ist ganz klar eine Unterscheidung vorzunehmen zwischen dem Angriff eines einzelnen Angreifers, oder einer Menge von einem Angreifer gekapterten und gesteuerten Computer, also eines Botnets.

Wird das Internet, hin zum zu schützenden Netzwerk, verlassen, so wird zuerst ein Router vorgefunden, welcher Aufgaben wie die Network Address Translation vornimmt. Hinter diesem Router befände sich im Produktiveinsatz nun das zu entwickelnde System. Router und zu entwickelndes System sind ebenfalls über eine Verbindung mit ausreichend, in diesem Fall 25Gbit/s, Bandbreite verbunden. Das System selbst agiert als Mittelsmann zwischen Router, also im Allgemeinen dem Internet, und dem internen Netz. Um mehrere Systeme gleichzeitig schützen zu können, aber dennoch die Kosten gering zu halten, ist dem zu entwickelnden System ein Switch nachgeschaltet, mit welchem wiederum alle Endsysteme verbunden sind.

Leider ist durch Begrenzungen im Budget, der Ausstattung der Universität sowie der Unmöglichkeit das Internet in seiner Gesamtheit nachzustellen ein exakter Nachbau des Systems für dieses Projekt nicht möglich, weswegen ein alternativer Aufbau gefunden werden musste, der allerdings vergleichbare Charakteristika aufweisen muss.

Der für das Projekt verwendete Versuchsaufbau untergliedert sich ebenfalls in drei Teile, auch hier

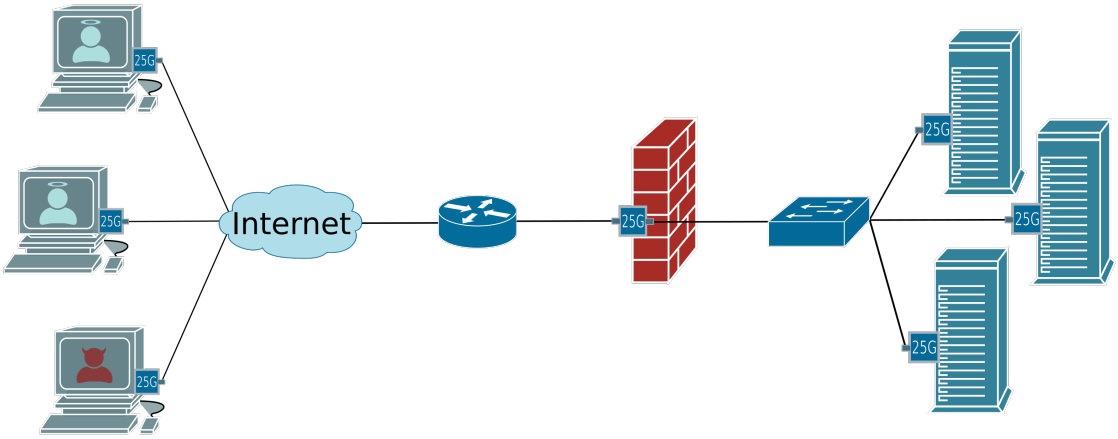

Abbildung 8.1: Realaufbau unter Verwendung eines Angreifers

beginnt die Darstellung 8.2 ganz links mit dem System, welches Angreifer und legitimen Nutzer in sich vereint. Um also die Funktionalität von Angreifer und Nutzer gleichzeitig bereitstellen zu können, setzt der Projektstab in diesem Fall auf das Installieren zweier Netzwerkkarten in einem Computer. Eine 10Gbit/s Netzwerkkarte ist mit der Aufgabe betraut, legitimen Verkehr zu erzeugen. Da aufgrund der Hardwarerestriktionen keine direkte Verbindung zur Middlebox aufgebaut werden kann, wird der ausgehende Verkehr dieser Netzwerkkarte in einen Eingang einer zweiten, in dem selben System verbauten Netzwerkkarte mit einer maximalen Datenrate von 25Gbit/s eingeführt. Von dieser führt ein 25Gbit/s Link direkt zur Middlebox. Intern wird nun im System der rechten Seite sowohl legitimer Verkehr erzeugt als auch Angriffsverkehr kreiert, wobei diese beiden Paketströme intern zusammengeführt werden, und über den einzigen Link an die Middlebox gemeinsam übertragen werden. Die Middlebox selbst ist nicht nur mit dem externen Netz verbunden, sondern hat über die selbe Netzwerkkarte auch noch eine Verbindung ins interne Netz. Das gesamte interne Netz wird im Versuchsaufbau durch einen einzelnen, mit nur 10Gbit/s angebundenen Computer realisiert.

Die Entscheidung zur Realisierung in dieser Art fiel, da insbesondere der Fokus darauf liegen soll, ein System zu erschaffen, welches in der Lage ist, mit bis zu 25Gbit/s an Angriffsverkehr und legitimen eingehenden Verkehr zurechtzukommen. Aus diesem Grund ist es ausreichend, eine Verbindung zum internen Netz mit nur 10Gbit/s aufzubauen, da dieses System bei erfolgreicher Abwehr und Abschwächung der Angriffe mit eben diesen maximalen 10Gbit/s an legitimen Verkehr zurecht kommen muss. Ursächlich für die Verwendung der 10Gbit/s Netzwerkkarte im externen Rechner, welcher hierüber den legitimen Verkehr bereitstellen soll, ist, dass der Fokus bei einem solchen Schutzmechanismus natürlich darauf beruht, die Datenrate des Angreifers zu maximieren, um das zu entwickelnde System in ausreichendem Maße belasten und somit Stresstests unterwerfen zu können.

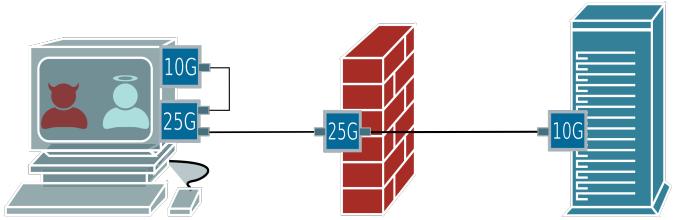

Abbildung 8.2: Versuchsaufbau

8.2 Angriffsvarianten und Abwehrmechanismen

Das zu entwickelnde System muss Schutz bieten, gegen die bereits in der Analyse erwähnten Angriffe. Hierunter fallen die SYN-Flut, der SYN-FIN beziehungsweise SYN-FIN-ACK Angriff, sowie die beiden Ausformungen des Sockstress, TCP Small- und TCP Zero-Window. Des weiteren muss Schutz gegen die UDP Flut geboten werden.

Zur Verhinderung der Angriffe ist im Aufbau 8.2 vorgesehen, dass sich eine Middlebox zwischen dem zu schützenden System und dem externen Netz befindet. Um einen effektiven Schutz gegen oben genannte Angriffe bieten zu können, wird eine Inspektion der Verbindungsströme zum Einsatz kommen. Prinzipiell ist vorgesehen, dass die Middlebox alle gesendeten Pakete abfängt, inspiziert, klassifiziert und bei Erkennung von Angriffen auf diese reagiert, sowie legitimen Verkehr an die entsprechenden Server und Nutzer weiterleitet. Die Detektion und Klassifizierung geschieht dabei individuell nach Angriffsmuster.

8.2.1 SYN-FIN und SYN-FIN-ACK Attacke

Eine SYN-FIN-(ACK) Attacke zeichnet sich dadurch aus, dass ein Angreifer, oftmals in Form eines Botnets, in Abbildung 8.3 auf der rechten Seite zu sehen, unablässig TCP Pakete mit gesetztem SYN und FIN Flag an das Opfer schickt. In Folge dessen ist es möglich, dass der Server in einem Close_Wait Zustand gefangen ist, und in Folge dessen betriebsunfähig wird.

Zur Erkennung einer SYN-FIN sowie SYN-FIN-ACK Attacke werden die gesetzten TCP Flags verwendet, sobald in einem TCP Paket sowohl SYN als auch FIN oder in zweitem Fall zusätzlich noch das ACK-Flag gesetzt sind, klassifiziert die Software das Paket als verdächtig und verwerfen dieses in Folge der Abwehrstrategie.

Abbildung 8.17 zeigt den wohl einfachsten Ablauf einer Abwehrstrategie, bei der SYN-FIN-(ACK) Attacke reicht der Analyzer selbst aus um die Abwehr zu realisieren. Zuerst erhält der Analyzer die Paketinfos in Form eines rte_ring Pointers, es folgt die Detektion des Angriffs, sobald ein Paket erkannt wurde, welches die Angriffscharakteristika aufweist, wird dieses gedroppt.

8.2.2 SYN-Flut

Eine SYN-Flut Attacke läuft prinzipiell wie folgt ab. Der Angreifer, in Abbildung 8.4 mittig dargestellt, schickt viele TCP Pakete mit gesetztem SYN-Flag an das Opfer, und bekundet somit den Wunsch eine Verbindung aufzubauen. Der Server antwortet mit TCP Pakten mit gesetzten SYN-ACK Flags und speichert erste Informationen über die sich im Aufbau befindende Verbindung, welche allerdings aufgrund von gefälschten IP Absenderadressen oder einer Verwerfstrategie am Angriffsrechner nie mit dem vom Server erwarteten TCP Paket mit ACK Flag beantwortet wird. Der Server hält also Informationen für Verbindungen vor, welche nie korrekt aufgebaut wurden

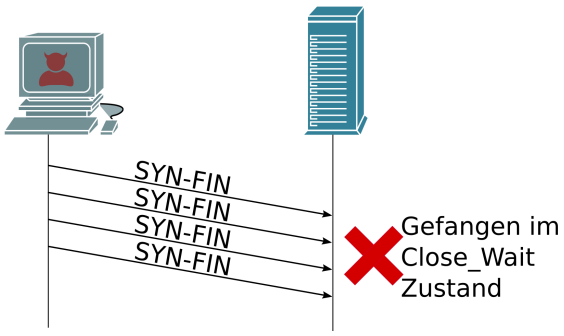

Abbildung 8.3: Schematische Darstellung der SYN-FIN Attacke

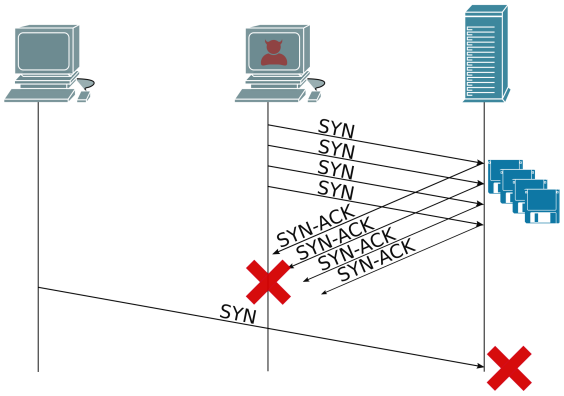

Abbildung 8.4: Schematische Darstellung der SYN-Flut Attacke

und werden. Dies führt im schlechtesten Fall zu abgelehnten Verbindungen von legitimen Nutzern oder einem Totalausfall des Servers.

Die Erkennung einer SYN-Flut beruht auf der Zählung der eingehenden TCP Pakete, welche das SYN-Flag gesetzt haben, den ausgehenden Paketen, welche das SYN und ACK Flag gesetzt haben, sowie der Anzahl an zu den Verbindungsaufbauanfragen gehörenden ACKs, also den letztendlich gelungenen Verbindungsaufbauten.

Um einer SYN-Flut standzuhalten, setzt das zu entwickelnde System auf die Verwendung von SYN-Cookies, hierbei speichert die Middlebox keine Informationen über halboffene Verbindungen, sondern lagert diese Informationen in den Paketheader aus. Die Sequenznummer eines Paketes enthält nun das Ergebnis einer Hashfunktion, welche die Informationen, welche vorher gespeichert waren, kodiert. Dies beschreibt wie sich die Middlebox selbst vor einer SYN-Flut schützt. Um allerdings die dahinterliegenden Server zu schützen, übernimmt die Middlebox auch eine weitere Funktion. Sie baut für alle zu schützenden Systeme die TCP Verbindungen mit externen Hosts auf, sobald dies erfolgreich geschehen ist, wird durch das zu entwickelnde System eine weitere Verbindung mit dem Zielrecher des ursprünglich von außen ankommenden Pakets auf, und leitet fortan Pakete von der externen Verbindung an die Interne weiter. Dies geschieht unter anderem durch eine Anpassung von Sequenznummern mittels eines verbindungsspezifischen Offsets, nicht zu vergessen ist hierbei die Anpassung der Checksumme und anderer Headerfelder.

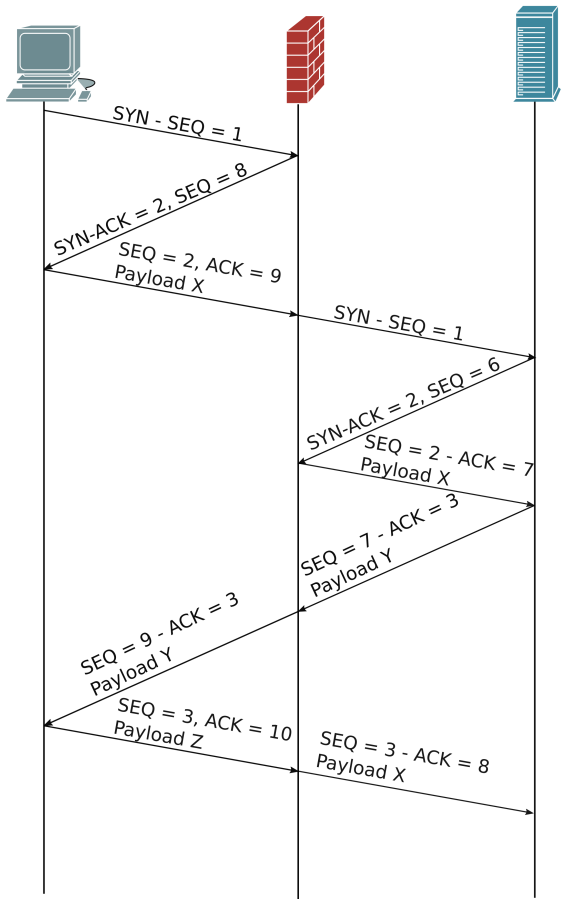

Abbildung 8.5: Schematische Darstellung des Abwehr der SYN-Flood

Abbildung 8.5 soll den geplanten Aufbau nochmals verdeutlichen und insbesondere zum besseren Verständnis des Sequenznummermappings beitragen. Wie in der Abbildung angedeutet, findet zuerst ein Verbindungsaufbau zwischen Middlebox und externem System statt, sobald die Middlebox ein passendes ACK vom externen Nutzer erhalten hat, wird von der Middlebox eine Verbindung zum eigentlichen Ziel aufgebaut. In Folge dessen muss nun für jedes nachfolgende Paket, unabhängig davon, ob dieses von Innen nach Außen oder umgekehrt geschickt wird, eine Anpassung der Sequenznummern vorgenommen werden. Anzumerken ist hierbei, dass in Abbildung 8.5, die Payload X als Anfrage, und die Payload Y als Antwort hierauf zu verstehen ist.

8.2.3 Zero-Window

Die TCP-Zero-Window Attacke, siehe Abbildung 8.6 beruht darauf, dass ein Angreifer mehrere Verbindungen zu einem Opferserver aufbaut, und sein Empfangsfenster auf Größe 0 setzt. Somit werden auf dem Server Ressourcen für eine Verbindung verbraucht, welche keinen legitimen Zweck hat. Werden sehr viele solcher Verbindungen aufgebaut, kann es gar geschehen, dass ein Server keinerlei legitime Verbindungen mehr aufbauen kann. Allerdings kann es auch in legitimen

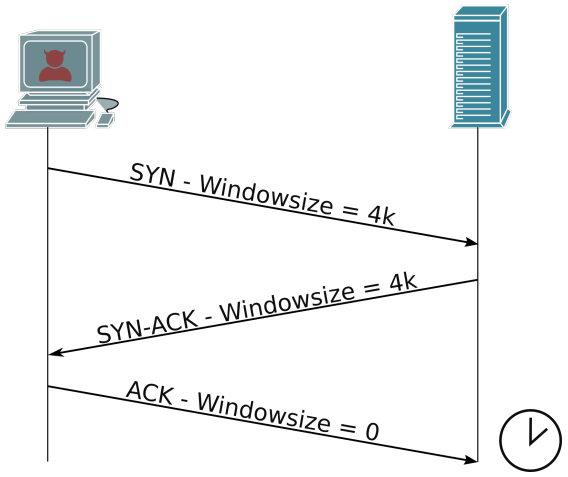

Abbildung 8.6: Schematische Darstellung der Zero-Window Attacke

Verbindungen kurzzeitig dazu kommen, dass das Empfangsfenster des Nutzers auf die Größe null gesetzt wird.

Um die TCP Zero Window Attacke zu erkennen, wird unter Anderem die Zählung aller Verbindungen, welche ein Zero-Window ankündigen, verwendet. Da dies alleine allerdings zur Erkennung nicht ausreichend ist, muss eine zeitliche Statistik über diese Verbindungen geführt werden. Sobald eine TCP Verbindung über einen gewissen Zeitraum ein Zero-Window ankündigt und dies nie verändert, wird die Verbindung als auffällig erkannt. Bei genauerer Kenntnis der zu schützenden Systeme wäre es ebenfalls möglich zu zählen, wie viele TCP-Verbindungen zu einem gewissen System mit Fenstergröße null aufgebaut sind. Hierdurch könnte nun bei Kenntnis der maximal erlaubten konkurrierenden Verbindungen eines Webservers, errechnet werden, ob die Anzahl der Verbindungen mit Fenstergröße null einen kritischen Anteil erreicht. Als Lösungsstrategie ist hier ein Timeout für TCP-Verbindungen mit Fenstergröße 0 vorgesehen, sowie eine Ratenbegrenzung von TCP-Verbindungen pro Zielsystem.

8.2.4 Small-Window

Ein TCP-Small Window Angriff, siehe 8.7 beruht prinzipiell darauf, das ein Angreifer eine TCP Verbindung zu einem Server aufbaut und direkt eine sehr große Datei anfordert. Hierbei setzt der Angreifer gleichzeitig sein Empfangsfenster auf eine sehr kleine Größe, sodass der sendende Server die angeforderte Ressource zuvor in viele kleine Fragmente aufteilen muss. Dies kostet nicht nur Speicherplatz, sondern auch Rechenleistung. Als Resultat ist es möglich, dass der Server Speicherüberläufe erleidet und gänzlich abstürzt.

Die Erkennungsstrategie von TCP Small Window stellt sich als schwieriger zu Realisieren dar, hier sind bislang verschiedene Optionen im Gespräch. Eine Auffälligkeit, welche ein Angreifer aber leicht umgehen könnte, sind der konstante Einsatz von nur einer Fenstergröße. Unter Umständen lässt sich die Erkennung noch verbessern, wenn das zu entwerfende System eine Übersicht über alle abrufbaren Ressourcen der zu schützenden Systeme besitzt. Die Verhinderung oder Abschwächung von TCP Small Window Angriffen beruht auf dem Setzen eines absoluten Verbindungstimeout, dessen Wert sich an der mittleren Verbindungsdauer für diesen Server oder dieses Netzwerk orientiert. Um den Schutz noch effektiver zu gestalten wird des Weiteren eine

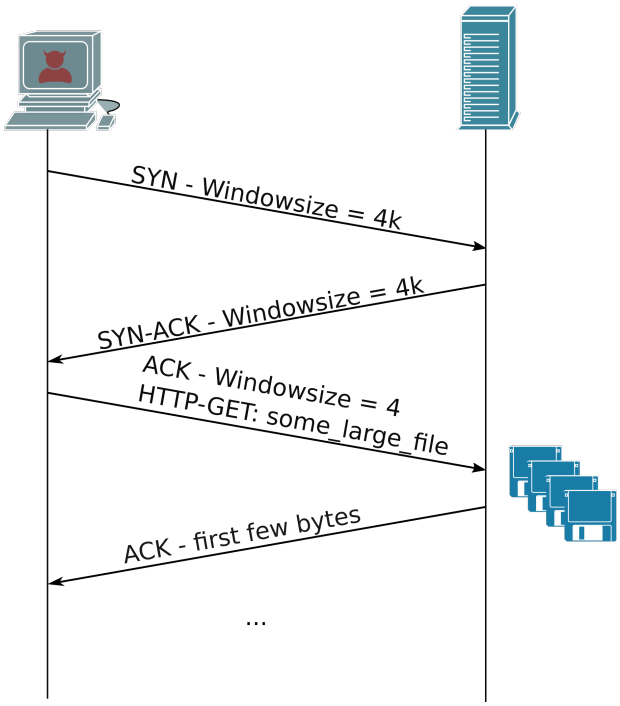

Abbildung 8.7: Schematische Darstellung der Small-Window Attacke

minimale, nicht zu unterschreitende Datenrate definiert. Fallen mehrere Verbindung unter diese Rate, so sind die ältesten als nicht legitim zu erachten und zu beenden.

8.2.5 UDP-Flood

Die UDP-Flut, siehe Abbildung 8.8, zeichnet sich durch eine hohe Anzahl an UDP-Nachrichten an einzelne Zielsysteme aus. Für jede Nachricht prüft das Opfersystem, ob gerade eine Anwendung auf diesem Port lauscht, ist dies nicht der Fall, so reagiert es mit Port unreachable Nachrichten. Die Überprüfung und der Versand dieser Nachrichten benötigt Rechenleistung und Zeit. Im schlechtesten Fall wird durch diesen Angriff der Verkehr komplett zum Erliegen gebracht.

Zur Erkennung dieser würde auf die Verbindungsstatistik, welche darstellt, auf welchen Zielrechnern vermehrt Ports angefragt werden, gesetzt. Wird bei einem zu schützenden Server eine fest definierte Abfragerate überschritten, würde das System eine UDP Flood erkennen. Da dies allerdings im angestrebten Abwehrszenario nicht benötigt wird, um den Angriff abzuwehren, findet im System diese Erkennung ausschließlich zu statistischen Zwecken statt. Um die UDP-Flut allerdings erfolgreich Abwehren zu können, setzt das System auf eine Ratenlimitierung für UDP Traffic pro zu schützendem System.

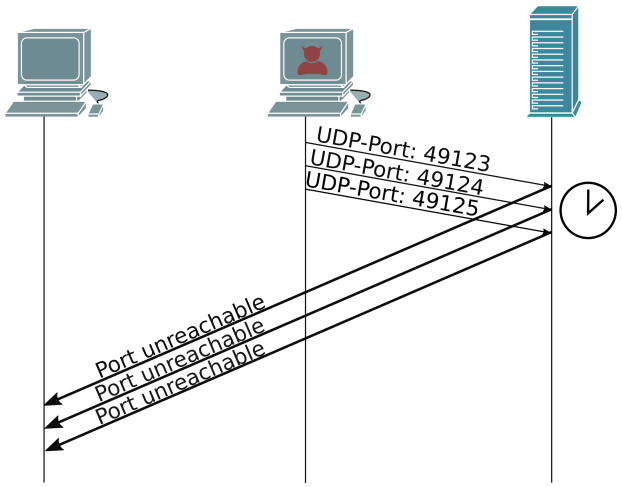

Abbildung 8.8: Schematische Darstellung der UDP-Flood

8.3 Grundlegender Aufbau der Software

Das Grundprinzip der zu entwickelten Software soll sein, Pakete auf einem Port der Netzwerkkarte zu empfangen und diese zu einem anderen Port weiterzuleiten. Zwischen diesen beiden Schritten werden die Pakete untersucht, Daten aus diesen extrahiert und ausgewertet. Im weiteren Verlauf des Programms werden Pakete, welche einem Angriff zugeordnet werden verworfen, und legtime Pakete zwischen dem interen und externen Netz ausgetauscht. Es bietet sich an, hier ein Pipelinemodell zu verwenden wobei die einzelnen Softwarekomponenten in Pakete aufgeteilt werden. Im ConfigurationManagement werden die initialen Konfigurationen vorgenommen. Das NicManagement ist eine Abstraktion der Netzwerkkarte und sorgt für das Empfangen und Senden eines von Paketen. Die PacketDissection extrahiert Daten von eingehenden Paketen. Die Inspection analysiert diese Daten und bestimmt, welche Pakete verworfen werden sollen. Das Treatment behandelt die Pakete nach entsprechenden Protokollen. Um die Abarbeitung dieser Pipeline möglichst effizient zu gestalten soll diese jeweils von mehreren Threads parallel und möglichst unabhängig voneinander durchschritten werden.

In den folgenden Sektionen wird näher auf den Kontrollfluss innerhalb des Programms, auf den Einsatz von parallelen Threads und auf die einzelnen Pakete näher eingegangen.

8.3.1 Kontrollfluss eines Paketes

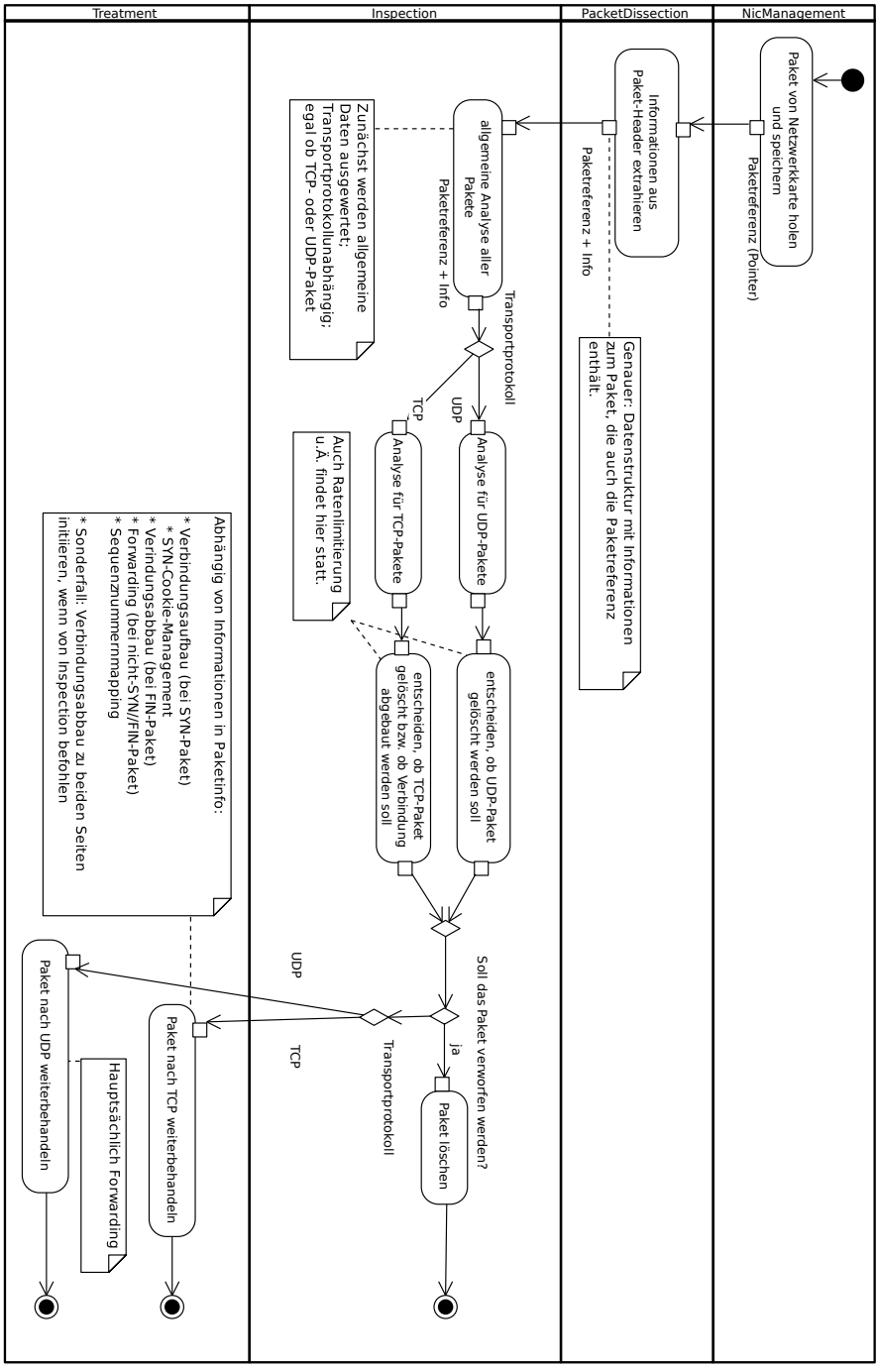

Abbildung 8.9: Schematische Darstellung des Kontrollflusses

In diesem Abschnitt soll veranschaulicht werden, wie die Behandlung eines Paketes vom Nic-Management bis zum Treatment erfolgt. Dabei werden die Pakete selbst als Akteure angesehen und noch nicht deren Klassen. Hinweis: Ein Thread soll später mehrere Pakete auf einmal durch die Pipeline führen. In diesem Diagramm wird zur Übersichtlichkeit jedoch nur der Fluss eines Paketes gezeigt. Dieser lässt sich dann einfach auf eine größere Menge von Paketen anwenden. Ein Aktivitätsdiagramm ist unter Abbildung 8.9 am Ende der Sektion 8.3 zu finden. Die Aktion „Paket nach TCP behandeln" ist auf Abbildung 8.21 genauer veranschaulicht.

8.3.2 Einsatz von parallelen Threads

Zunächst ist jedoch ein wichtiger Aspekt der Architektur hervorzuheben. Von der Mitigation-Box wird gefordert, eine hohe Paket- und Datenlast verarbeiten zu können. Das Hardwaresystem, auf welchem das zu entwickelnde Programm laufen wird, besitzt eine Multicore-CPU, d.h. das System ist in der Lage, Aufgaben aus unterschiedlichen Threads parallel zu bearbeiten. Dies hat das Potential, die Rechengeschwindigkeit zu vervielfachen und so die Bearbeitungszeit für jedes Paket zu verringern.

Dabei stellt sich die Frage, was die Threads im Programm genau tun. Es wäre zum Beispiel möglich, dass jeder Thread eine Aufgabe übernimmt, d.h. es gäbe einen Thread, der nur Daten analysiert oder einen Thread, der nur Paketinformationen extrahiert. Eine solche Aufteilung würde allerdings zu einem hohen Grad an Inter-Thread-Kommunikation führen. Diese ist nicht trivial und kann einen Großteil der verfügbaren Ressourcen benötigen, was den durch die Parallelisierung erzielten Gewinn wieder zunichte machen könnte. Um dieses Risiko zu vermeiden soll stattdessen jeder Thread die gesamte Pipeline durchlaufen. So ist kaum Inter-Thread-Kommunikation notwendig. Außerdem ist es dann verhältnismäßig einfach, den Entwurf skalierbar zu gestalten: Wenn ein Prozessor mit größerer Anzahl an Kernen verwendet werden würde, könnten mehr Pakete parallel bearbeitet werden ohne dass die Architektur geändert werden muss.

Verwendung von Receive-Side-Scaling

Ein weiterer grundlegender Vorteil ergibt sich durch das von der Netzwerkkarte und von DPDK unterstützte Receive Side Scaling (RSS), siehe Abbildung 8.10: Ein auf einem Port eingehendes Paket wird einer von mehreren sogenannten RX-Queues zugeordnet. Eine RX-Queue gehört immer zu genau einem Netzwerkkartenport, ein Port kann mehrere RX-Queues besitzen. Kommen mehrere Pakete bei der Netzwerkkarte an, so ist die Zuordnung von Paketen eines Ports zu seinen RX-Queues gleich verteilt - alle RX-Queues sind gleich stark ausgelastet. Diese Zuordnung wird durch eine Hashfunktion umgesetzt, in die Source und Destination Port-Nummer und IP-Adresse einfließen. Das führt dazu, dass Pakete, die auf einem Port ankommen und einer bestimmten Verbindung zugehören immer wieder zu der selben RX-Queue dieses Ports zugeordnet werden.

Ferner besteht die Möglichkeit, Symmetric RSS einzusetzen. Dieser Mechanismus sorgt dafür, dass die Pakete, die auf dem einen Port der Netzwerkkarte ankommen nach genau dem selben Prinzip auf dessen RX-Queues aufgeteilt werden, wie die auf dem anderen Port ankommenden Pakete auf dessen RX-Queues. So ergeben sich Paare von RX-Queues, die jeweils immer Pakete von den gleichen Verbindungen beinhalten. Angenommen, die RX-Queues sind mit natürlichen Zahlen benannt und RX-Queue 3 auf Port 0 und RX-Queue 5 auf Port 1 sind ein korrespondierendes RX-Queue-Paar. Wenn nun ein Paket P, zugehörig einer Verbindung V auf RX-Queue 3, Port 0 ankommt, dann weiß man, dass Pakete, die auf Port 1 ankommen und der Verbindung V angehören immer auf RX-Queue 5, Port 1 landen.

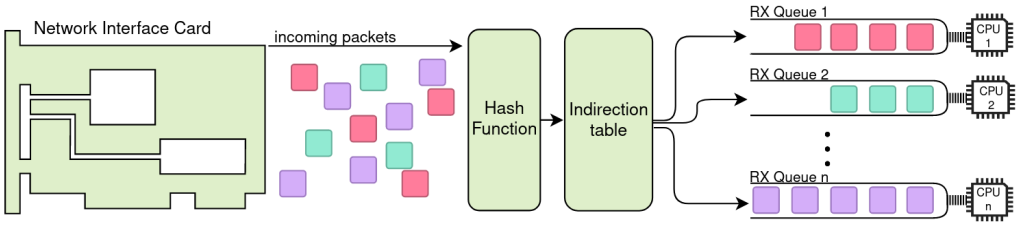

Abbildung 8.10: Beispielhafte Paketverarbeitung mit Receive Side Scaling

Neben RX-Queues existieren auch TX-Queues (Tranceive-Queues), die ebenfalls zu einem bestimmten Port gehören. Darin befindliche Pakete werden von der Netzwerkkarte auf den entsprechenden Port geleitet und gesendet.

Auf Basis dieses Mechanismus sollen die Threads wie folgt organisiert werden: Einem Thread gehört ein Paar von korrespondierenden RX-Queues (auf verschiedenen Ports) und daneben eine TX-Queue auf dem einen und eine TX-Queue auf dem anderen Port. Das bringt einige Vorteile mit sich: Es müssen zwei Arten von Informationen entlang der Pipeline gespeichert, verarbeitet und gelesen werden: Informationen zu einer Verbindung und Analyseinformationen/Statistiken. Daher ist kaum Inter-Thread-Kommunikation nötig, weil alle Informationen zu einer Verbindung in Datenstrukturen gespeichert werden können, auf die nur genau der bearbeitende Thread Zugriff haben muss. Bei Analyseinformationen ist das leider nicht der Fall. Diese müssen zumindest teilweise global gespeichert werden. Hier bietet es sich an, einen Thread für das Management dieser globalen Informationen zu verwenden, der dann mit den restlichen Threads kommuniziert. So müssen zumindest nur mehrere Threads mit genau einem Thread Informationen austauschen und nicht alle Threads untereinander.

Im Falle eines Angriffes ist die Seite des Angreifers (entsprechender Port z.B. „Port 0") viel stärker belastet, als die Seite des Servers (z.B. „Port 1"). Wegen der gleich verteilten Zuordnung des eingehenden Traffics auf die RX-Queues und weil ein Thread von RX-Queues von beiden Ports regelmäßig Pakete pollt, sind alle Threads gleichmäßig ausgelastet und können die Pakete bearbeiten. Ein günstiger Nebeneffekt bei DDOS-Angriffen ist, dass die Absenderadressen von Angriffspaketen oft sehr unterschiedlich sind. Das begünstigt die gleichmäßige Verteilung von Paketen auf RX-Queues.

8.3.3 Paketdiagramm

Grundsätzlich ist es angedacht, wie im Paketdiagramm 8.11 ersichtlich, die zu entwickelnde Software in insgesamt 5 Teile zu untergliedern.

Das NicManagement wird eingesetzt, um die Kommunikation und Verwaltung der Netzwerkkarten und Ports zu ermöglichen, hier finden Operationen wie der Versand und Empfang von Paketen statt. Verwendet wird das NicManagement von der PacketDissection, welche ihrerseits dazu zuständig ist, die für die Erkennung und Klassifizierung von Angriffen benötigten, Informationen aus den einzelnen Headern eines Netzwerkpakets zu extrahieren. Die hieraus gewonnenen Informationen werden von der Inspection verwendet um sowohl Angriffe erkennen zu können, als auch über den allgemeinen Zustand des Systems in Form von Statistiken Auskunft zu geben. Das Treatment, welches sich um die Abwehrwehrmaßnahmen der verschiedenen Angriffe kümmert, verwendet hierzu die durch die Inspection bereitgestellten Ergebnisse und Informationen. Für

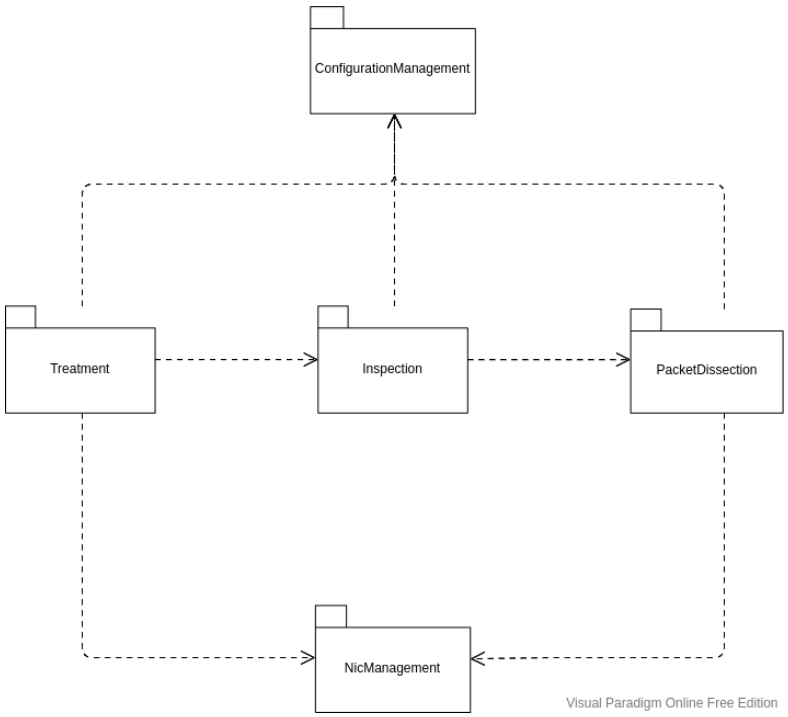

Abbildung 8.11: Paketdiagramm

das Versenden und Verwerfen von Pakten, sowie den Aufbau und das Terminieren von Verbindungen, verwendet das Treatment wiederum das NicManagement. Sowohl Treatment, als auch Inspection sowie PacketDissection verwenden den ConfigurationManagement, welches nicht nur die Aufgaben des Programmstarts übernimmt und für die Initialisierung sorgt, sondern auch wichtige Paramter für andere Programmbestandteile in Form von Konfigurationsdateien vorhält und einpflegt. Zudem ist das ConfigurationManagement die einzige Möglichkeit für den Nutzer, um aktiv Einstellungen am System vorzunehmen.

8.3.4 Klassendiagramm

Das Klassendiagramm untergliedert sich grundlegend in die selben Bestandteile wie das Paketdiagramm, siehe Bild 8.11.

NicManagement

Das NicManagement, siehe Abbildung 8.12, ist eine Abstraktion der Netzwerkkarte. Hier werden die Pakete von dem Interface gepollt und in den Speicher geschrieben. Um später mit diesen Paketen in anderen Paketen weiterarbeiten zu können, übergibt das NicManagement Pointer auf die Speicherbereiche, in denen sich die Pakete befinden. Der NetworkPacketHandler verwendet einen packet_store rte-ring zum Abspeichern der empfangenen und zu versendenden Pakete.

Als Funktionalitäten stellt der NetworkPacketHandler sowohl send_packets(pkt_info_arr:PacketInfo*[]) mit Rückgabetyp void, als auch poll_packets() mit Rückgabetyp rte_mbuf*[] bereit.

rte_mbuf*[] in der Methode poll_packets() steht für ein Array aus Pointern, die jeweils auf ein

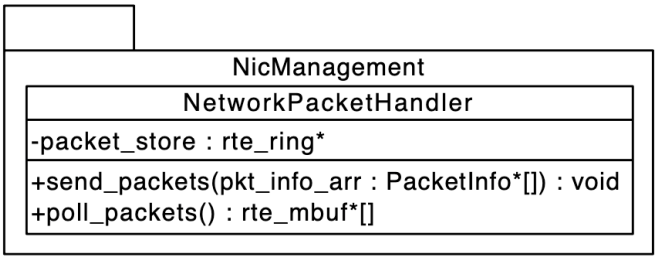

Abbildung 8.12: Klasse NetworkPacketHandler im Package NicManagement

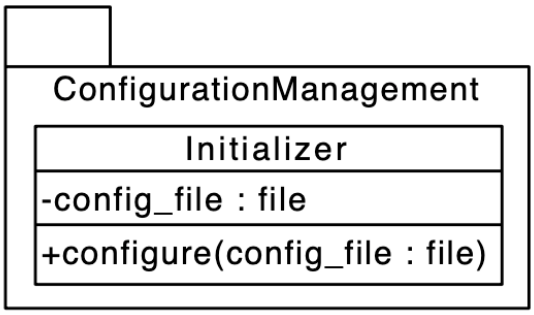

Abbildung 8.13: Klasse Initializator im Package ConfigurationManagement

mbuf-Objekt zeigen. Hier ist nicht relevant, ob dies später auch wirklich als Array implementiert wird. Es soll vielmehr gezeigt werden, dass mehrere mbuf-Pointer auf einmal zurückgegeben werden.

ConfigurationManagement

Im Paket ConfigurationManagement, Abbildung 8.13 findet die Initialisierung des Programms, sowie die Verwaltung der Konfigurationsdateien statt.

Als Attribut ist hier ausschließlich eine Konfigurationsdatei config_file vom Typ conf vorgesehen.

Die Operationen beschränken sich auf configure(config_file:file), wobei der Nutzer hier die Möglichkeit hat, über die Eingabe einer Konfigurationsdatei einzelne Parameter anzupassen.

PacketDissection

Die PacketDissection, Abbildung 8.14 nimmt Pointer auf Pakete entgegen und extrahiert daraus die für die Erkennung und Behandlung wichtigen Headerinformationen. Weitergegeben werden dann Zeiger auf von der PacketDissection angelegte PacketInfo-Strukturen, genau eine für jedes Paket. Diese Struktur beinhaltet die extrahierten Informationen und den Pointer auf das zugehörige Paket, der vom NicManagement erhalten wurde.

Hierzu werden zwei Klassen verwendet, HeaderExtractor kümmert sich um das Extrahieren der Headerinformationen, PacketInfo wiederum wird als eigener Datentyp verwendet.

Die Klasse HeaderExtractor verwendet vorerst keinerlei Attribute und bietet nur die Methode

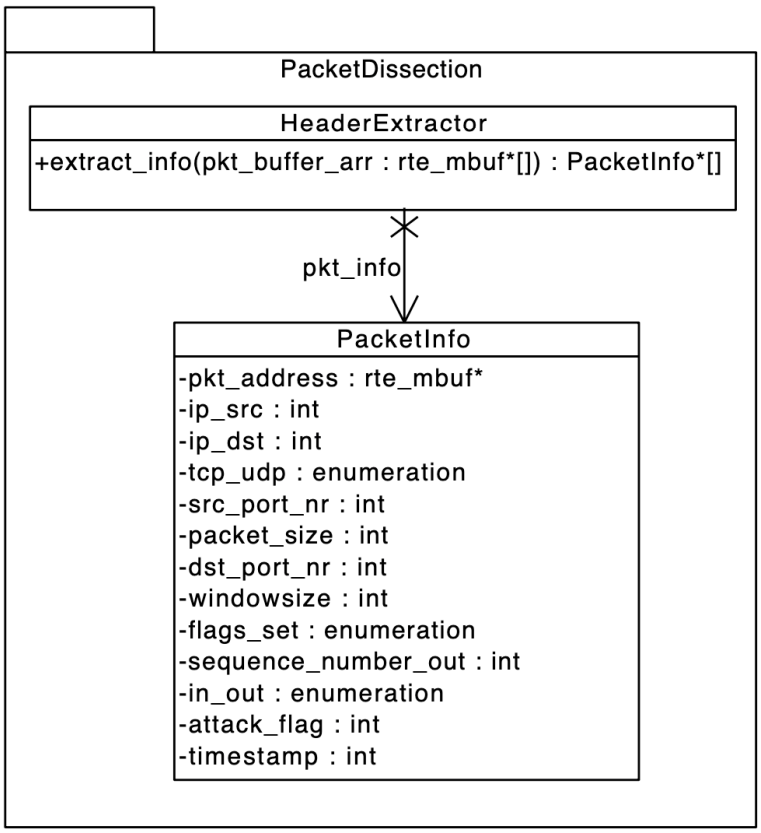

Abbildung 8.14: Klassen HeaderExtractor und PacketInfo im Package PackageDissection

extract_info(pkt_buffer_arr: rte_mbuf*[]): PacketInfo*[] bereit, welche ihrerseits ein Array aus rte_mbuf Pointern erhält und als Resultat ein Array aus PacketInfo Pointern zurückgibt.

Die Klasse PacketInfo ist als Kapselung mehrerer individueller Informationen aus den Netzwerkpaketen gedacht, hierunter fallen beispielsweise die Ziel-IP oder Paketgröße.

Inspection

Die Inspektion, siehe Abbildung 8.15, nimmt die bereitgestellten PacketInfo-Strukturen entgegen. Aus den beinhalteten Daten werden Statistiken und Regeln erstellt und ausgewertet. Gegebenenfalls sind manche Regeln dynamisch anzupassen. In der Inspektion wird beurteilt, ob ein Paket legitim ist oder nicht. Ist es nicht legitim, wird es gelöscht. Wenn es nicht gelöscht wird, wird der Pointer auf die entsprechende PacketInfo-Struktur der Treatment-Komponente übergeben. Dieser Pointer ist die einzige Schnittstelle zwischen Inspektion und Treatment.

Weitere Anweisungen, wie mit dem Paket im Treatment verfahren werden soll können über die PacketInfo-Struktur mitgeteilt werden. Diese Möglichkeit wird genutzt, wenn eine TCP-Verbindung zu beiden Seiten abgebaut werden soll. Es wird eine entsprechende Direktive in die PacketInfo-Struktur geschrieben, die von der Treatment-Komponente ausgewertet wird.

Die Klasse Analyzer nutzt einen packet_info_buf vom Typ Patchmap zum Speichern der Packe-

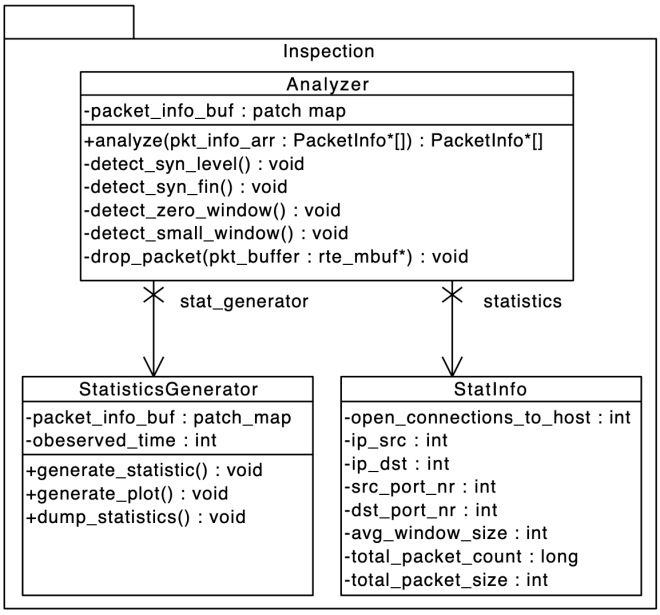

Abbildung 8.15: Klassen StatisticsGenerator, Analyzer und StatInfo im Package Inspection

tInfo Daten, und enthält einen stat_generator vom Typ StatisticsGenerator sowie statistics vom Typ StatInfo. Die Operation analyze(pkt_info_arr: PacketInfo*[]): PacketInfo*[] ist für die Verwaltung und den Aufruf aller weiterer detect-Methoden zuständig. Die jeweiligen detect-Methoden sollen je eine Angriffsart erkennen, und gegebenenfalls bereits erste, als schädlich erkannte Pakete mittels der drop_packet-Methode verwerfen.

Die Klasse StatisticsGenerator ist für die Generierung von Statistiken zuständig, hierzu verwendet es einen packet_info_buf des Typs Patchmap, sowie eine Variable observed_time, welche die bisher verstrichene Zeit speichert. Die Operation generate_statistic() wird dafür eingesetzt, Statistiken für die Analyse durch den Analyzer zu erstellen, sowie informative Daten für den Administrator zu erheben. generate_plot() erzeugt aus den ausgewerteten Daten anschauliche Diagramme, welche dem Administrator des Systems einen grundsätzlichen Überblick über den Zustand des Systems gewährt. dump_statistics() speichert die angefertigten Statistiken dann persistent auf dem System.

Die Klasse StatInfo dient als Speicherkonstrukt für die erhobenen Daten, welche beim Erzeugen und Auswerten einer Statistik anfallen.

Das Paket Inspection könnte, wenn nötig, außerdem eine Tabelle zu jeder Verbindung beinhalten (ähnlich der im TCP-Treatment). Aber es soll auf keinen Fall dieselbe Tabelle wie im TCP-Treatment genutzt werden, um die UML-Pakete zu kapseln.

8.3.5 Treatment

Das Paket Treatment, siehe Abbildung 8.16, hat zur Aufgabe, die zum Schutz der Server benötigten Operationen auszuführen. Hierbei werden zwei Klassen genutzt, wovon sich eine rein um die Behandlung von TCP-Paketen, eine andere nur um die Behandlung von UDP-Paketen vornimmt. Das Treatment nimmt PacketInfo-Struktur-Pointer entgegen und behandelt das Paket

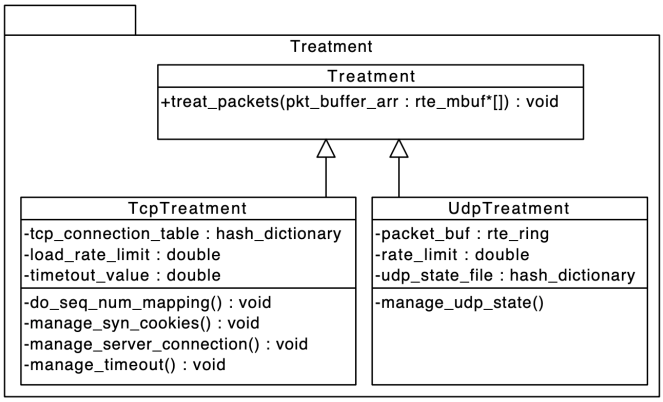

Abbildung 8.16: Klassen StatisticsGenerator, Analyzer und StatInfo im Package Inspection

nach dem entsprechende Transportprotokoll.

Die Klasse TcpTreatment nutzt als Attribute eine tcp_connection_table vom Typ Hashmap, welche alle Informationen zu den einzelnen Verbindungen enthält, ein load_rate_limit des Typs double, sowie einen timeout_value des Typs double. Die Operation manage_syn_cookies() mit Rückgabetyp void erledigt jede, zur Verwaltung der SYN-Cookies benötigte, Aufgabe, wie das Berechnen des Hashwertes oder das Eintragen dieses in den Paketheader. manage_server_connection() übernimmt alle Aufgaben, welche sich mit der Verwaltung der Verbindungen zu internen Servern befassen. Diese Methode wiederum verwendet die Methode do_seq_num_mapping(), um die Verbindungen von externen Servern zur Middlebox, sowie die Verbindungen von internen Servern zur Middlebox zusammenzuführen. Die Funktionen limit_rate() und manage_timeout() werden verwendet, um Angriffen wie TCP-Zero- und Small-Window-Angriffen entgegenzuwirken.

Die Klasse UdpTreatment hat einen packet_buf des Typs rte_ring, ein rate_limit des Typs double, sowie ein udp_state_file des Typs hash_dictionary als Attribute. Als Funktionen stehen hier manage_udp_state zur Verfügung, welches aus dem eigentlich stateless UDP-Protokoll einen State zuweist. Des weiteren ist auch hier die Funktion limit_rate() sowie manage_timeout() verfügbar.

8.4 Sequenzdiagramme

In folgendem Abschnitt finden sich Sequenzdiagramme, welche den genauen Ablauf der Behandlung verdeutlichen sollen.

8.4.1 SYN-FIN-ACK

Abbildung 8.17 beschreibt den Ablauf der Erkennung und Behandlung eines SYN-FIN Angriffs. Zuerst erhält der Analyzer die Paketinformationen und überprüft diese auf Vorhandensein der Flagkombination, welche zur Detektion benötigt wird. Ist ein Paket als schädlich erkannt, so wird es bereits hier verworfen.

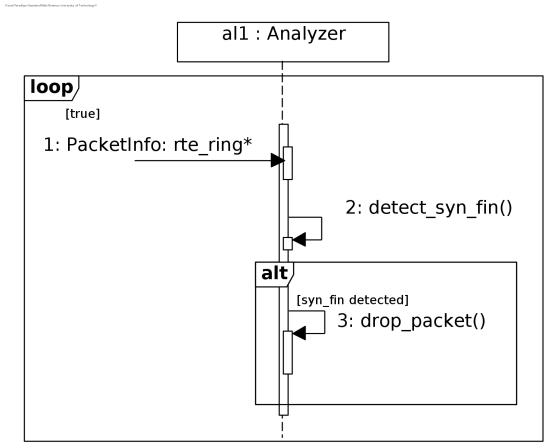

Abbildung 8.17: Sequenzdiagramm der SYN-FIN-ACK-Abwehr

8.4.2 SYN-Flood

Abbildung 8.18 beschreibt den Ablauf der Erkennung und Behandlung einer SYN-Flut. Zuerst erhält der Analyzer die Paketinformationen in Form eines rte_ring Pointers. Die Funktion detect_syn_level() verwendet die dann vom StatisticsGenerator angeforderten StatInfos um das Level eines SYN-Angriffs zu informativen Zwecken zu berechnen. Es folgt die Übergabe der Packetinfos an das TCP-Treatment, welches seinerseits parallel das Management der SYN-Cookies sowie der internen Serververbindung vornimmt. Das Management der internen Serververbindungen selbst verwendet die Funktion do_seq_num_mapping() um die Verbindungen zueinander führen zu können.

8.4.3 Small Windows

Das Diagramm 8.19 zeigt, wie das System Zero-Window- und Small-Window-Angriffe erkennt. Zunächst werden für eine Menge an Paketen die Methoden „detect_zero_window()" und „detect_small_window()" aufgerufen. Wenn ein Angriff erkannt wird, werden die entsprechenden Pakete verworfen. Danach werden alle gültigen Pakete zur normalen Weiterbehandlung and das Treatment weitergegeben.

8.4.4 UDP Flood

Bei der Abwehr eines UDP-Flood-Angriffes muss der Angriff zunächst erkannt werden, zu sehen in Abbildung 8.20. Dies geschieht über die Methode „detect_udp_flood()". Wird eine UDP-Flood erkannt, verwirft das Analyzer-Objekt das Paket, ansonsten wird dieses einem UdpTreatent-Objekt übergeben und gesendet.

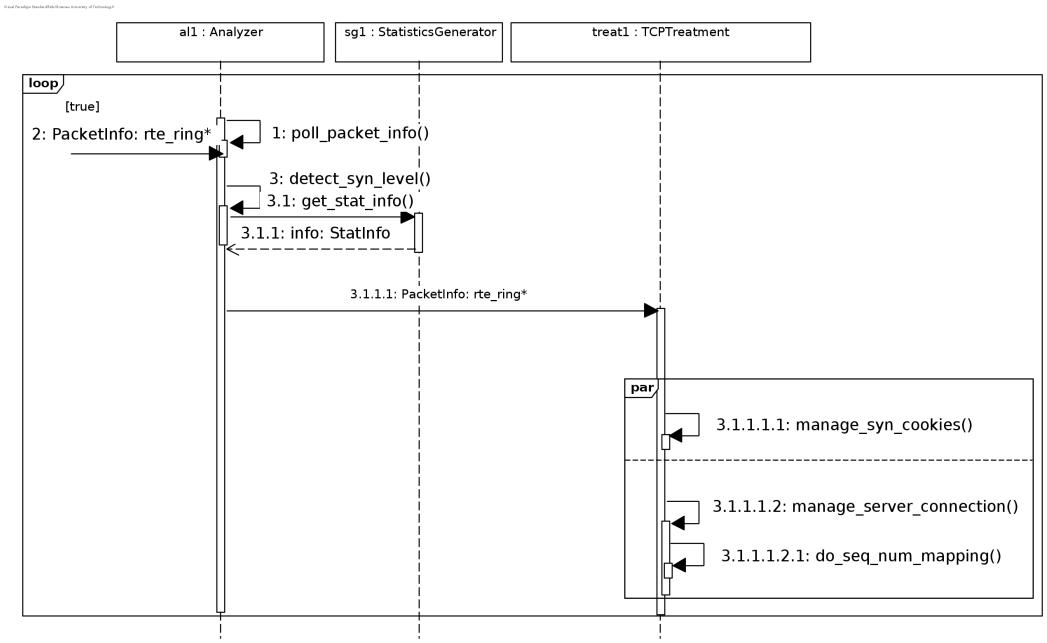

Abbildung 8.18: Sequenzdiagramm der SYN-Flood-Abwehr

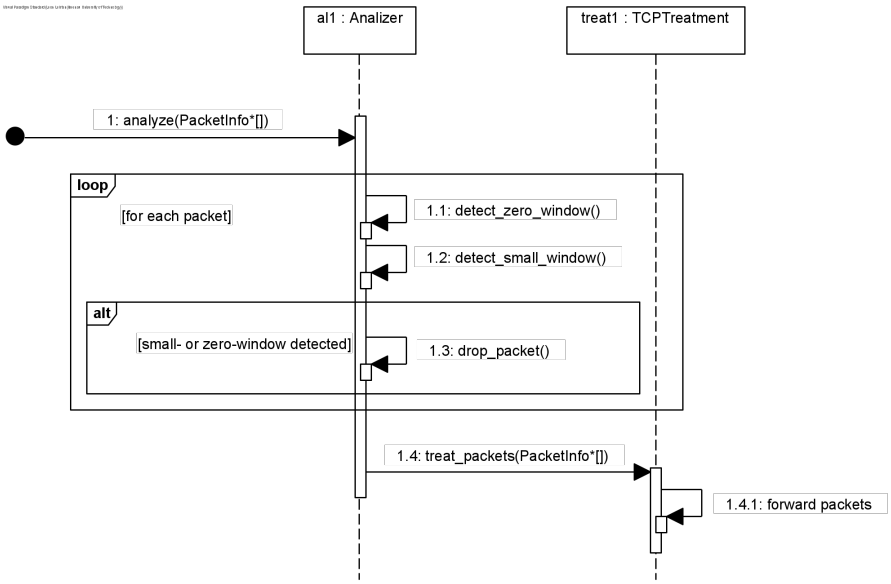

Abbildung 8.19: Sequenzdiagramm der Small-Window Abwehr

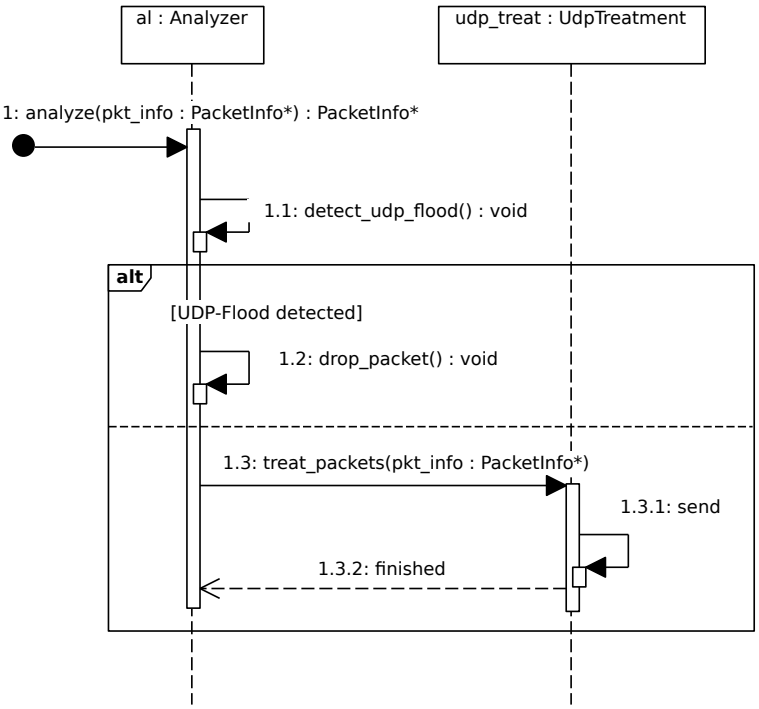

Abbildung 8.20: Sequenzdiagramm der UDP-Flood-Erkennung

8.5 Aktivitätsdiagramme

8.5.1 Behandlung eines Paketes nach TCP

Unter Abbildung 8.21 ist der Ablauf der Aktivität „Paket nach TCP behandeln" zu finden. Die Idee hierbei ist, dass ein Paket-Pointer zusammen mit Informationen über das Paket, die sowohl aus dem Header extrahiert als auch von der Inspektion hinzugefügt wurden dem TcpTreatment übergeben wird. Das Paket wird dann anhand verschiedener Kriterien kategorisiert (SYN-Paket, ACK-Paket, FIN-Paket...) und infolgedessen entsprechend behandelt. Hier wird auch der SYN-Cookie-Mechanismus umgesetzt, welcher vor SYN-Flood-Attacken schützen soll.

Im Diagramm wird folgender Aufbau angenommen: A ist der Initiator eines Verbindungsauf- oder abbaus, B ist derjenige, der auf die Anfrage reagiert. M ist die Mitigation Box. Bei Verbindungsauf- und abbau ist also derjenige, der das erste SYN- bzw. FIN-Paket geschickt hat „A". Der andere Kommunikationsteilnehmer ist demzufolge „B". Hierbei wird nicht nach Angreifer und nicht-Angreifer unterschieden, es geht nur um die Kommunikation über TCP. Wenn bei einer Aktion „A<-M" steht, bedeutet das, dass das die erwähnte Nachricht von M nach A geschickt wird. Wenn bei einem Pin (A->M) steht, bedeutet das, dass das entsprechende Paket von A gekommen ist.

Mit **„initiales ACK"** ist das dritte Paket eines 3-Way-Handshakes gemeint, also das erste, das beim Verbindungsaufbau Nutzdaten enthält. **„SN"** ist eine Abkürzung für „Sequenznummer". **„V-Tabelle"** steht für „Verbindungstabelle", in einem Tupel stehen alle Daten zu genau einer Verbindung; zum Beispiel die Differenz der Sequenznummern von A und B für die Sequenznummerzuordnung. Das Wort „V-Tabelle" ist nicht allgemein und ist nur eine Abkürzung. Im weiteren Entwurf soll diese Tabelle bzw. die entsprechende Datenstruktur einen passenden englischen Namen erhalten.

Um bestimmte Mechanismen auseinanderzuhalten werden zusammengehörige Aktionen farblich gekennzeichnet. **Grün** repräsentiert das Sequenznummermapping, **Hellgrün** das Zwischenspeichern und senden der Nutzdaten aus dem initialen ACK-Paket und **Cyan** das SYN-Cookie-Management.

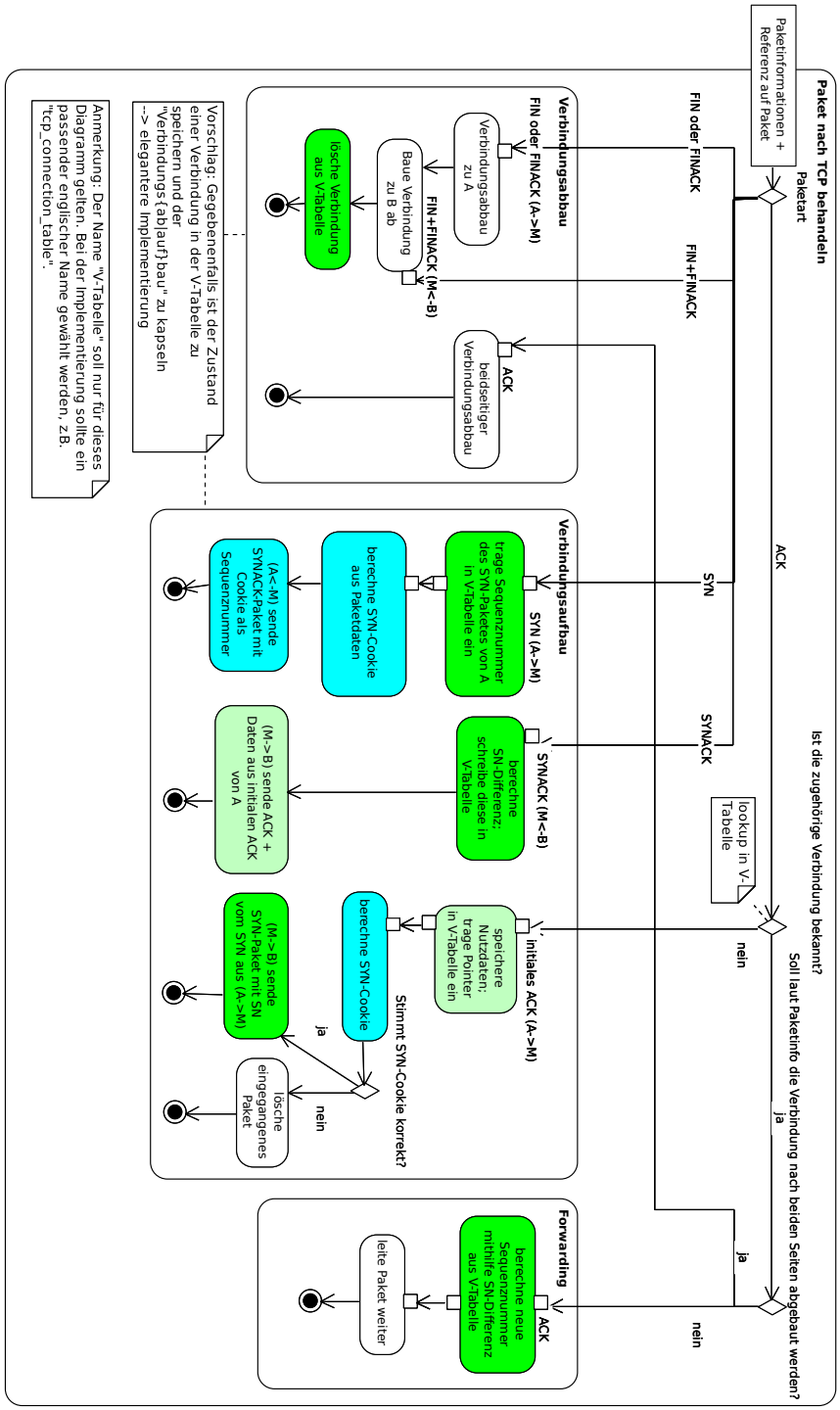

Abbildung 8.21: Paketbehandlung nach TCP

Kapitel 9

Technologien und Entwicklungswerkzeuge

9.1 Hardware

Die Software hat eine Mindestanforderung an ihre zu verwendende Hardware, hierzu gehört, dass eine sehr gute CPU vorliegt, wie z.B. ein Ryzen 9 5900X. Des weiteren wird ein genügend großer Arbeitsspeicher benötigt, dass heißt ab einer Speicherkapazität von 32GB un mehr, und genügend viel Speicherplatz, dass heißt ab 200GB aufwärts. Um das System gut einsetzen zu können müssen auch genügend schnelle Netzwerkkarten vorliegen (z.B. 25GbE), welche DPDK unterstützten.

9.2 Programmiersprache und Bibliotheken

Die für die Software verwendete Programmiersprache ist C++ auf der Version 17 und wird auf dem Betriebssystem Ubuntu entwickelt. Als Programmiereditor und IDE wird VSCodium verwendet, welches eine Community betriebene, frei lizenzierte Distribution der Microsoft IDE VSCode ist. Zur Unterstützung der Entwicklung werden zusätzliche Erweiterungspakete für VSCodium verwendet, unter anderem Meson (1.3.0) und das C/C++ Extension Pack (1.0.0). Es wurde die Programmiersprache C++ gewählt, da die Mitglieder dieses Projektes innerhalb ihres Studiums schon Erfahrungen mit dieser Sprache gewonnen haben, sei es in Form von Praktika oder aus eigenen Bedürfnissen.

Ein Hauptbestandteil des Projektes ist die Verwendung des *Data Plane Development Kit* (kurz *DPDK*), auf der LTS-Version 20.11, das aus Bibliotheken zur Beschleunigung von Paket-verarbeitungs-Workload besteht, die auf einer Vielzahl von CPU-Architekturen laufen. Es ermöglicht die Entwicklung von Hochgeschwindigkeits-Datenpaket-Netzwerkanwendungen durch Kernel Bypässe. Mit einer Reihe von Bibliotheken ist es möglich Netzwerkschnittstellen-Controller-Treiber für den Polling-Modus zu verwenden, um die Paketverarbeitung statt im Betriebssystem-Kernel auf Anwendungen im Userspace zu ermöglichen. Durch den Einsatz von DPDK kann ein höherer Paketdurchsatz erreicht werden, als mit der Linux Nativen interruptgesteuerten Verarbeitung im Kernel. Dies ist für dieses Projekt wichtig, da hohe Übertragungsraten erreicht werden sollen.

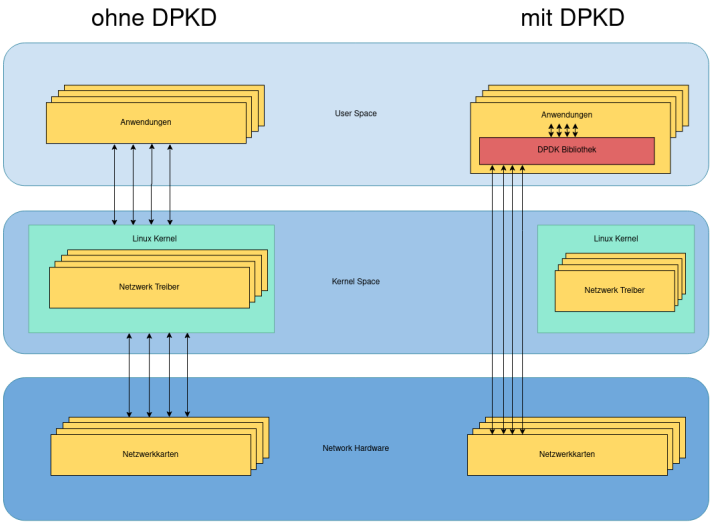

Abbildung 9.1: Kernel Bypass mit DPDK gegenüber Kernel Verarbeitung

9.3 Entwicklungswerkzeuge

Das Buildsystem wird mit Meson erstellt. Meson ist ein Open-Source-Build-System, das sowohl extrem schnell als auch benutzerfreundlich ist (im Vergleich zu CMake). Meson unterstützt viele Plattformen und nutzt Ninja für extrem schnelle vollständige und inkrementelle Builds ohne Einbußen bei der Korrektheit. Dazu gehören unterschiedliche Funktionen wie Unit-Tests, Code-Coverage-Reporting, vorkompilierte Header und dergleichen.

9.4 Tools

Für die Dokumentation wird Doxygen und Latex verwendet und für die Visualisierung von Grafiken VisualParadigm und Inkscape.

Kapitel 10

Ergebnisse der Machbarkeitsanalysen und Beispielrealisierungen

Zur Machbarkeitsanalyse wurde zunächst die Testumgebung vorbereitet. Diese besteht aus drei Rechnern: Einem Angreifer, einem Server, der das zu schützende Netzwerk repräsentiert und der Mitigation-Box selbst. Auf allen Computern wurde Ubuntu 20.04 LTS installiert. Sie sind zum Ersten untereinander über extra eingebaute Netzwerkkarten verbunden, zum Zweiten besteht über die interne Netzwerkkarte jedes Computers Kontakt zum Internet, sodass man via SSH auf die Computer zugreifen kann. Zum jetzigen Zeitpunkt ist die Testumgebung bereits vollständig in einem Labor des Fachgebietes „Telematik/Rechnernetze" aufgebaut. Eine Illustrierung ist bei 8.2 auf Seite 30 zu finden.

Der Server ist über eine 10Gbps-Leitung mit der Mitigation-Box verbunden. Zwischen Mitigation Box und Angreifer besteht eine 25Gbps-Verbindung. Um legitimen Datenverkehr zu simulieren, besteht zwischen zwei Netzwerkkarten-Ports des Angreifers eine Verbindung mit einer maximalen Datenrate von 10Gbps. Von dem Computer auf dem der angreifende Prozess läuft werden also legitime Pakete mit einem Prozess gesendet, der nicht der angreifende ist. Diese gelangen über eine Eigenschleife von einer Netzwerkkarte zu einer anderen. Auf das zweite Interface hat das Angreiferprogramm Zugriff, welches den legitimen dann zusammen mit dem bösartigen Verkehr zu der Mitigation Box weiterleitet. Laut Pflichtenheft soll die Mitigation-Box mit bis zu 20-25Gbps eingehenden Traffic (von Seiten des Internets und des Angreifers) umgehen können. Von der Mitigation Box zum Server soll eine 10Gbps-Verbindung bestehen.

Zumindest im Nicht-Angriffsfall sollen Pakete mit insgesamt 10Gbps zum Server möglichst ohne Verzögerung weitergeleitet werden. Um zu beurteilen ob das realisierbar ist, kann man sich die Benchmarks [9], [10] für Mellanox-Karten bzw. [11] für Intel-Karten ansehen.

In [10] ist ein Test beschrieben, der die Leistung einer Mellanox ConnectX-5 25GbE beschreibt. Diese Netzwerkkarte wird in der Mitigation-Box für das empfangen und senden der Pakete genutzt. Laut Test 2 erreicht das Interface eine Frame Rate von 74,4Mpps (Million packets per second) bei einer Frame Size von 64Byte. Umgerechnet in Gbps sind das $74,4 \cdot 10^6 Pakete \cdot 64 \cdot 8bit = 38092 Mbps = 38,092 Gbps$. In [10] wurde ein Prozessor mit 24 Kernen und einer Taktfrequenz

von 2,70GHz verwendet. In der Mitigation-Box ist ein AMD Ryzen 9 5900X verbaut, welcher nach [9] 12 Kerne (halb so viel) und eine Basistaktfrequenz von 3,7GHz (um Faktor 1,3 schneller) beinhaltet. Angenommen, es wird die gleiche Anzahl von Queues auf der Netzwerkkarte, wie bei [10] (4 pro Port), verwenden, dann wäre bei dem gleichen Test einen Durchsatz von $38Gbps \cdot 0,5 \cdot 1,3 = 24,7Gbps$ erzielbar. Bei Test 2 wurde das weiterleiten von Paketen getestet. Angenommen, es wird zwischen Angriffsfall und Nicht-Angriffsfall unterschieden und in ersteren werden die Pakete in jedem Fall weiter geleitet und werten parallel die Daten aus um einen Angriffsfall zu erkennen, dann ist der erforderte Durchsatz von 10Gbps realistisch. Befände sich das System dann aber im Angriffsfall, sodass nahezu jedes Paket untersucht werden müsste, ist es noch unklar, ob eine solch hohe Datenrate erreicht werden kann. Ein Teil der Abwehrmaßnahmen besteht daraus, Headerinformationen eines Paketes auszulesen und auf deren Basis das Paket zu löschen oder weiterzuleiten. Ein aufwändigeres Verfahren ist die Abwehr von SYN-Flood-Angriffen. Rechnet man mit einer Halbierung bis Drittelung der Datenrate, so ist ausgehender Verkehr von 10Gbps in Richtung des Servers bei von außen kommender Datenrate von 25Gbps gerade noch denkbar, das wird dabei allerdings stark von der Implementierung abhängen.

In dem System werden Datenstrukturen benötigt, um z.B. Headerinformationen abzuspeichern oder um Sequenznummern zu mappen. Da das System eine möglichst geringe Verzögerung des Datenverkehrs zu erreichen versucht, müssen sehr effiziente Strukturen verwendet werden. Hierzu werden Dictionaries bzw. `unordered_maps` verwendet, da diese mit Hash-Verfahren eine sehr gute Zugriffs- und Einfügzeit bieten. `unordered_maps` sind um einiges schneller als maps, da diese keine Ordnung von Schlüsseln benötigen, was in diesem System nicht notwendig ist. Die Standard-Biblihotek stellt die `std::unordered_maps` zur Verfügung, welche in diesem Projekt jedoch nicht verwendet wird, da es außerhalb der Standardbibliothek effizientere bzw. schnellere `unordered_maps` gibt. Andere, wie z.B. `Robin_Hood_unordered_map` und Patchmap bieten aufgrund schnellerer, bzw. besserer Hash-Verfahren eine sehr gute alternative zur `std::unordered_map`. Siehe hierzu die Benchmarks für `Robin:Hood_unordered_mas` [12] und patchmaps [13]. Ein weiterer Vorteil dieser Maps ist, dass diese relativ ähnlich wie `std::unordered_maps` zu verwenden sind, und nur über ein Header-File eingebunden werden müssen.

Kapitel 11

Testdrehbuch

Wann immer Software entwickelt wird, ist es notwendig zu überprüfen, ob diese Software wie geplant funktioniert. Im Verlauf der Implementierungsphase werden allgemeine Tests durchgeführt, welche die grundlegene Funktionen auf Funktionstüchtigkeit überprüfen. Diese Tests sollten so gestaltet sein, dass sie einzelne Teile des Programms überprüfen. Dadurch kann sichergestellt werden, dass diese Elemente den Belastungen gewachsen sind.

In der Validierungsphase werden die meisten Tests durchgeführt. Diese konzentrieren sich auf das Finden von Fehlern und das Ermitteln von optimierbaren Komponenten. Falls in dieser Phase aber noch grundlegene Fehler gefunden und behoben werden, müssen auch die Tests aus der Implementierungsphase wiederholt werden.

11.1 Wichtige Testfälle

Es ist geplanen den Großteil der nichtfunktionalen und einzelne der funktionalen Anforderungen mit Tests zu überprüfen. Dabei wird mit der grundlegenden Funktionen der Mitigation-Box begonnen, zu den komplexeren vortgeschritten, bevor die gewünschten Leistungsparametern überprüft werden.

Zuerst muss das System in der Lage sein, Pakete zu empfangen und weiterzuleiten. Danach wird die Analysefähigkeit der Mitigation-Box getestet, indem überprüft wird, ob sie erkennen kann, dass sie angegriffen wird und dann auch zwischen den einzelnen Angriffsarten unterscheiden kann.

Parallel finden Tests statt, wie viel Datenverkehr der Server verarbeiten kann und welche Attacke diesen wie schnell zum Betriebsausfall bringt.

Im Folgenden soll überprüft werden, ob die Mitigation-Box die verschiedenen (D)DoS-Varianten einzeln, sowie gemischt und verkettet abwehren kann. Insbesondere sollen die sämtliche SYN-Paket basierten Attaken vollständig abgewehrt werden.

Da die Mitigation-Box den Betrieb des zu schützenden Systems nicht einschränken soll, wird auch überprüft, wie stark die zu entwickelnde Software den Datenverkehr verlangsamt und wie viele legitime Pakete sie verwirft.

Zum Schluss wird getestet, ob das System selbst anfällig gegen (D)DoS-Attacken ist. Außerdem wird sein Leistungslimit in Hinsicht auf Daten- und Paketrate geprüft.

11.2 Testplanung

11.2.1 Test 1: Paketweiterleitung

Zunächst wird das simple Weiterleiten von Paketen getestet. Dafür werden Pakete mit DPDK von einem Port der Netzwerkkarte entgegengenommen und auf den anderen Port weitergegeben. Danach wird begonnen, einzelne Ping-Anfragen vom äußeren System über die Mitigation-Box zum Server laufen zu lassen. Darauf aufbauend wird ein erster kleiner Lasttest durchgeführt. Dabei wird möglichst viel Traffic an den Server gesendet. Dieser Teil des Tests gilt als erfolgreich, wenn am Server die Netzwerkkarte ausgelastet ist oder der Server aufgrund von zu hoher Datenlast abstürzt. Währenddessen darf die Mitigation-Box selbst nicht ausfallen. Die Auslastung der Netzwerkkarte lässt sich serverseitig mit dem vorinstallierten System-Monitor überprüfen.

Der Erfolg in beiden Teiltests ist Voraussetzung für alle weiteren Tests.

11.2.2 Test 2: Lasttest Server

Nachdem Test 1 erfolgreich abgeschlossen wurde, kann damit begonnen werden, Angriffe zu erzeugen und auszuloten wie viel der Test-Server verkraften kann. Dies wird über den Verlauf des Projekts mehrfach wiederholt, um das System mit komplexeren und potenteren Angriffen konfrontieren zu können. So wird auch ein Vergleichsmaß erzeugt, mit dem die Effektivität der Abwehrmaßnahmen abgeschätzt werden kann.

11.2.3 Test 3: (D)DoS Erkennung

Parallel zu Test 2 wird getestet, ob das System die verschieden (D)DoS-Angriffe erkennen kann. Hierfür wird zu Anfang ein Strom legitimen Verkehrs etabliert und später (D)DoS-Pakete beigemischt. Es wird mit simplen Attacken wie Syn-Flood gestartet. Später werden die Attacken um komplexere Angriffe erweitert.

Dieser Test gilt als erfolgreich, wenn die Mitigation-Box alle Angriffe erkennen kann. Er dient dabei auch der Schärfung der Entscheidungsgrenzen, ab wann ein Angriff als wie gefährlich eingestuft wird und folglich abgeschwächt werden muss.

In einem zweiten Schritt wird das System mit mehreren parallelen und sich abwechselnden Strategien angegriffen. Dabei verzeichnet der Angreifer die einzelnen Attacken in einer log-Datei. Sobald die Mitigation-Box einen Angriff feststellt, trägt sie diesen in ihre log-Datei ein. Beide log-Dateien werden zur Abschätzung der Fehlerrate miteinander verglichen.

11.2.4 Test 4: (D)DoS Abwehr

Dieser Test setzt den Abschluss von Test 3 voraus. Hier wird die Effektivität der Abwehrmaßnahmen getestet. Beginnend mit den einzelnen Attacken, wird die Wirksamkeit der Verteidigungsmaßnahmen getestet.

Dabei gibt es zwei Maßzahlen für die Effektivität: Einerseits die Responsivität der Mitigation-Box und andererseits der herausgefilterte Anteil an schädlichen Paketen. Letzteres lässt sich durch Vergleichen von Sende- und Empfangslogs des Servers und Angreifers überprüfen. Hingegen

wird die Responsivität nur am legitimen Sender geprüft. Dieser wird ein log führen, in welchem aufgeschrieben wird, zu welchem Zeitpunkt ein Paket gesendet wird und wann die Antwort eingeht.

In späteren Iterationen dieses Test soll der Server mit mehreren parallelen Angriffen konfrontiert werden.

Dieser Test ist nicht als Stresstest konzipiert, sondern lediglich zum Überprüfen der Abwehrmaßnahmen gedacht. Es sollen Fragen geklärt werden wie: Sind die Maßnahmen effektiv genug? Müssen höhere Verluste an legitimen Paketen in kauf genommen werden? Kann mehr Verkehr durchgelassen werden, ohne die Verfügbarkeit des Servers zu gefährden?

11.2.5 Test 5: Transparenz

In diesem Test soll der Einfluss der Mitigation-Box auf legitimen Verkehr getestet werden. Es gibt zweierlei Arten, wie das System auf diese Verbindungen einwirken kann: Einerseits, indem es Pakete verzögert, bis sie als legitim deklariert wurden, und andererseits, indem legitime Pakete als bösartig deklariert und gelöscht werden.

Ersteres kann nur im Leerlauffall, d.h. ohne beigemischte (D)DoS-Pakete, gemessen werden. Da es nicht möglich ist, verlässliche RTTs bei laufendem (D)DoS-Angriff auf einen ungeschützten Server zu messen. Trotzdem wird versucht, die RTT während aller Durchläufe dieses Tests zu messen. Dafür wird ein Anfragen-Pool bestimmt, dessen Anfragen in jedem Test abgesendet werden. Diese Anfragen werden zusammen mit der Wartezeit auf die Antwort in einer log-Datei verzeichnet. Ein Vergleich der unterschiedlichen log-Dateien ergibt die Verzögerung. Die Wegwerfrate ergibt sich aus dem Anteil der Anfragen, die keine oder keine vollständige Antwort erhalten.

Dieser Test wird mehrfach mit unterschiedlichen Angriffslasten, aber konstanter Nutzlast durchgeführt. Der erste Testlauf wird ohne Angriffslast und ohne die Mitigation-Box durchgeführt um Vergleichswerte zu ermitteln. Danach werden mehrere Iterationen mit dem System zwischen Angreifer und Server folgen. Bei diesen Testläufen wird die Angriffslast von 0 Gbit/s schrittweise auf 20 Gbit/s erhöht.

11.2.6 Test 6: Eigensicherheit

In diesem Test wird das Ziel sämtlicher Angriffe direkt oder indirekt die Mitigation-Box selbst sein.

Dafür findet dieser Test in zwei Teilen statt. Im ersten werden die (D)DoS Angriffe auf die Mitigation-Box und nicht auf den Server gerichtet sein. Im zweiten Teil werden die (D)DoS-Angriffe so gestaltet, dass sie maximalen Arbeitsaufwand im System benötigen.

In diese Angriffe soll auch Wissen über Implementierungsdetails einfließen. Alles ist erlaubt, solange es nicht den Link zur Mitigation-Box überlastet.

Das System soll trotz allem in der Lage sein, legitimen Datenverkehr zu ermöglichen. Der Eingang von Antworten am legitimen Sender ist hierbei Erfolgskriterium.

11.2.7 Test 7: Paketflut

Zuletzt soll die maximal verarbeitbare Paketrate getestet werden. Hiefür wird das System mit einer steigenden Rate von SYN, SYN-FIN und SYN-FIN-ACK Paketen angegriffen. Dabei werden

nicht nur für den Angriff, sondern auch für den legitimen Verkehr ausschließlich kleine TCP-Pakete, die Verbindungen auf- und abbauen, verwendet, um die Paketrate zu maximieren.

Die maximale Paketrate gilt als überschritten, wenn kein Verbindungsaufbau mehr zustande kommt.

In diesem Test werden TCP-SYN-Pakete verwendet, obwohl es für die Mitigation-Box aufwendigere Angriffe gibt. Dies ist eine Idealisierung, um ein absolutes Maximum an Paketen zu ermitteln, die das System verarbeiten kann.

Implizit fungiert dieser Test auch als Beweis, dass die verschiedenen TCP-SYN-Angriffe vollständig abgewehrt werden können.

11.2.8 Test 8: Datenrate

Dieser Test ist kein richtiger Test, denn er wird in jedem Test durchgeführt, in dem diesen Bestimmungen nicht ausdrücklich widersprochen wird.
Die Soll-Datenrate für legitimen, zum Server gesendeten Verkehr liegt bei 5 Gbit/s.

Der Sollwert für die Datenrate von Angriffspaketen liegt bei 20 Gbit/s.

Bei den tatsächlichen Werte können während der Test Schwankungen auftreten . Ursache wird z.B. der maximale Durchsatz des Links zwischen Mitigation-Box und Angreifer sein, da dieser Pakete in beide Richtungen leiten muss.

11.3 Tabellarische Übersicht

Test	**Name**	**Kurzbeschreibung**	**getestete Anforderungen**
Nr 1	Paketweiterleitung	reines weiterleiten von Paketen	F07
Nr 2	(D)DoS-Angriffe	Effektivität von (D)DoS-Angriffen testen	
Nr 3	(D)DoS Erkennung	Kalibrierung von (D)DoS Erkennung	F05
Nr 4	(D)DoS Abwehr	Überprüfung Abwehrmaßnahmen	F03, F09
Nr 5	Transparenz	Analyse Effekt auf Verbindungen	NF02, NF05, NF07, NF09
Nr 6	Eigensicherheit	Überprüfung Widerstandsfähigkeit der Miditation-Box	F02
Nr 7	Paketrate	Ermittlung maximaler Paketrate	NF04, NF06
Nr 8	Datenrate	allgemeine Bestimmungen	NF03

Kapitel 12

Abkürzungsverzeichnis

DDoS Distributed Denial-of-Service

DoS Denial-of-Service

DRoS Distributed Reflected Denial-of-Service

Gbps Giga bit pro sekunde

Mpps Million packets per second

Literaturverzeichnis

[1] infopoint security, "Cyber-angriffe auf deutsche krankenhäuser sind um 220 prozent gestiegen," 2021. Aufgerufen 23.05.2021.

[2] tecchannel, "Trend micro: Latente gefahr durch botnet sdbot." Website, 2009. Aufgerufen 23.05.2021.

[3] NEUSTAR, "2016 ddos report." Website. Aufgerufen 23.05.2021.

[4] datacenterknowledge.com, "Study: Number of costly dos-related data center outages rising," 2016. Aufgerufen 23.05.2021.

[5] ProjectShield, "Projectshield webseite." Website, 2021. Aufgerufen 23.05.2021.

[6] Cloudflare, "Cloudflare ddos protection." Website, 2021. Aufgerufen 23.05.2021.

[7] Amazon, "Aws shield." Website, 2021. Aufgerufen 23.05.2021.

[8] M. Z. et al., "Mitigating volumetric ddos attacks with programmable switches." Website, 2000. Aufgerufen 11.05.2021.

[9] "Ryzen website." Website. Zugriff: 2021-05-17.

[10] "Mellanox nic's performance report with dpdk 20.05." Website. Zugriff: 2021-05-17.

[11] "Dpdk intel nic performance reportrelease 20.02." Website. Zugriff: 2021-05-17.

[12] T. Goetghebuer-Planchon, "Benchmark of major hash maps implementations."

[13] W. Brehm, "Memory efficient hash tables and pseudorandom ordering."

Abbildungsverzeichnis