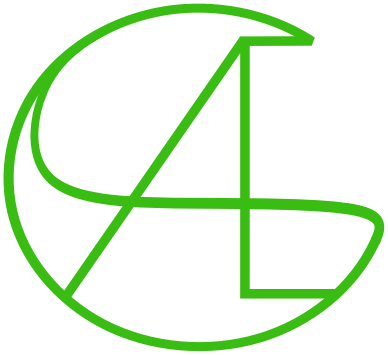
Technische Universität Ilmenau
Fakultät Informatik und Automatisierung
Fachgebiet Telematik/Rechnernetze

TECHNISCHE UNIVERSITÄT
ILMENAU

Zweite Review zum Thema

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

Codename: AEGIS



Autoren:

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

Betreuer: Martin Backhaus

Entwurf 23. Juni 2021

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

Denial-of-Service-Angriffe stellen eine ernstzunehmende Bedrohung dar. Im digitalen Zeitalter sind viele Systeme über das Internet miteinander verbunden. Viele Unternehmen, Krankenhäuser 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 große Schaden bei deutlich leistungsfähigeren Empfängern auslösen. In Botnetzen können die Angriffe von mehreren Computern gleichzeitig koordiniert werden 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

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

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.

1.2 Überblick

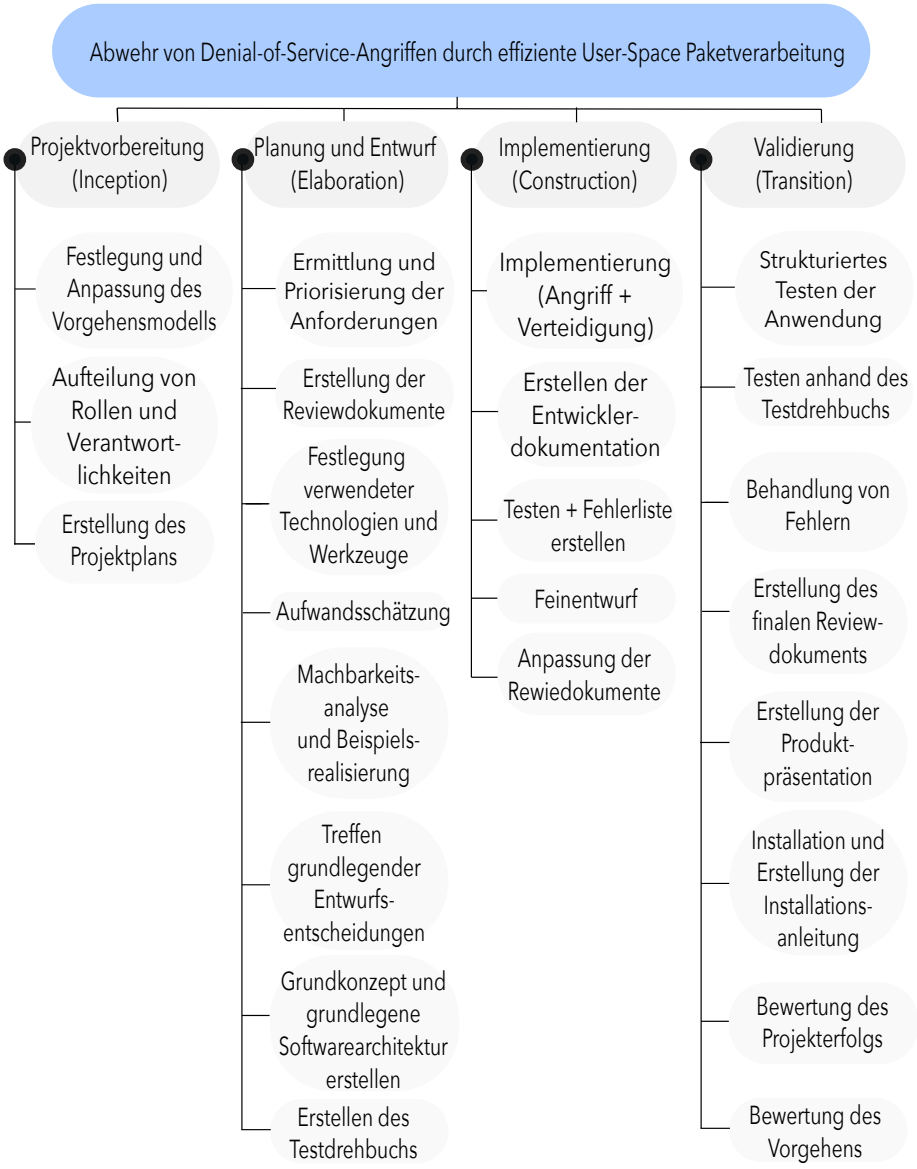


Abbildung 1.1: Projektstrukturplan

Das Softwareprojekt wurde vom zuständigen Fachgebiet in drei Teile aufgeteilt. Die Planungs- und Entwicklungsphase, die Implementierungsphase und die Validierungsphase dauern jeweils einen Monat und werden durch ein Review abgeschlossen. Zu diesen Reviews werden die Ergebnisse der vergangenen Phase vorgestellt und die erforderlichen Review-Dokumente abgegeben.

Zu Beginn des Projekts wurde sich auf den Unified Process als Vorgehensmodell geeinigt, damit sowohl ein gewisser Grad an Flexibilität als auch die Planbarkeit der Ergebnisse gewährleistet werden kann. Prinzipiell besteht dieses Vorgehensmodell aus vier Phasen, von denen die Konzeption und die Ausarbeitung beide in der Planungs- und Entwurfsphase lagen. Die Konstruktionsphase und die Inbetriebnahme decken sich zeitlich mit der Implementierungs- und der Validierungsphase.

Dieses zweite Review-Dokument bezieht sich auf die Implementierungsphase (bzw. Konstruktionsphase bei Verwendung der Begriffe des Unified Processes). Das heißt, dass es auf den Ergebnissen der vorhergehenden Phase und dem ersten Review-Dokument vom 26. Mai 2021 aufbaut.

Das erste Review-Dokument enthält die gängigen Inhalte eines Pflichtenhefts wie die funktionalen und nichtfunktionalen Anforderungen, eine Aufwands- und Risikoanalyse und Überlegungen zum Vorgehen und der internen Organisation. Außerdem umfasste es eine Entwurfsdokumentation für den Grobentwurf, die Anforderungsanalyse, ein Kapitel zu den Technologien und Entwicklungswerkzeugen, Ergebnisse zu den Machbarkeitsanalysen und Beispielrealisierungen und ein Testdrehbuch.

Im Kapitel zum Grobentwurf werden nun zusätzlich zur erneuten Erläuterung der grundlegenden Architektur die für den Unified Process üblichen Überarbeitungen des Grobentwurfs dargestellt und begründet. Dabei wird, genauso wie beim darauffolgenden Feinentwurf, Paket für Paket vorgegangen. Schließlich werden in einem Bug-Review die offenen Anforderungen und Fehler beschrieben und die mittels des Tools Kimai erfassten Arbeitszeiten ausgewertet.

Es bleibt anzumerken, dass einige Teile dieses Dokuments wie die Problemstellung oder der ursprüngliche Grobentwurf dem ersten Review-Dokument entnommen sind, weil dies vom Fachgebiet empfohlen wurde und dadurch die Veränderungen besonders gut dargestellt werden können.

Die Erstellung dieses Review-Dokuments stellt allerdings nur einen Teil der in dieser Phase erledigten Aufgaben dar. Hauptsächlich ging es um die Implementierung des bisher geplanten Systems unter Berücksichtigung aller Muss-Anforderungen, aber auch um die Erstellung einer Entwicklungsdokumentation mittels Doxygen. Im Projektstrukturplan in Abb. 1.1 lässt sich das und auch die Aufgaben der letzten Phase, der Validierungsphase, gut erkennen.

Kapitel 2

Grobentwurf

Dieses Kapitel behandelt zunächst den Grobentwurf, wie er in der Planungs- und Entwurfsphase des Projekts erarbeitet wurde. Schließlich wird auf dessen Überarbeitung und dazugehörige Diagramme eingegangen.

2.1 Grundlegende Architektur

In folgendem Unterkapitel werden die grundlegenden Entscheidungen des Entwurfs erklärt und durch die Rahmenbedingungen begründet. Ein intuitiver Einstieg soll schrittweise an das System heranführen über Erklärung des Netzwerkaufbaus, dem grundlegenden Aufbau der Software, dem Kontrollfluss eines Pakets und verwendeter Verfahren.

2.1.1 Netzwerkaufbau

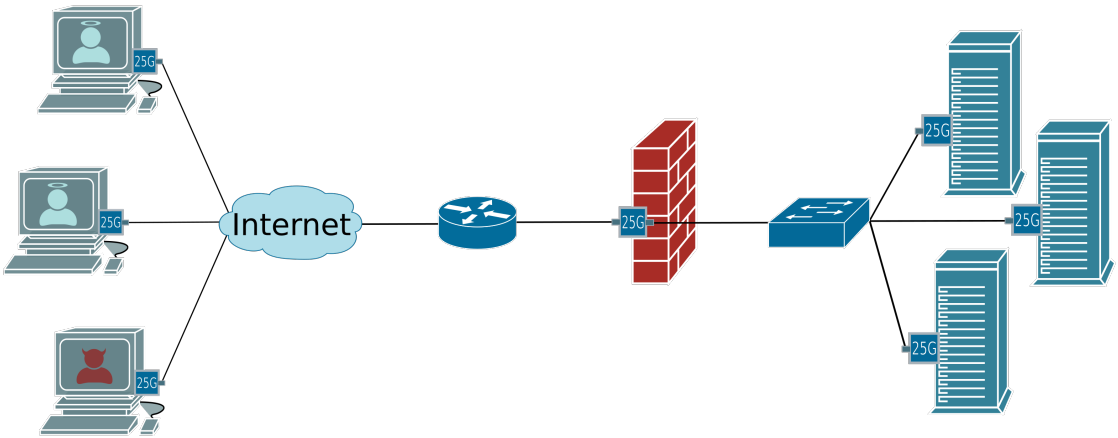


Abbildung 2.1: Realaufbau unter Verwendung eines Angreifers

Die Abbildung 2.1 zeigen den typischen, zu erwartenden Netzwerkaufbau, welcher in dieser Form im Internet und in der Produktivumgebung vorkommt. 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.

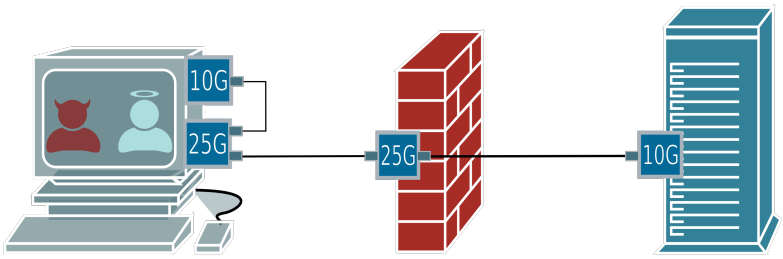


Abbildung 2.2: Versuchsaufbau

Der für das Projekt verwendete Versuchsaufbau untergliedert sich ebenfalls in drei Teile, auch hier beginnt die Darstellung 2.2 ganz links mit dem System, welches Angreifer und legitimen Nutzer in sich vereint. Um 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 demselben 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, das sich in der Abbildung 2.2 auf der rechten Seite befindert, 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 erfolgrei-

cher 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.

2.1.2 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 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 auf den Kontrollfluss innerhalb des Programms, auf den Einsatz von parallelen Threads und auf die einzelnen Komponenten näher eingegangen.

2.1.2.1 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 insgesamt zu verringern.

Dabei stellt sich die Frage, wozu die Threads im Programm genau zuständig sind. 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.

2.1.2.2 Kontrollfluss eines Paketes

In diesem Abschnitt soll veranschaulicht werden, wie die Behandlung eines Paketes vom NicManagement 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 2.3 am Ende der Sektion 2.1.2 zu finden.

2.1.2.3 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 2.4: 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. Mit „Port" im Folgenden entweder der physische Steckplatz einer Netzwerkkarte gemeint oder jener Teil der Netzwerkadresse, die eine Zuordnung zu einem bestimmten Prozess bewirkt. Die Bedeutung erschließt sich aus dem Kontext.

Ferner besteht die Möglichkeit, Symmetric RSS einzusetzen. Dieser Mechanismus sorgt dafür, dass die Pakete, die auf dem einen Port der Netzwerkkarte ankommen nach genau der selben Zuordnung auf dessen RX-Queues aufgeteilt werden, wie die auf dem anderen Port ankommenden Pakete auf dessen RX-Queues. Dabei ist die Zuordnung auf dem einen Port „symmetrisch" zu der auf dem anderen Port. Das heißt, wenn bei Port 0 ein Paket mit `Src-IP: a, Dst-IP: b, Src-Port: x, Dst-Port: y` ankommt, wird es genauso dessen RX-Queues zugeteilt, wie ein Paket mit `Src-IP: b, Dst-IP: a, Src-Port: y, Dst-Port: x` auf RX-Queues von Port 1. 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.

Neben RX-Queues existieren auch TX-Queues (Transmit-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. An dieser Stelle soll auch kurz auf eine Besonderheit von DPDK eingegangen werden: Im Linux-Kernel empfängt ein Programm Pakete durch Interrupt-Handling. Gegensätzlich dazu werden bei DPDK alle empfangenen Pakete, die sich derzeit in den RX-Queues der Netzwerkkarte befinden auf einmal von der Anwendung gepollt. In der zu entwickelnden Software geschieht dieses Polling durch den einzelnen Thread stets zu Beginn eines Pipeline-Durchlaufes.

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, weil das Tupel aus besagten Adressen der Schlüssel der RSS-Hash-Funktion sind.

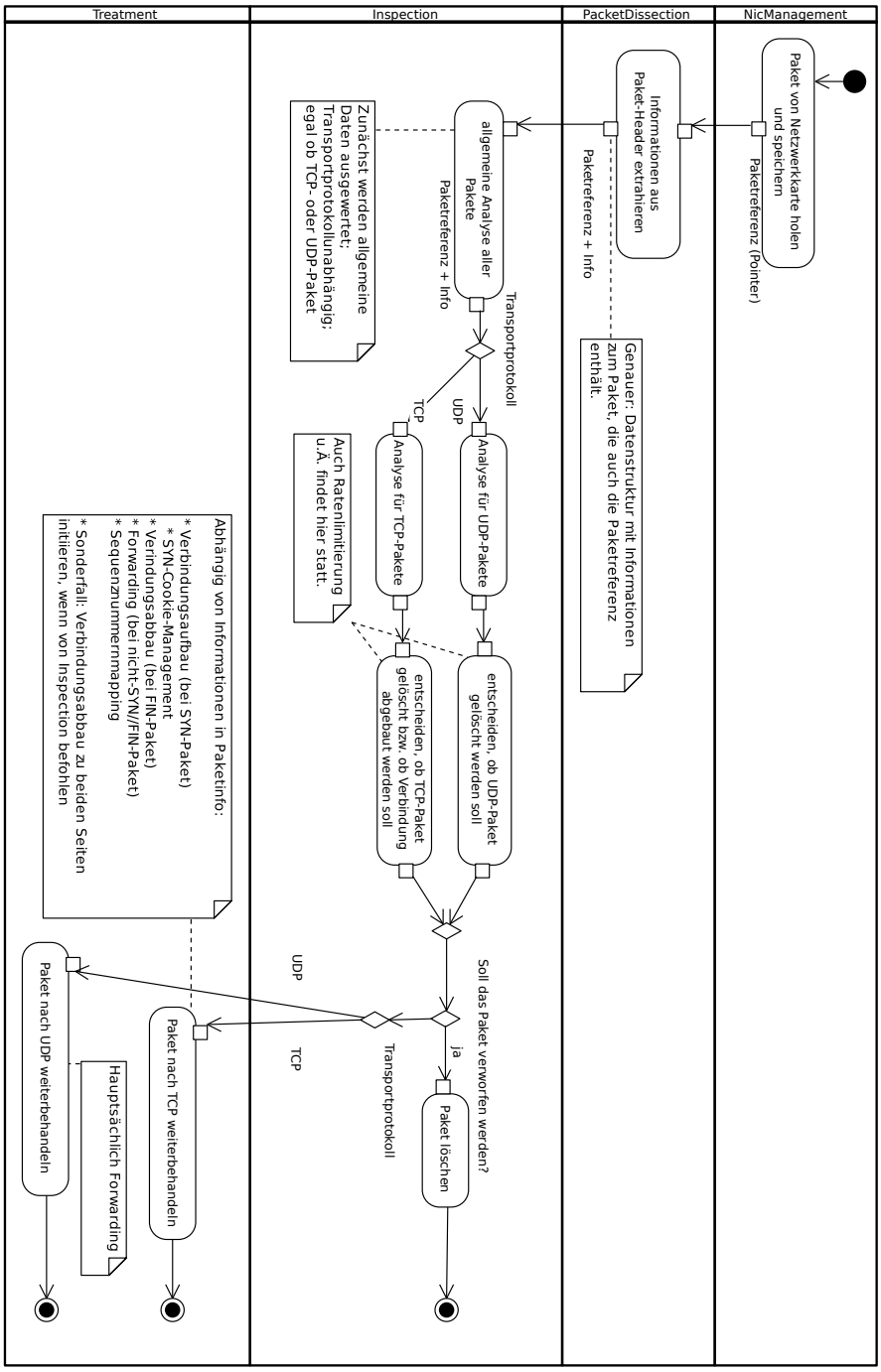


Abbildung 2.3: Schematische Darstellung des Kontrollflusses

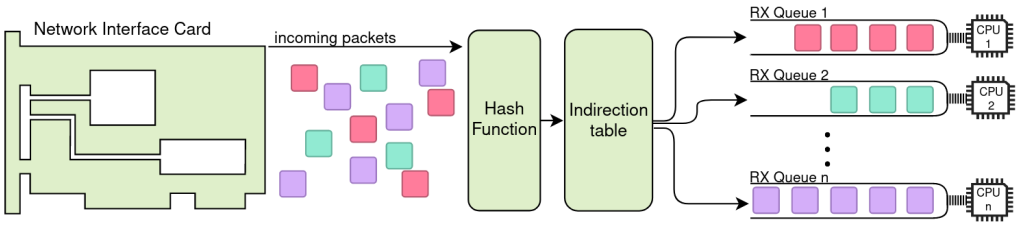


Abbildung 2.4: Beispielhafte Paketverarbeitung mit Receive Side Scaling

2.2 Überarbeiteter Grobentwurf

Die in diesem Abschnitt erläuterten Änderungen wurden im Laufe der Implementierungsphase vorgenommen. Für das bei diesem Softwareprojekt genutzte Vorgehensmodell des Unified Process ist es typisch, dass sich auch während der Implementierung Änderungen am Entwurf ergeben. Für die Teammitglieder ist es besonders aufgrund der geringen Erfahrung bezüglich der Thematik des Projekts unerlässlich, wichtige Verbesserungen direkt vornehmen zu können.

2.2.1 Paketdiagramm

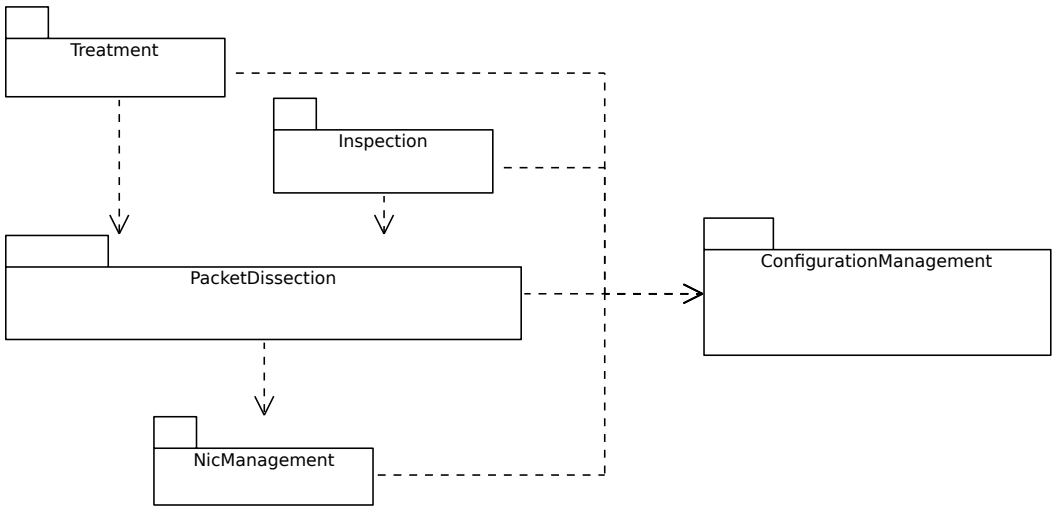


Abbildung 2.5: Paketdiagramm

Grundsätzlich ist es angedacht, wie im Paketdiagramm 2.5 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. Diese Komponente beinhaltet Klassen zur Paketrepräsentation für das Treatment und die Inspection. Sie liefert Operationen zum Löschen, Senden, Empfangen und Bearbeiten von Paketen. In der PacketDissection werden auch Informationen aus den einzelnen Headern eines Netzwerkpakets extrahiert.

Die extrahierten 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 für die Abwehrwehrmaßnahmen der verschiedenen Angriffe zuständig ist, verwendet hierzu die von der Inspection bereitgestellten Ergebnisse und Informationen. Für das Versenden und Verwerfen von Pakten, sowie den Aufbau und das Terminieren von Verbindungen, verwendet das Treatment die PacketDissection, welche die Anweisung an das NicManagement weitergibt.

Sowohl Treatment, als auch Inspection und PacketDissection verwenden das ConfigurationManagement, welches Parameter für die Programmbestandteile in Form von Konfigurationsdateien

vorhält. Das ConfigurationManagement bietet die Möglichkeit für den Nutzer, aktiv Einstellungen am System vorzunehmen.

2.2.2 NicManagement

Das NicManagement übernimmt wie im letzten Review-Dokument erwähnt das Senden, das Pollen und das Löschen von Paketen. Das Paket wurde eingeführt, um bestimmte Funktionen und Initialisierungsschritte vom DPDK zu kapseln. Es hat sich allerdings herausgestellt, dass die Operationen „Senden", „Empfangen" und „Löschen" in der Implementierung sehr wenig Aufwand bereiten. Das Zusammenbauen von Paketen wird von der Komponente PacketDissection übernommen. Der aufwändigere Teil ist die Initialisierung des DPDK, insbesondere die Ermöglichung von Multithreading und die Konfigurierung von symmetric Receive-Side-Scaling. Die dazu notwendigen Schritte werden jedoch von Initializer bzw in der main.cpp-Datei vor dem Starten der einzelnen Threads durchgeführt und sind nicht mehr Teil des NicManagements.

Aus diesem Grund und weil jeder nicht notwendige Funktionsaufruf Rechenzeit kostet könnte das NicManagement aufgelöst und die bereitgestellten Funktionen an anderer Stelle implementiert werden. Die einzige Klasse, die das NicManagement zum jetzigen Zeitpunkt verwendet ist die PacketContainer-Klasse in der Komponente PacketDissection. Es wäre möglich, den Inhalt der NicManagement-Aufgaben in diese Klasse zu verschieben.

2.2.3 ConfigurationManagement

Das Paket „ConfigurationManagement " kümmert sich um die Initialisierung der Software und desweiteren werden hier die ablaufenden Threads konfiguriert und verwaltet. Grundlegend ist das Paket in drei Klassen eingeteilt, Configurator, Initializer und Thread.

Die Klasse „Configurator" bietet eine Schnittstelle zu der Konfigurationsdatei, welche im Projekt liegt und die grundelegenden Einstellungen der Software enthält. An anderer Stelle kann dann über verschiedene Methoden auf Konfigurationsinformationen zugegriffen werden.

Die Klasse „Initializer" dient dazu die für die Bibliothek DPDK notwendigen Vorrausetzungen zu schaffen.

Die Klasse „Thread" enthält den Ablaufplan für die Workerthreads des Systems.

2.2.4 PacketDissection

Der Zweck dieses Pakets ist, sämtliche Daten, die Analyser und Treatment für ihre Arbeit brauchen, aus den Paketen zu extrahieren.

Dafür war geplant, in dieser Komponente die Repräsentation eines Paketes - die Klasse „PacketInfo" - unterzubringen. Jedes Paket sollte einzeln repräsentiert durch die Pipeline des Programmes gereicht werden. Es hat sich herausgestellt, dass dieses Vorgehen ineffizient ist. Näheres dazu ist im Feinentwurfskapitel beschrieben.

Aus diesem Grund wurde eine neue Klasse namens „PacketContainer" eingeführt. Diese dient als Repräsentation einer Folge von Paketen, die empfangen wurden. Enthalten sind sowohl die Pointer auf die tatsächlichen Pakete als auch Metadaten in Form mehrerer Objekte der PacketInfo-Klasse. Auf dem PacketContainer ist es möglich, Pakete zu entnehmen, hinzuzufügen und zu löschen. Weiterhin gibt es jeweils eine Methode zum pollen neuer Pakete und zum senden aller vorhandener Pakete.

Die PaketInfo Klasse stellt immernoch alle relevanten Header-Informationen eines Paketes zur Verfügung. Allerdings werden Informationen nur noch auf Abruf extrahiert. Hierbei werden für die IP Versionen 4 und 6, sowie die Layer 4 Protokolle TCP, UDP und ICMP unterstützt. Darüber hinaus soll sie auch das verändern einzelner Informationen im Header ermöglichen.

Die letzte Klasse in der PacketDissection ist der namensgebende HeaderExtractor. Seine Aufgabe wandelte sich vom Extrahieren der Informationen zum Vorbereiten des Extrahieren auf Bedarf.

2.2.5 Inspection

Die zuvor globale Auswertung von Angriffen aller Threads durch eine einzige Instanz wurde ersetzt durch eine lokale threadeigene Auswertung. Berechnete Zahlen und Statistiken wie Paketrate und Angriffsrate werden per Interthreadkommunikation nur noch an eine globale Statistikinstanz gesendet. Dadurch können die Threads unabhängig voneinander agieren und reagieren, die implementation der Methoden ist deutlich einfacher ausgefallen und die Interthreadkommunikation konnte auf ein einwegiges minimum begrenzt werden was der Auswertungsgeschwindigkeit jedes Inspection-Threads zugute kommt.

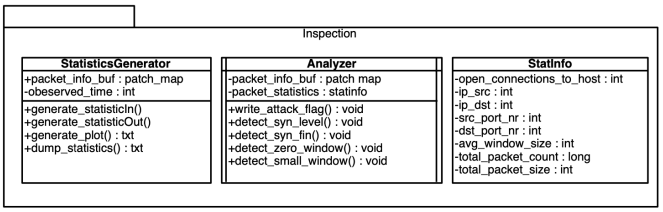


Abbildung 2.6: Altes Klassendiagramm der Inspection aus der Implementierungsphase

Durch den Einsatz von symetric Receive Side Scaling ist sowohl die Auslastung jeder Inspektion ausgeglichen und zusätzlich werden gleiche Paketströme (selbe Paketquelle und -Empfänger) durch denselben Thread verarbeitet. Dies erleichtert die Erkennung legitimer Pakete, da diese über eine eigene Patchmap für bestimmte Fälle von großteilig illegitimen Verkehr unterscheidbar ist und die Variationen geringer sind.

Die Statistik wird statt durch eine eigene Klasse direkt in der Inspection erstellt und das Ergebnis an eine globale Statistik Instanz gesendet, um diese an den Nutzer auszugeben. Die Inspection Klasse ist dadurch schlanker und folgt einem linearen Pipelinemodell für Paketverarbeitung.

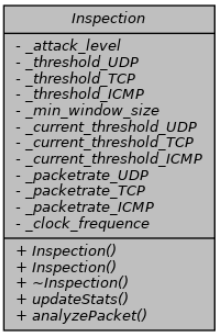


Abbildung 2.7: Aktuelles Klassendiagramm der Inspection aus der Planungs- und Entwurfsphase

2.2.6 Treatment

Treatment
- _timestamp: u_int8_t - _cookie_secret: u_int64_t - _densemap: google:: dense_hash_map<Data, Info, MyHashFunction> - _ackmap: google:: dense_hash_map<Data, PacketInfo*, MyHashFunction> - _packet_to_inside: PacketContainer* - _packet_to_outside: PacketContainer*
+ Treatment (pkt_to_inside: PacketContainer*, pkt_to_outside: PacketContainer*) + treat_packets (): void + calc_cookie_hash (timestamp: u_int8_t, extip: u_int32_t, intip: u_int32_t, extport: u_int16_t, intport: u_int16_t) : u_int32_t + check_syn_cookie (cookie_value: u_int32_t, d: Data) : boolean + create_cookie_secret () : u_int64_t + increment_timestamp () : void

Abbildung 2.8: Aktuelles Klassendiagramm des Treatments aus der Implementierungsphase

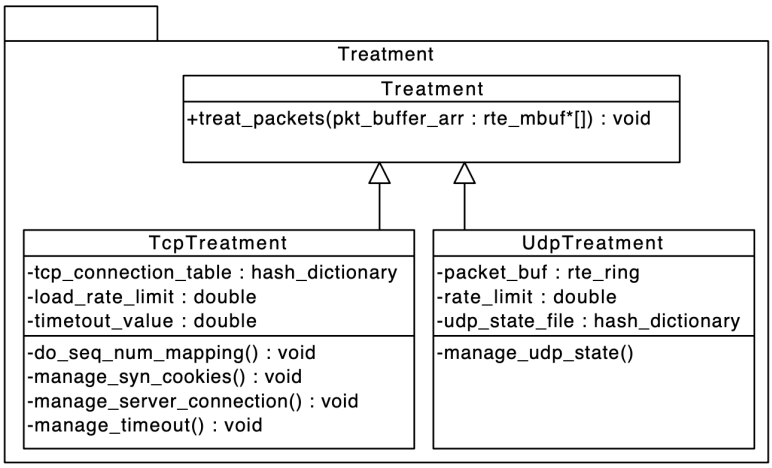


Abbildung 2.9: Altes Paket Treatments mit verschiedenen Klassen aus der Planungs- und Entwurfsphase

Abbildung 2.8 zeigt das während der Implementierungsphase überarbeitete Klassendiagramm. Auf den ersten Blick unterscheidet sich dieses stark vom Grobentwurf des `Treatments` aus der Planungs- und Entwurfsphase (vgl. Abb. 2.9).

Das `Treatment` hat fortan die Aufgabe, die Implementierung von TCP-SYN-Cookies sowie die Realisierung eines TCP-Proxies zu übernehmen. Zur Realisierung des TCP-Proxies gehört insbesondere die Sequenznummernanpassung zwischen internen und externen Verbindungen.

Es fällt auf, dass keine Vererbung mehr verwendet wird. Das heißt, dass nicht mehr zwischen `TcpTreatment` und `UdpTreatment` unterschieden wird. Der Grund hierfür ist die Auslagerung des UDP-Treatments in den `Analyzer` (Paket `Inspection`). Es wird allerdings nicht nur das UDP-Treatment ausgelagert, sondern auch die Behandlung der SYN-FIN-Attacke sowie des TCP-Small- und Zero-Window-Angriffs. Dies ist darin begründet, dass bereits im `Analyzer` alle hierzu benötigten Informationen und Funktionalitäten bereitstehen. Dies führt letztlich dazu, dass Funktionsaufrufe oder function calls reduziert werden, welches es dem Programm ermöglicht,

insgesamt eine bessere Performanz aufzuweisen. Durch den Wegfall der Klasse `UdpTreatment` entfällt die Notwendigkeit der Vererbung und die gesamte Implementierung des Treatments kann in einer einzigen Klasse erfolgen.

Das ursprüngliche Attribut `tcp_connection_table` wurde umbenannt in `_densemap`. Der Unterstrich vor dem Variablennamen zeigt, dass es sich um eine Member-Variable handelt. Durch die Umbennenung wird deutlich, dass es sich um eine Google-Densemap handelt und nicht um eine beliebige Map. Hinzu kommt zusätzlich die `_ackmap`, bei der es sich ebenfalls um eine Densemap handelt. Die ACK-Map hat zur Aufgabe, diejenigen Pakete zwischenzuspeichern, welche im letzten ACK des externen Verbindungsaufbaus am System ankommen und nach erfolgreichem Verbindungaufbau mit dem internen System an ebendieses weitergeleitet werden müssen. Der Wegfall von `load_rate_limit` und `timeout_value` ist ähnlich wie beim `UdpTreatment` durch die Auslagerung in den `Analyzer` zu begründen. Die Variable `_timestamp`, die in der Implementierungsphase hinzugekommen ist, wird benötigt, um das Alter der ACKs, welche als Reaktion auf ein SYN-ACK erhalten werden, zu bestimmen. Das `_cookie_secret` wird im SYN-Cookie verwendet, um es einem potentiellen Angreifer schwieriger zu machen, eine illegitime Verbindung zum System aufzubauen, indem den Cookies ein weiterer schwieriger zu erratender Wert hinzugefügt wird. Bei den Variablen `_packet_to_inside` und `_packet_to_outside` handelt es sich um Pointer zu `PacketContainern`. Diese speichern die dem Treatment im Konstruktor übergebenen `PacketContainer`-Pointer für den weiteren internen Gebrauch.

Um ein Objekt der Klasse `Treatment` zu erzeugen, muss der Konstruktor aufgerufen werden und die beiden Parameter `pkt_to_inside` und `pkt_to_outside` vom Typ `PacketContainer*` übergeben werden.

Die Sequenznummernzuordnung, die ursprünglich in der Methode `do_seq_num_mapping()` vorgenommen werden sollte, ist nun Teil der Methode `treat_packets()`, welche allumfassend für das gesamte Verbindungsmanagement des TCP-Verkehrs zuständig ist. Der Inhalt der Methode `manage_syn_cookies()` wurde mit der Überarbeitung auf verschiedene Methoden aufgeteilt: Der Hash-Wert des TCP-SYN-Cookies wird in der Methode `calc_cookie_hash()` berechnet. Das dazu benötigte Cookie-Secret ist der globale Wert `_cookie_secret`, der durch den Rückgabewert der Methode `create_cookie_secret()` initialisiert wird. Dieser Wert ändert sich während des Ablaufs des Programms nicht. `Check_syn_cookie()` vergleicht den Cookie eines ankommenden, zum Verbindungsaufbau gehörenden ACKs mit dem für diese Verbindung erwarteten Wert. Die Methode `manage_timeout()` wurde aus oben genannten Effizienzgründen und der Zugehörigkeit zur Behandlung der Sockstress-Attacken (TCP-Small- bzw. TCP-Zero-Window) ebenfalls in den `Analyzer` verschoben. Die Methode `manage_server_connection()` wurde mit der Methode `treat_packets()` konsolidiert, um auch hier Funktionsaufrufe einzusparen.

Kapitel 3

Feinentwurf

- falls wir es nicht in ein extra Paket packen, wird hier alles erklärt, was im Root-Ordner liegt. Also Thread (WorkerThread) und das was in main.cpp passiert. Also der Initialisierungskram
- Außerdem: Klassendiagramm der ganzen Software, ohne Methoden, Attribute, nur die Namen der Klassen und die Beziehungen zwischen den Klassen, vielleicht visuell angeordnet nach Paketen (hübsch mit gestrichelten Linien wenn möglich)

- Initialisierung:
  - init_dpdk (wird im Allgemeinen nicht genauer erklärt)
  - symmetric rss
  - Starten von Threads

3.1 NicManagement

- Klassendiaramm, doxygen

3.2 ConfigurationManagement

3.2.1 Configurator

Für die Software gibt es eine Konfigurationsdatei, „config.json", welche innerhalb dieser Klasse eingelesen wird und diese Informationen global zur Verfügung stellt. Von der Klasse soll es im ganzen Programmablauf nur ein Objekt geben, damit keine Inkonsistenzen entstehen können. Aufgrund dieser Anforderungen kann ein spezielles Entwurfsmuster verwendet werden, der Singleton. Der Singleton ist ein Erzeugungsmuster, welches automatisch dafür sorgt, dass nur eine Instanz des Configurator existieren kann, und stellt ähnlich globalen Variablen Informationen global dar. Der Vorteil des Singleton diesen gegenüber besteht darin, dass der Singleton nur dann verwendet wird, wenn er wirklich benötigt wird. Die Klasse des Configurator hat nur einen

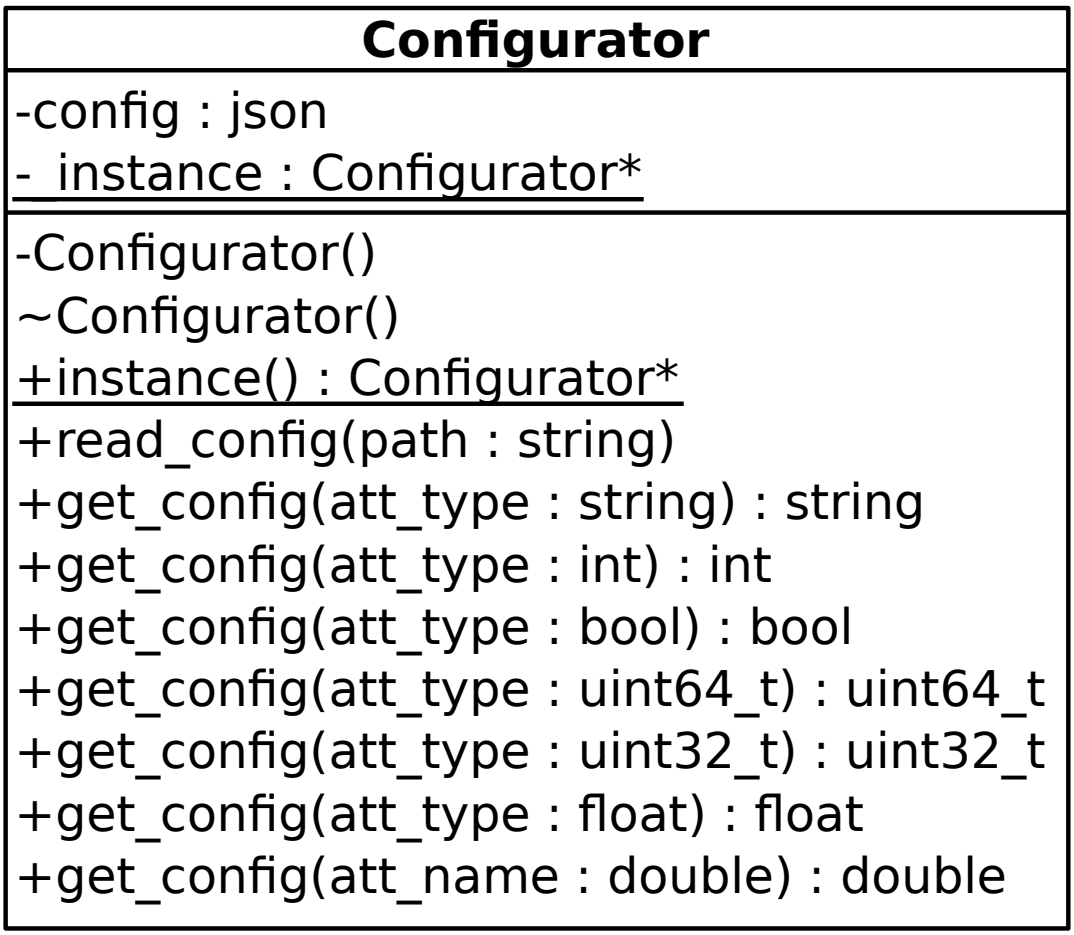


Abbildung 3.1: Klassendiagramm Configurator

privaten Konstruktor, welcher in der zum Singleton gehörigen Methode der Instanziesierung, verwendet wird. In der ersten Verwendung des Configurators wird die Methode, „read_config()", zur Einlesung der Daten ausgeführt.Falls die Configurationsdatei nicht findbar ist, so wird eine Exception geworfen, da die Software ohne diese nicht ablaufen kann. Die ausgelesenen Daten werden dann in einem privaten json-Objekt hinterlegt. Hierzu wichtig zu erwähnen ist, dass nlohmann::json verwendet wird. Die Informationen der json-Datei werden über eine Schnittstelle „get_config(Datentyp)" anderen Klassen zur Verfügung gestellt, wobei es unterschiedliche Methode je nach Datentyp gibt. Der explizite Aufruf des Auslesens erfolgt über die Methode „instance()", mithilfe jener ein Zeiger auf das Configurator-Objekt zurückgegeben wird.

3.2.2 Initializer

Die Klasse Initializer dient dazu, dass grundlegende Initialisierungen für „DPDK" vorgenommen werden, hierzu gibt es die Methode „init_dpdk(int argc, char** argv)".

3.2.3 Thread

Diese Klasse dient dazu, dass parallele Threads erzeugt werden können, welche dann die gesamte Paketbehandlung des Systems durchlaufen. Hierzu werden jedem Thread zwei PacketContainer übergeben. Ein PacketContainer dient zum annehmen von Paketen welche von außerhalb des Netzwerkes in das Netzwerk kommen und der andere analog dazu mit Paketen, welche von innerhalb des Netzwerkes nach außen sollen. Die run-Methode der Thread-klasse besteht daraus, dass eine bestimmte Anzahl an Paketen gepollt wird mittels der Methode „poll_packets(int number)". Dies gilt für beide PacketContainer. Nachdem die Pakete behandelt wurden, was innerhalb dieses Pollings passiert, werden die restlichen Pakete nun in jeweilige Richtung weitergeschickt mittels „send_packets()".

3.2.4 Initializer

Die Klasse Initializer dient dazu, dass grundlegende Initialisierungen für „DPDK" vorgenommen werden, hierzu gibt es die Methode „init_dpdk(int argc, char** argv)".

3.3 PacketDissection

Die Aufgabe der PacketDissection ist es, Informationen über die zu untersuchenden Pakete bereitzustellen. Zusätzlich wird auch die Kommunikation mit dem NicManagement über die PacketDissection geleitet.

Im Diagramm 3.2 wird das Polling von Paketen unter Benutzung des PacketContainers dargestellt. Der PacketContainer fungiert hierbei als zentrales Element, dass den Ablauf steuert.

3.3.1 Thread

Diese Klasse dient dazu, dass parallele Threads erzeugt werden können, welche dann die gesamte Paketbehandlung des Systems durchlaufen. Hierzu werden jedem Thread zwei PacketContainer übergeben. Ein PacketContainer dient zum annehmen von Paketen welche von außerhalb des Netzwerkes in das Netzwerk kommen und der andere analog dazu mit Paketen, welche von innerhalb des Netzwerkes nach außen sollen. Die run-Methode der Thread-klasse besteht daraus, dass eine bestimmte Anzahl an Paketen gepollt wird mittels der Methode „poll_packets(int number)". Dies gilt für beide PacketContainer. Nachdem die Pakete behandelt wurden, was innerhalb dieses Pollings passiert, werden die restlichen Pakete nun in jeweilige Richtung weitergeschickt mittels „send_packets()".

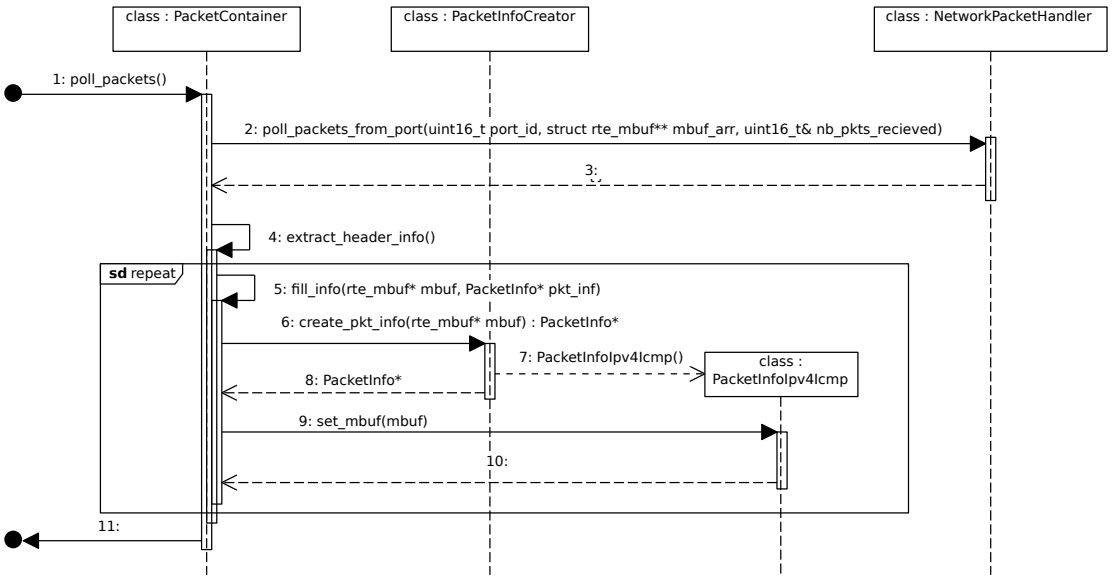


Abbildung 3.2: Sequenzdiagramm zum polling von Paketen über den PacketContainer

3.4 PacketDissection

3.4.1 PacketContainer

DPDK liefert beim Pollen von Paketen ein Array von Pointern auf sogenannte `mbuf`-Strukturen. Auch beim Senden muss dem Framework ein solches Array übergeben werden, denn die `mbuf`-Strukturen repräsentieren Pakete innerhalb von DPDK. Um nur die PacketInfo-Objekte durch das Programm reichen zu müssen, wäre das Array von `mbuf`-Strukturen zu durchlaufen und die Pointer jeweils in die PacketInfo-Objekte zu schreiben. Ein `mbuf` (Paket) gehört dabei genau einer PacketInfo. Wenn dann am Ende der Pipeline Pakete gesendet werden, müssten die Pointer der `mbuf`-Strukturen den PacketInfo-Objekten wieder entnommen und in ein Array geschrieben werden. Dies ist überflüssiger Aufwand, da es möglich ist, das empfangene `mbuf`-Array beizubehalten. Dies setzt der PacketContainer um.

Der PacketContainer ist keine aktive Klasse und wird aufgerufen um spezielle, in Dagramm **??** angegebene Aufgaben umzusetzen. Eine dieser Aufgaben ist das polling von Paketen, der Ablauf wird im Sequenzdiagramm 3.2 dargestellt. Eine weitere Aufgabe ist das verwerfen von Paketen, welches durch *drop_packet(int index)* umgesetzt wird. Hierbei wird dem NetworkPacketHandler mitgeteilt, welcher `mbuf` verworfen werden soll und die Referenzen im PacketContainer selbst gelöscht. Es ist aber auch möglich, mittels *take_packet(int Index):PacketInfo** Pakete aus dem PacketContainer zu entfernen, ohne sie zu löschen. Dafür werden nur die PacketContainer internen Referenzen auf den `mbuf` und seine PacketInfo zu Nullreferenzen gesetzt und die PacketInfo zurückgegeben. Diese entnommenen Pakete können später wieder mit *add_packet(PacketInfo* pkt_info):int* eingefügt werden. Dafür wird dieses Paket hinter die bereits existenten Pakete im `mbuf`-Array gespeichert. Selbiges wird für die zugehörige PacketInfo gemacht. Zurückgegeben wird der Index, unterdem das neue Paket zukünftig erreichbar sein wird. Es können nicht nur zuvor entnommene Pakete einem PacketContainer hinzugefügt werden, sondern auch komplett neue. Dieses erstellen von Paketen ist mit dem Befehl *get_empty_packet(PacketType*

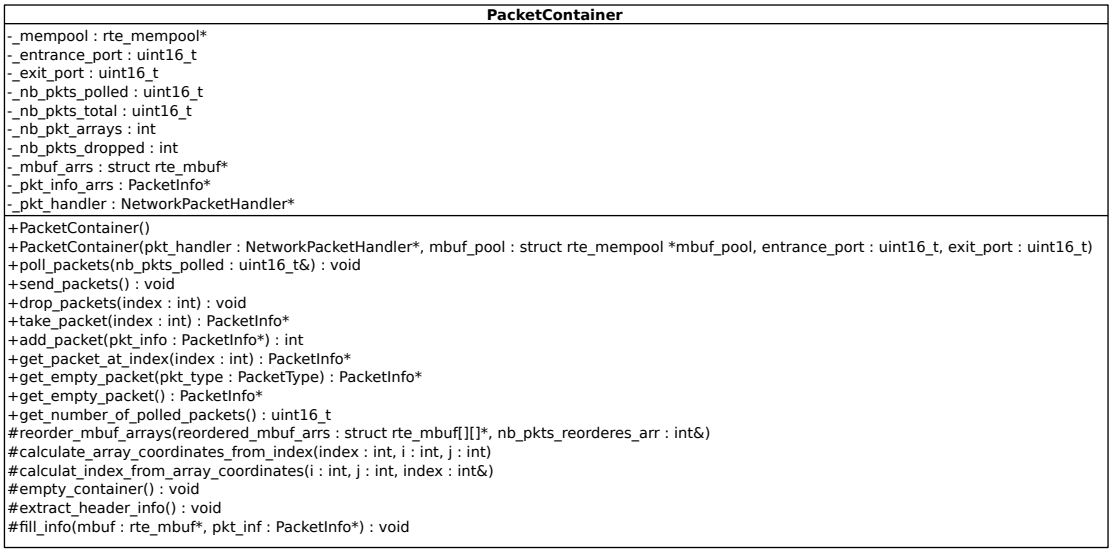


Abbildung 3.3: Klassendiagramm PacketContainer

*pkt_type):PacketInfo** möglich. Hierbei wird für einen neuen `mbuf` Speicher aus einem `mempool` alloziert und eine zugehörige PacketInfo vom gewünschten PacketType erstellt. Mithilfe dieser PacketInfo, kann der Paket Kopf im Anschluss befüllt werden. Zuletzt müssen all diese Pakete auch wieder mit *send_packets()* versendet werden. Dafür wird das `mbuf`-Array falls notwendig umsortiert, da durch *drop_packet(int Index)* Lücken entstehen können und DPDK nicht mit Nullreferenzen umgehen kann. Zuletzt wird das Array über den NetworkPacketHandler an DPDK zur Versendung übergeben.

Auch wenn bisher immer nur von je einem Array für `mbuf`s und PacketInfos gesprochen wurde, können es mehrere werden. Es gibt bei DPDK eine sogenannte BurstSize, welche angibt wie viel Pakete maximal auf einmal entgegengenommen und wieder versendet werden. Daran sind auch die Arrays größentechnisch angepasst. Da es aber durch Maßnahmen des Treatments und Analysers zur Erzeugung von neuen Paketen kommen kann, gibt es zusätzliche Arrays falls die ersten bereits voll sind. Die Verwaltung dieser Arrays ist in allen Funktionen enthalten und hat nach außen keinen sichtbaren Effekt.

3.4.2 PacketInfo

Die genaue Umsetzung, sowie die daraus resultierende Befüllung hat sich im Laufe der Entwicklungsphase sehr stark verändert. Dies hatte vor allem Performance-Gründe. In der aktuellen Variante ist die PacketInfo selbst nur für die Verwaltung des `mbuf`s sowie das Speichern seines Layer 3 und 4 Protokolls verantwortlich.

Um ausschließich notwendigen Informationen zu speichern, wird diese PacketInfo in eine protokollspezifische Variante gecastet. Diese spezialisierten Varianten erben von der eigentlichen PacketInfo und erweitern sie um Getter- und Setterfunktionen für die relevanten Header-Informationen ihrer jeweiligen Protokolle.

Auch wenn in Diagramm **??** PacketInfos mit IPv6 aufgeführt werden, sind diese noch nicht funktionsfähig. Es wurde sich entsprechend der Anforderungen zuerst auf IPv4 konzentriert.

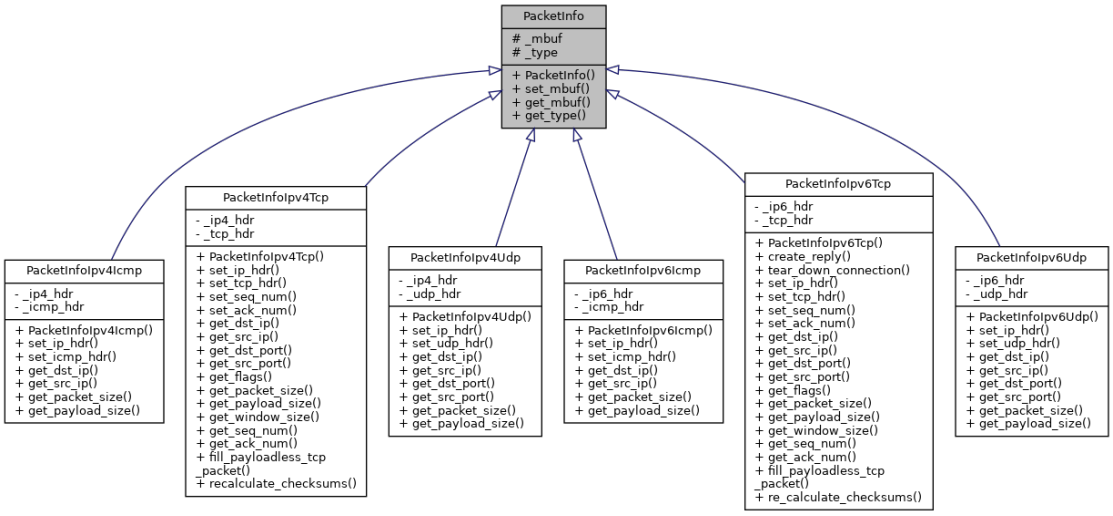


Abbildung 3.4: Klassendiagramm aller PacketInfo Varianten

3.4.3 HeaderExtractor

Wie bereits erwähnt, wurde die Extraktion der Header Informationen dezentralisiert und wird nur bei Abruf entsprechender Informationen durchgeführt. Dies führte zu einer Verringerung von Code für den HeaderExtractor im Laufe der Entwicklung, weshalb er in den PacketContainer integriert wurde. In obigen Sequenzdiagramm stellt er die Funktionen *extract_header_info()* und *fill_info(rte_mbuf* mbuf, PacketInfo* pkt_inf)*.

Dabei wird in *extract_header_info()* über die einzelnen Elemente des PacketContainers iteriert und für jeden `mbuf` die Funktion *fill_info(rte_mbuf* mbuf, PacketInfo* pkt_inf)* aufgerufen. Welche wiederum den PacketInfoCreator ausführt und den `mbuf` mit der zugehörigen PacketInfo verknüpft.

3.4.4 PacketInfoCreator

Diese Klasse ist ein Hilfsmittel, um Vererbungsketten zu vermeiden. Ihre Aufgabe ist es die zum Paket passende PacketInfo Version zu erzeugen. Dabei ließt der PacketInfoCreator die Layer 3 und Layer 4 Protokoll IDs aus, legt die entsprechenden structs auf den Speicher und speichert sie in der frisch erzeugten PacketInfo.

3.5 Inspection

Die Inspection ist für die Erkennung böswilliger IP Pakete zuständig und untersucht diese daher auf verdächtige Strukturen und Muster. Dazu wird auch eine eigene lokale Statistik erstellt, zur Auswertung genutzt und zur Informationsweitergabe mit einer globalen Statistik geteilt.

Der Initializer erstellt für jeden genutzten Thread eine eigene Inspektion welche alle Pakete dieses Threads analysiert und DDoS Attacken erkennt. Dazu wird der Inspection jeweils ein PacketContainer übergeben, der eine Menge von Paketen enthält, die über das NIC Management eingegangen sind.

Die Erkennung basiert auf einer Mustererkennung von zeitlich aufeinanderfolgenden Paketen nach einer Auftrennung in die Protokolle UDP, TCP und ICMP. UDP und ICMP Pakete werden rein mit einem vorher festgelegten Threshold geprüft, der sich an eine selbst berechnete Angriffsrate anpasst. TCP Pakete werden zusätzlich auf Zero und Small Window sowie auf SYN-FIN und SYN-FIN-ACK Muster überprüft.

Der Ablauf und Reihenfolge der Prüfungen der Inspection ist aus einer Versuchdurchführung mit einem DecisionTree für DDoS-Abwehr enstanden um einen möglichst schnellen und effizienten Ablauf zu finden. Implementiert wurde eine statische, nicht veränderliche Pipeline, die nach größten auzuschließenden Faktoren jedes Pakets vorgeht.

Der Ablauf kann grob in drei Filterstufen, auch Security Layers genannt, unterteilt werden.

1. RFC Compliance
2. Static Rules
3. Dynamic Rules

Ob ein Paket dem RFC Standard entspricht wird bereits bei der PacketInfo klar. Die Inspection bietet die Möglichkeit, bestimmte Fehler zuzulassen oder Pakete mit bestimmten Fehlern zu blockieren und zu löschen.

Die zweite Stufe der Filter setzt sich aus fest definierten Angriffen und Angriffsmustern zusammen. So sind zum Beispiel bei SYN-FIN und SYN-FIN-ACK Angriffen immer die Flags SYN und FIN oder SYN, FIN und ACK gesetzt, können sofort erkannt und das Paket verworfen werden. Weitere Angriffe die in der statischen Abwehr erkannt werden sind Zero- und Small-Window Angriffe.

In der dynamischen Filterstufe werden die Filterregeln entsprechend dem aktuellen Netzwerkverkehr und vorher eingegangenen Paketen angepasst. So dient ein Limit der Paketrate (engl. Threshold) dazu, UDP und TCP Floods abzuwehren. Eigene Verbindungstabellen der ausgehenden Netzwerkverbindungen lassen jedoch legitime Pakete die als Antwort auf eine bestehende Verbindung dienen weiterhin zu, um den legitimen Netzwerkverkehr nicht einzuschränken.

Die Verknüpfung und Ablauf der Filterung wird in dem folgenden Diagramm vereinfacht dargestellt.

Im Diagramm zu Unterscheiden gilt: die Computer 1 bis 3 sind Angreifer mit unterschiedlichen Angriffen, die ebenso in unterschiedlichen Filterstufen als Angriff erkannt werden und Computer 4 als Nutzer mit legitimen Anfragen an den Server, die den Filterregeln entsprechen. Ausgehender Verkehr aus dem internen System wird grundsätzlich vertraut und nicht zusätzlich gefiltert. Jedoch wird ausgehender Verkehr analysiert um die dynamischen Regeln anzupassen.

Nach jedem Durchlauf eines PacketContainer werden die lokalen und globalen Statistiken aktualisiert. Die Weitergabe der Informationen an die Statistik erfolgt über einen eigenen interthread Kommunikationskanal zum globalen Statistik-Thread. Die globale Statistik führt alle einzelnen Informationen zusammen und macht sie dem Nutzer in einfacher weise abrufbar.

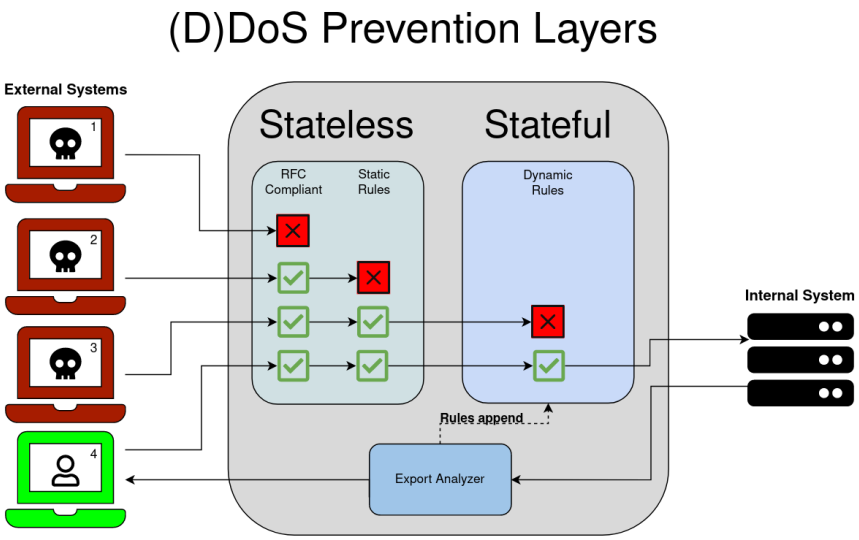


Abbildung 3.5: Stufen der Sicherheit

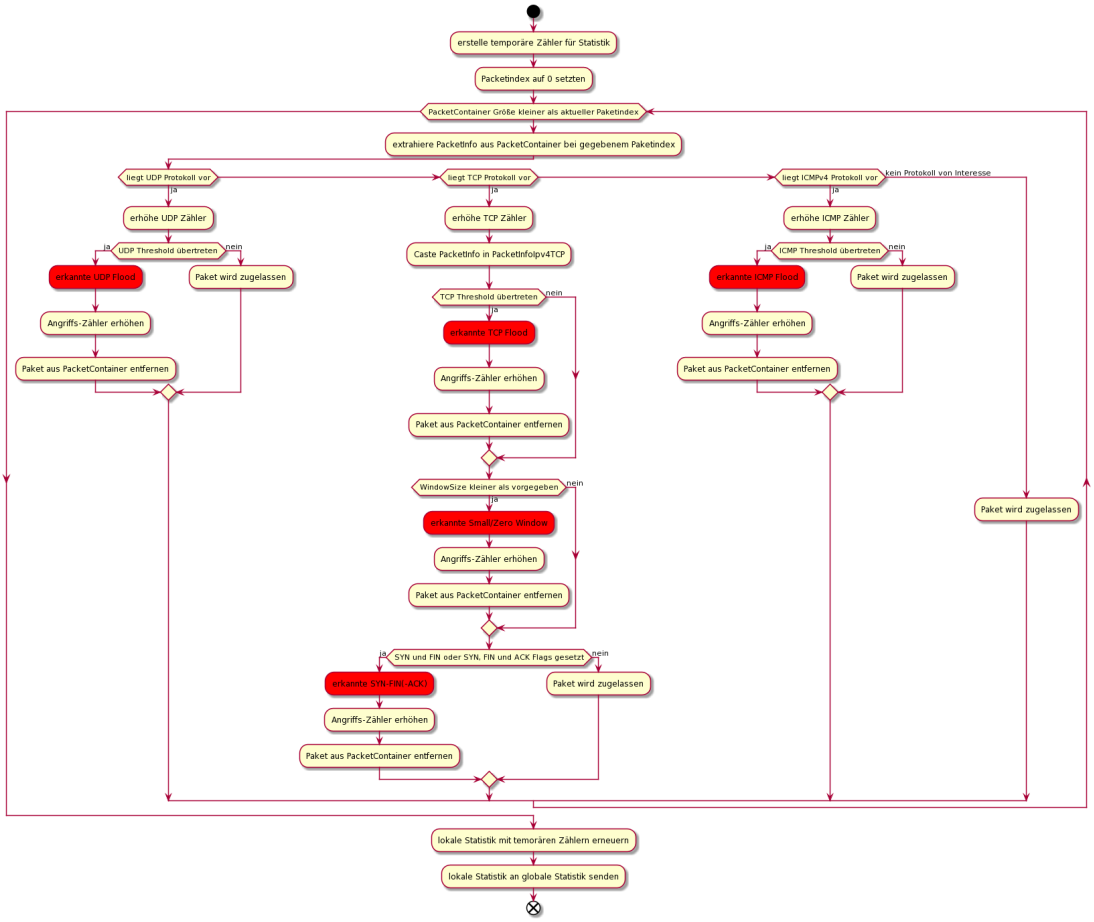


Abbildung 3.6: Aktivitätsdiagramm der Methode `analyzePacket()` der Inspection

3.6 Treatment

Das `Treatment`, welches für die Behandlung der SYN-Flut zuständig ist, erhält vom `Thread` zwei Pointer auf `PacketContainer`. Für jede Senderichtung, sowohl von Intern nach Extern als auch umgekehrt, existiert einer dieser Container. Der Ablauf in der Behandlung von Paketen unterscheidet sich basierend auf deren Senderichtung. Jedes Paket wird im `Treatment` zwar einzeln, allerdings im Kontext der gesamten Verbindung betrachtet. Die Behandlung im `Treatment` beginnt mit dem Iterieren über die Einträge im jeweiligen `PacketContainer`. Hierbei wird zugleich geprüft, ob das gerade betrachtete Paket bereits gelöscht wurde, oder von einem Typ ist, welcher nicht im `Treatment` behandelt wird. Dies ist auch im globalen Ablauf der Funktion `treat_packets()` in Abbildung 3.7 sowie 3.8 zu erkennen. Sollte dies der Fall sein, wird ebendieser Eintrag übersprungen. Sollte es sich bei dem gerade betrachteten Paket beispielsweise um ein UDP Paket handeln, so wird dieses im Treatment nicht weiter betrachtet, da dies bereits im `Analyzer` geschah.

Nach diesen ersten Tests findet jeweils eine Fallunterscheidung statt. Für Pakete, welche von extern nach intern geschickt werden sollen, gilt:

Falls es sich bei dem Paket um ein TCP-SYN-Paket handelt, so wird als Antwort hierauf ein SYN-ACK generiert, dessen Sequenznummer durch einen, vom Programm berechneten, SYN-Cookie ersetzt wird. Hierzu existiert die Methode `calc_cookie_hash()`, welche 24 der 32 Bit langen Sequenznummer generiert, welche später mit 8 Bit Timestamp in der Methode `treat_packets()` aufgefüllt werden. Dieser SYN-Cookie enthält Informationen über die Verbindungsentitäten, sowie zur Verbesserung der Effektivität einen Zeitstempel und ein Secret. Dieser SYN-Cookie ermöglicht es, im Verlauf des Verbindungsaufbaus auf das Speichern von Informationen über die Verbindung zu verzichten. Somit wird die Angriffsfläche von SYN-Floods effektiv minimiert.

Sollte ein ACK als Reaktion auf dieses SYN-ACK erhalten werden, so ist durch die Funktion `check_syn_cookie()` zu überprüfen, ob der empfangene Cookie in Form der Sequenznummer plausibel ist (Siehe Abb. 3.9). Das bedeutet, dass der Zeitstempel maximal eine Zeiteinheit alt ist, welche in diesem Programm 64 Sekunden dauert, und der restliche Cookie mit dem erwarteten Cookie übereinstimmt. Der Cookie setzt sich insgesamt zusammen aus 8 Bit Timestamp, sowie 24 Bit Hashwert über externe und interne IP-Adresse, externe und interne Portnummer sowie dem Timestamp und dem Cookie_Secret. Desweiteren ist eine Verbindung mit dem internen Server, spezifiziert in der DestIP des ACK-Paketes, aufzubauen. Zudem muss die dem ACK hinzugefügten Payload gespeichert werden, auch dies geschieht in einer separaten Map, der ACKmap. Dieses ACK-Paket muss nach erfolgreichem Verbindungsaufbau mit dem internen Server an ebendiesen verschickt werden.

Wird ein SYN-ACK von extern empfangen, so ist dies ohne Veränderung an das interne Netz zuzustellen. Allerdings muss hier ein Eintrag in der Offsetmap erzeugt werden, wobei der Offset realisierungsbedingt null ist.

Werden Pakete ohne gesetzte Flags, beziehungsweise nur mit gesetztem ACK-Flag verschickt, so findet eine Sequenznummernzuordnung und eine Anpassung von Sequenznummern statt. Hierzu wird eine Densemap mit individueller Hashfunktion, in diesem Fall XXH3, verwendet. Bei der Densemap handelt es sich um eine besonders effiziente Hashmap, welche ein Einfügen, Suchen und Löschen in bis zu vier mal weniger Zeit als eine unordered_map ermöglicht. Die Auswahl der Hashfunktion XXH3 ist dadurch motiviert, dass sie extrem schnell ist und dennoch kaum Kollisionen erzeugt. Insbesondere werden durch sie bereits auf handelsüblichen Computersystemen Hashraten von bis zu 31.5 Gbit/s erzielt.

Der Ablauf bei Empfang eines solchen Paketes ist wie folgt: Bei eingehenden Paketen wird ein zuvor berechneter Offset, welcher in der Offsetmap für jede Verbindung gespeichert ist, von der ACK-Nummer subtrahiert.

Wird ein ACK empfangen, welches zu einer Verbindung gehört, in deren Info finseen auf true gesetzt ist, so muss die ACK-Nummer angepasst, das Paket an den internen Server geschickt und der Eintrag in der Densemap verworfen werden.

Falls ein Paket mit gesetzem RST-Flag von extern empfangen wird, wird der Eintrag in der Densemap gelöscht und das empfangene Paket an den internen Server weitergeleitet. Hierbei muss keine Anpassung der ACK-Nummer vorgenommen werden.

Sollte ein FIN empfangen werden, so muss im Info-Struct, welches Teil der Offsetmap ist, der Wert finseen auf true gesetzt werden. In diesem Fall ist das Paket nach Anpassung der ACK-Nummer weiterzuleiten.

Im zweiten Fall der übergeordneten Fallunterscheidung erhält das Programm den `PacketContainer` der Pakete, welche das Netz von intern nach extern verlassen wollen. Auch hier wird, bevor ein Paket der Behandlung unterzogen wird, geprüft, ob das Paket nicht bereits gelöscht wurde, oder es sich um ein Paket falschen Typs handelt.

Falls ein Paket mit gesetzem RST-Flag von extern empfangen wird, wird der Eintrag in der Densemap gelöscht und das empfangene Paket an den internen Server weitergeleitet. Hierbei muss keine Anpassung der ACK-Nummer vorgenommen werden.

Sollte ein FIN empfangen werden, so muss im Info-Struct, welches Teil der Offsetmap ist, der Wert finseen auf true gesetzt werden. In diesem Fall ist das Paket nach Anpassung der ACK-Nummer weiterzuleiten.

Im zweiten Fall der übergeordneten Fallunterscheidung erhält das Programm den `PacketContainer` der Pakete, welche das Netz von intern nach extern verlassen wollen. Auch hier wird, bevor ein Paket der Behandlung unterzogen wird, geprüft, ob das Paket nicht bereits gelöscht wurde, oder es sich um ein Paket falschen Typs handelt.

Erhält das System ein SYN-Paket von einem internen Server, so wird dieses an das im Paket spezifizierte Ziel weitergeleitet. Eine Anpassung der Sequenznummer findet in diesem Fall nicht statt.

Erhält das System ein SYN-ACK aus dem internen Netz, so muss das System die Differenz aus der ACK-Nummer dieses Pakets, und der des in der ACKmap gespeicherten Paketes berechnen, und den Wert als Offset in der Offsetmap eintragen. Das von intern empfangene SYN-ACK Paket muss verworfen werden. Das zuvor in der ACKmap zwischengespeicherte Paket muss nun mit angepasster ACK-Nummer intACK = extACK-offset an den internen Host geschickt werden.

Wird ein Paket ohne gesetzte Flags oder mit gesetztem ACK-Flag von Intern nach Extern verschickt, so findet eine weitere Fallunterscheidung statt. Im Fall, dass finseen bereits auf true gesetzt ist, muss der Offset in der Offsetmap nachgeschlagen werden, der Eintrag daraufhin gelöscht werden und das empfangene Paket mit extSeq = intSeq + offset verschickt werden. Gesetzt den Fall, dass noch kein Eintrag in der Offsetmap existiert, muss ein neuer Eintrag in dieser erstellt werden. Der Offsetwert muss auf null, und finseen auf false gesetzt werden. Das empfangene Paket muss hiernach nach Intern weitergeleitet werden. Trifft keiner der beiden obigen Fälle ein, so muss der Offset in der Offsetmap nachgeschlagen werden und das empfangene Paket nach Intern weitergeschickt werden. Vor dem Versenden muss hierbei die Sequenznummer wie folgt angepasst werden: extSeq = intSeq + offset.

Sollte ein Paket mit gesetzem FIN-Flag erkannt werden, so ist diese Information in der Info an Stelle Key = Hash(extip, intip, extport, intport) mit dem Vermerk finseen = true, zu speichern. Das empfangene Paket ist durch das System nach extern mit extSeq = intSeq + offset weiterzuschicken.

Wird ein RST erhalten, so ist eine Anpassung der Sequenznummer vorzunehmen, das Paket entsprechend weiterzuleiten und der Eintrag in der Offsetmap an entsprechender Stelle zu entfernen.

Desweiteren könnte es unter Umständen erforderlich werden, die Einträge mit einem Timestamp zu versehen, welcher speichert, wann dieser Eintrag zuletzt verwendet wurde, sollte es zu Situationen kommen, in denen sowohl Sender als auch Empfänger die Verbindung nicht korrekt terminieren können. Dies ist bisweilen allerdings nicht implementiert, die Idee wird allerdings basierend auf den Ausgängen der Tests auf dem Testbed weiter verfolgt oder verworfen.

Nachdem ein ACK als Reaktion auf ein SYN-ACK bei dem zu entwerfenden System angekommen ist, wird die Methode `check_typ_syn_cookie()` aufgerufen. Grundsätzlich wird hier überprüft, ob der Hash-Wert aus dem empfangenen Paket mit dem eigens berechneten Hash-Wert übereinstimmt. Falls dies nicht der Fall ist oder die Differenz der Zeitstempel zu groß ist, wird ein Paket mit gesetzten Reset-Flag (RST) an den Sender geschickt. Dieses Flag zeigt an, dass die Verbindung beendet werden soll. Andernfalls wird die Verbindung als legitim erkannt und das Paket in der ACKmap zwischengespeichert, bis die Verbindung mit dem internen System erfolgreich war.

Abbildung 3.10 zeigt die parameterlose Methode `create_cookie_secret()`. Zu Beginn werden drei 16-Bit lange Zufallszahlen generiert, wobei auf cie Funktion *rand()* aus der C Standardbibliothek zugegriffen wird. Der erste mit `rand()` generierte Wert wird um 48 Bit nach links verschoben, der zweite um 32 Bit. Diese beiden Werte werden danach bitweise ODER miteinander verknüpft. Dieser verknüpfte Wert wird dann wiederum mit der dritten zufälligen 16-Bit Zahl bitweise ODER verknüpft. Das Ergebnis dieser Verknüpfung ist eine 64-Bit lange Zufallszahl, die von der Methode zurückgegeben wird.

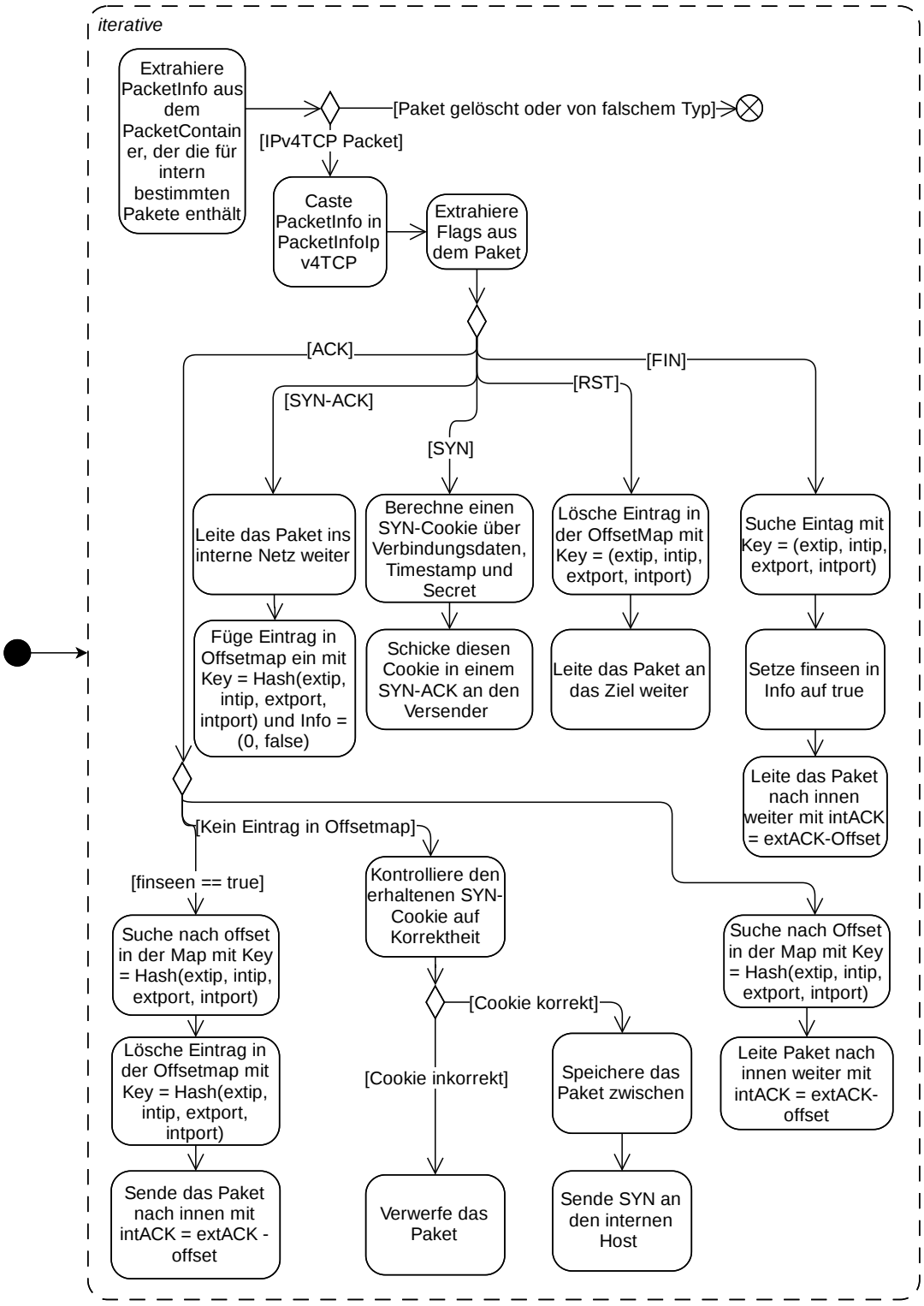


Abbildung 3.7: Aktivitätsdiagramm der Methode `treat_packets()`, Teil: Pakete nach Intern

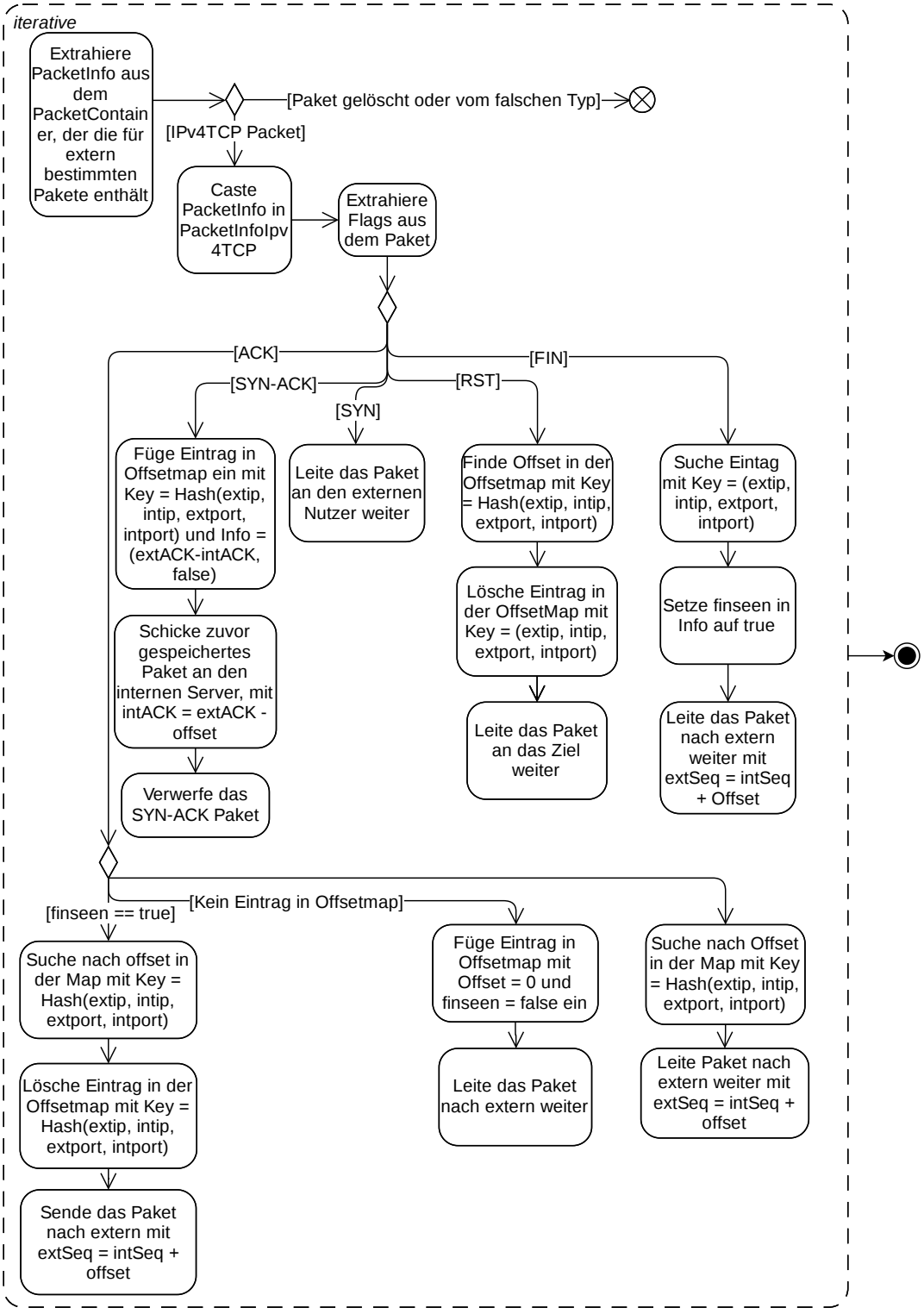


Abbildung 3.8: Aktivitätsdiagramm der Methode treat_packets(), Teil: Pakete nach Extern

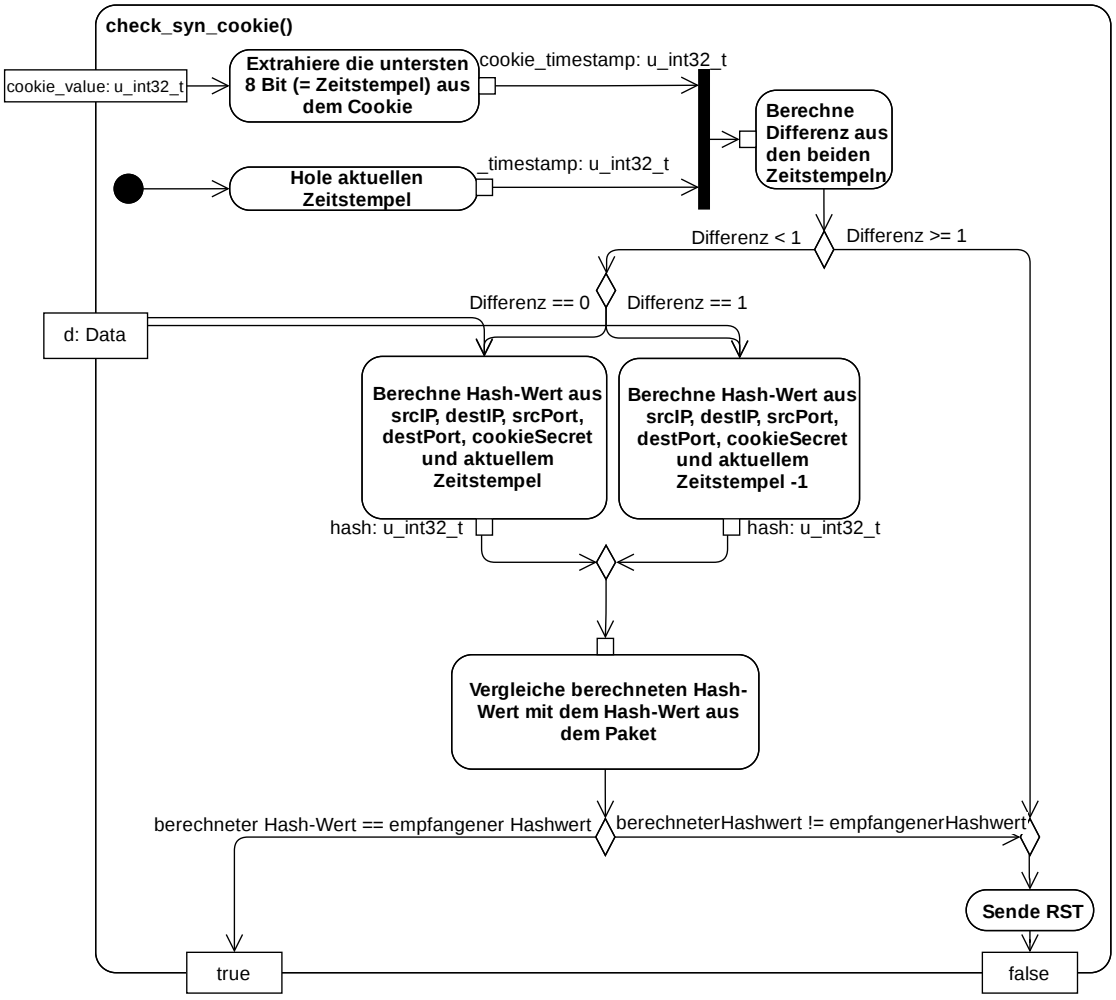


Abbildung 3.9: Aktivitätsdiagramm der Methode check_syn_cookie()

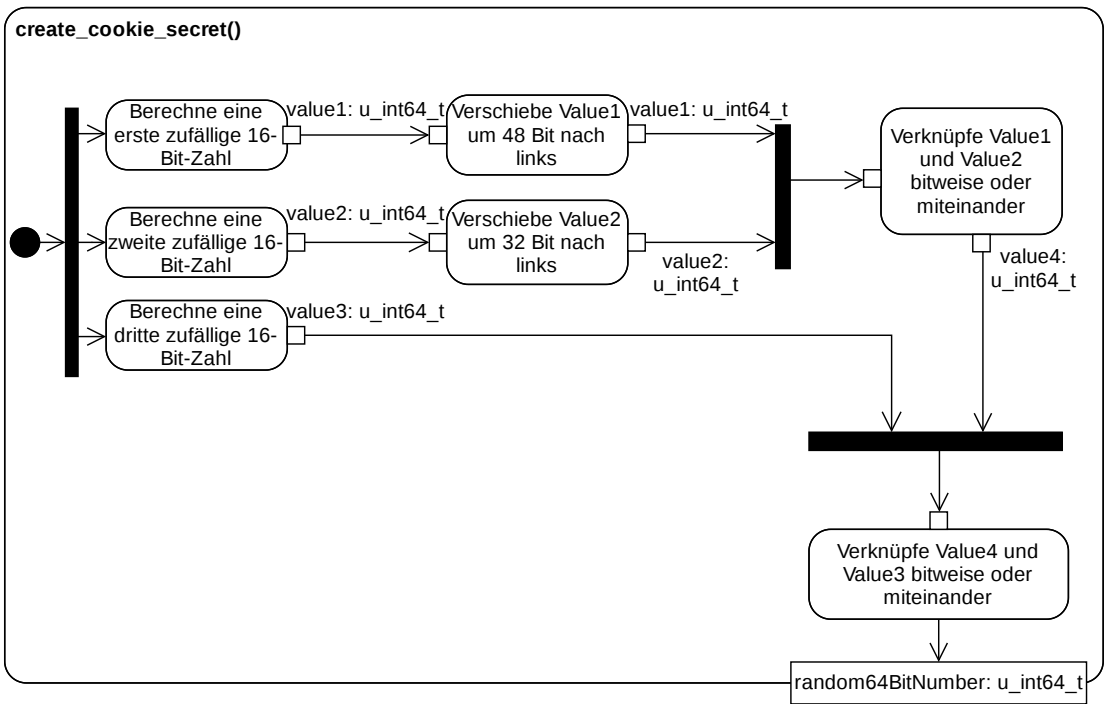


Abbildung 3.10: Aktivitätsdiagramm der Methode *create_cookie_secret()*

Kapitel 4

Bug-Review

In diesem Bug-Review werden verschiedene Fehler gesammelt. Dabei wird auf die Datei, in der sie auftreten, auf eine Beschreibung und eine Kategorisierung eingegangen. Major Bugs sind versionsverhindernd, critical bugs können auch zur Arbeitsunfähigkeit anderer führen und minor bugs haben eine geringe Auswirkung und somit eine niedrige Priorität.

Datei	**Beschreibung**	**Kategorie**
PacketInfoIpv4Icmp	Wenn von einem Paket die Header extrahiert werden soll (fill_info), wird zuerst der mbuf in der PacketInfo verlinkt, dann IP version (IPv4) und Layer 4 Protokol (ICMP) ermittelt. Danach wird die PaketInfo in die entsprechende protokolspezifische PacketInfo gecastet. Auf dieser verwandelten PacketInfo wird set_ip_hdr ausgeführt und es kommt zum segmentation fault, der im Abbruch des Threads mündet.	critical bug
Initializer	Die maximale Anzahl an Threads ist 16. Das stellt kein Problem dar, weil wir nur 12 Threads brauchen. mlx5_pci: port 1 empty mbuf pool; mlx5_pci: port 1 Rx queue allocation failed: Cannot allocate memory. Dieser Fehler tritt beim Ausführen von rte_eth_dev_start(port) auf. Womöglich handelt es sich dabei um ein mempool problem.	minor bug

Schon während der Implementierungsphase wurden Bugs wenn möglich behoben. An den in dieser Tabelle genannten Problemen wird noch gearbeitet. Die Fehlerliste wird auch in der Validierungsphase laufend erweitert, sodass im Idealfall für das abschließende Review-Dokument eine vollständige Bug-Statistik erstellt werden kann.

Kapitel 5

Auswertung der erfassten Arbeitszeiten

Mithilfe des Zeiterfassungssystems Kimai können Arbeitszeiten der Teammitglieder erfasst werden. Dabei werden nicht nur der Beginn und das Ende der Bearbeitung einer Projektaufgabe von der Software aufgenommen, sondern es wird dieser auch eine passende Aktivitätskategorie und fakultativ ein kurzer Text hinzugefügt. Die verschiedenen Kategorien sind in diesem Projekt:

- Administration (z.B. Aktualisieren des Gantt-Diagramms)
- Entwurf (z.B. Erstellen des Klassendiagramms oder der Aktivitätsdiagramme)
- Dokumentation (z.B. Arbeiten am Review-Dokument)
- Implementierung (z.B. Ausprogrammieren einer konkreten Klasse)
- Installation (z.B. Herunterladen von DPDK)
- Meetings (z.B. Montagsmeeting mit dem Betreuer um 18:30 Uhr)
- Präsentationsvorbereitung (z.B. Erstellen der Präsentationsfolien)
- Recherche (z.B. Vergleichen verschiedener Maps durch Reports im Internet)
- Test (z.B. Ausführen von Unit Tests und anderen Tests)

Mithilfe dieses Tools kann somit jedes Teammitglied regelmäßig einen Überblick erhalten, was es wann wie lange für das Software-Projekt gemacht hat.
Am Ende jeder der drei Phasen (Planung- und Entwurf, Implementierung, Validierung) werden die vom Kimai erfassten Daten ausgewertet. Dies ermöglicht, Probleme zu identifizieren und so eventuell später aufkommende Komplikationen vorzubeugen. Falls sich somit in der Auswertung Probleme zeigen, werden Verbesserungsvorschläge und Gegenmaßnahmen entwickelt.
In diesem Kapitel werden die bisher erfassten Zeiten analysiert und in Diagrammen dargestellt. Diese Diagramme wurden automatisch aus den Daten einer Excel-Datei generiert. Diese Daten stimmen mit denen aus dem Kimai überein.

Wir haben den drei Phasen folgende Kalenderwochen zugeordnet:

- Planungs- und Enturfsphase: KW 17 - KW 21
- Implementierungsphase: KW 22 - KW 24
- Validierungsphase: KW 25 - KW 29

Die Phasen werden in den folgenden Kapiteln einzeln ausgewertet und deren Ergebnisse am Ende verglichen.

5.1 Planungs- und Entwurfsphase

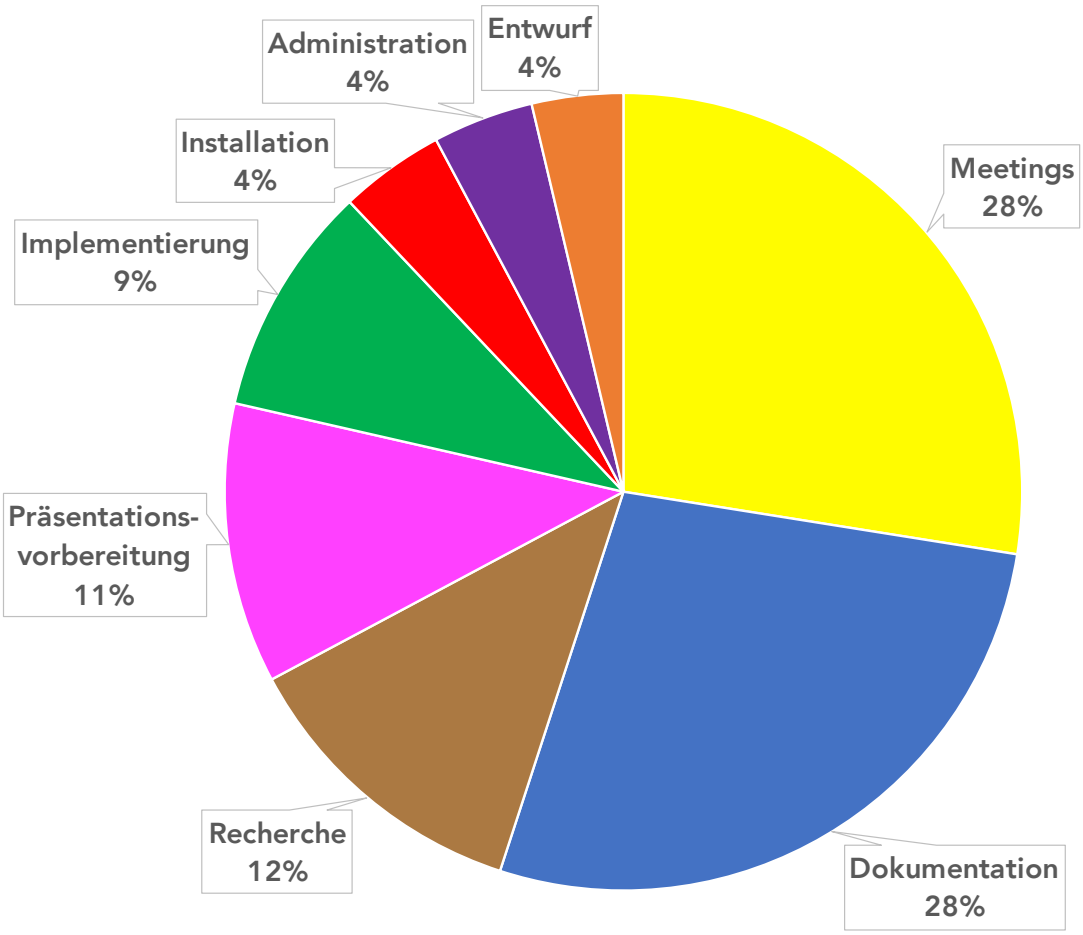


Abbildung 5.1: Planungs- und Entwurfsphase (KW 17-21): Anteile der Aufgabenkategorien an der insgesamt aufgebrachten Zeit

Das Diagramm 5.1 zeigt, dass die Zeit in der Planungs- und Entwurfsphase vorwiegend für **Meetings** aufgewendet wurde. Der Grund dafür könnten die häufigen langen Diskussionen in den Meetings mit allen Teammitgliedern sein. Zudem wurde oft die gemeinsame Arbeit an Aufgaben ebenfalls als Meeting gebucht, auch wenn eine andere Kategorie besser gepasst hätte. Beispielsweise wurde das Klassendiagramm in den wöchentlichen Meetings gemeinsam entworfen. Diese Entwurfsaktivität wurde hier aber unter Meetings und nicht unter Entwurf verbucht. Lösungsideen sind, Meetings themenspezifischer zu buchen und diese nach Möglichkeit in kleineren Gruppen abzuhalten, falls die Themen nicht alle Mitglieder betreffen. Jedoch ist zu beachten, dass hierbei die Konsistenz gewahrt wird. Das bedeutet, dass die „großen“ Meetings mit dem gesamten Team weiterhin unter Meetings verbucht werden müssen. Dies wurde sowohl bei einem Meeting

als auch durch einen Eintrag im Wiki allen Teilnehmern des Projekts kommuniziert. Weiterhin könnten weniger bzw. kürzere Meetings gehalten werden und detaillierte Fragen direkt mit den zuständigen Personen geklärt werden. Dies dann entweder in einem eigenen Meeting oder über die Plattform Zulip.

Außerdem wurde mit 26% viel Zeit für die Kategorie **Dokumentation** in Anspruch genommen. Ein möglicher Grund dafür ist, dass auch die Arbeit am Entwurf (z.B. am Klassen- oder Paketdiagramm) und die dazugehörige Entwurfsdokumentation nur unter der Kategorie Dokumentation, nicht aber unter der Kategorie Entwurf gebucht wurde. Daraus ergibt sich, dass das Team in Zukunft detaillierter buchen soll und die Kategorien womöglich angepasst werden muss. Die für die **Recherche** aufgebrachte Zeit ist tendenziell verhältnismäßig und bedarf keiner Änderung.

Es fällt auf, dass für die **Vorbereitung** von Präsentationen mit 11% in dieser Projektphase vergleichsweise viel Zeit aufgewendet wurde. Dies resultiert daruas, dass jedes Teammitglied in der ersten Woche eine Präsentation zu verschiedenen Themen wie DoS-Angriffen oder DPDK gehalten hat und ein Template für Präsentationen erarbeitet werden musste. Im weiteren Verlauf des Projektes sollte dieser Anteil sinken, da dann nur noch eine Präsentation pro Phase gehalten werden muss. Zudem müssen womöglich weniger Grafiken für das Review erstellt werden und das Präsentationsdesign kann wiederverwendet werden.

Für die **Installation** sind 6% der Zeit aufgebracht worden. Auch dieser Wert sollte im Verlauf sinken, weil ein Großteil der Installationen bereits erfolgt sind.

In der Planungs- und Entwurfsphase war der Zeitaufwand für die **Implementierung** sehr gering, was zu Beginn des Projektes allerdings kein Problem darstellt, da hier lediglich ein erster Prototyp mit stark reduziertem Funktionsumfang entwickelt wurde. Zudem kann es daraus resultieren, dass in dieser Phase mit der Implementierung lediglich zwei der acht Teammitglieder beauftragt waren. Es wird erwartet, dass mit Beginn der Implementierungsphase der für die Implementierung erfasste Aufwand sprunghaft ansteigt.

Der geringe Anteil des **Administrationsaufwandes** ist positiv zu bewerten, weil er auf eine effiziente Planung und Verwaltung hinweist.

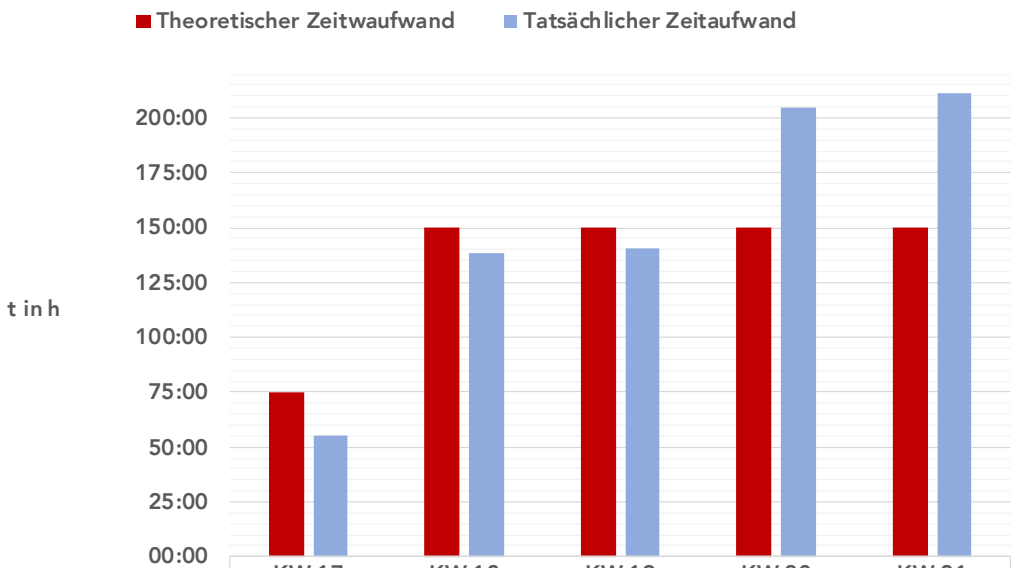
Die 3% der erfassten Zeit, die der **Entwurf** in Anspruch genommen hat, sind zu wenig. Das ist wahrscheinlich darauf zurückzuführen, dass andere Kategorien für Arbeiten am Entwurf beim Buchen verwendet wurden, beispielsweise Dokumentation oder Meetings (Erklärung siehe oben).

Im Balkendiagramm in 5.2 wird dem theoretischen Zeitaufwand pro Woche der tatsächliche Aufwand gegenübergestellt. Die hier angegebenen tatsächlichen Zeiten ergeben sich aus der Addition der wöchentlichen Zeiten aller Teammitglieder. Während Informatiker und Ingenieurinformatiker für das Softwareprojekt acht Leistungspunkte angerechnet bekommen, beträgt diese Punktzahl bei Wirtschaftsinformatiker lediglich sechs. Deshalb wird als Richtwert für die aufzubringende Zeit pro Woche zwischen den Studiengängen unterschieden: Bei Informatikern und Ingenieurinformatikern beträgt dieser Wert 20 Wochenstunden, bei Wirtschaftsinformatikern dagegen nur 15 Wochenstunden. Da das Team aus vier Informatikern, zwei Ingenieurinformatikern und zwei Wirtschaftsinformatikern besteht, beträgt der Soll-Wert pro Woche 150h.

$$2 \cdot 20h + 4 \cdot 20h + 2 \cdot 15h = 150h$$

In Woche 17 liegt Soll-Wert nicht bei 150h, sondern bei 75h, da das Projekt erst am Donnerstag begonnen hat und somit diese Woche kürzer als die anderen war.

In dieser Kalenderwoche liegt der Ist-Wert fast 20h unter diesen 75h, nämlich bei 55h 9min. Dieses Defizit ist darin zu begründen, dass in dieser ersten Woche das Zeiterfassungssystem den



	KW 17	KW 18	KW 19	KW 20	KW 21
Theoretischer Zeitwaufwand	75:00	150:00	150:00	150:00	150:00
Tatsächlicher Zeitaufwand	55:09	138:42	140:21	204:32	211:27

Abbildung 5.2: Planungs- und Entwurfsphase (KW 17-21): Vergleich des theoretischen Aufwands mit dem tatsächlichen Aufwand

Teammitgliedern noch nicht zur Verfügung stand. Deshalb mussten die Zeiten in der kommenden Woche nachgetragen werden, was jedoch nicht vom gesamten Team gemacht wurde.

In der Kalenderwoche 18 und der Kalenderwoche 19 liegt der tatsächliche Wert nur noch knapp unter den angestrebten 150 Stunden.

In den beiden letzten Wochen der ersten Phase werden diese 150h sogar stark übertroffen. Somit lässt sich generell ein positiver Trend ablesen. Es wird jedoch erwartet, dass der in der kommenden Projektphase aufgebrachte Aufwand nicht weiter stark ansteigen wird, da es neben dem Softwareprojekt noch zahlreiche andere Aufgaben für das Studium zu erledigen gibt. Somit ist es für die meisten Studierende kaum möglich, mehr als 20 bis 30 Stunden für dieses Projekt in Anspruch zu nehmen.

Den Abbildungen 5.3 und 5.4 ist zu entnehmen, dass teilweise große Unterschiede zwischen den Teammitgliedern in Bezug auf die wöchentlich aufgebrachten Stunden für das Softwareprojekt bestehen. [1] Aus dem Diagramm geht hervor, dass in Kalenderwoche 17 noch nicht jeder seine Arbeitszeiten in das Kimai eingetragen hat, da es zu diesem Zeitpunkt Ihnen noch nicht zur Verfügung stand und sie diese hätten nachtragen müssen. Somit liegt bei zwei Personen der Wert in KW17 bei 0. Es ist ein positiver Trend zu sehen, was auch in der Abbildung 5.2 deutlich wurde. Es ist hervorzuheben, dass einzelne Mitglieder weit über 20 Stunden pro Woche für das Softwareprojekt aufwenden. In der Kalenderwoche 20 liegt diese beispielsweise bei einem Teammitglied über 52 Stunden.

[1] Jede Linienfarbe in einem Projektmitglied zugeordnet. Auf die Beschriftung der einzelnen Linien wurde jedoch verzichtet, um die Anonymität der einzelnen Personen zu wahren.

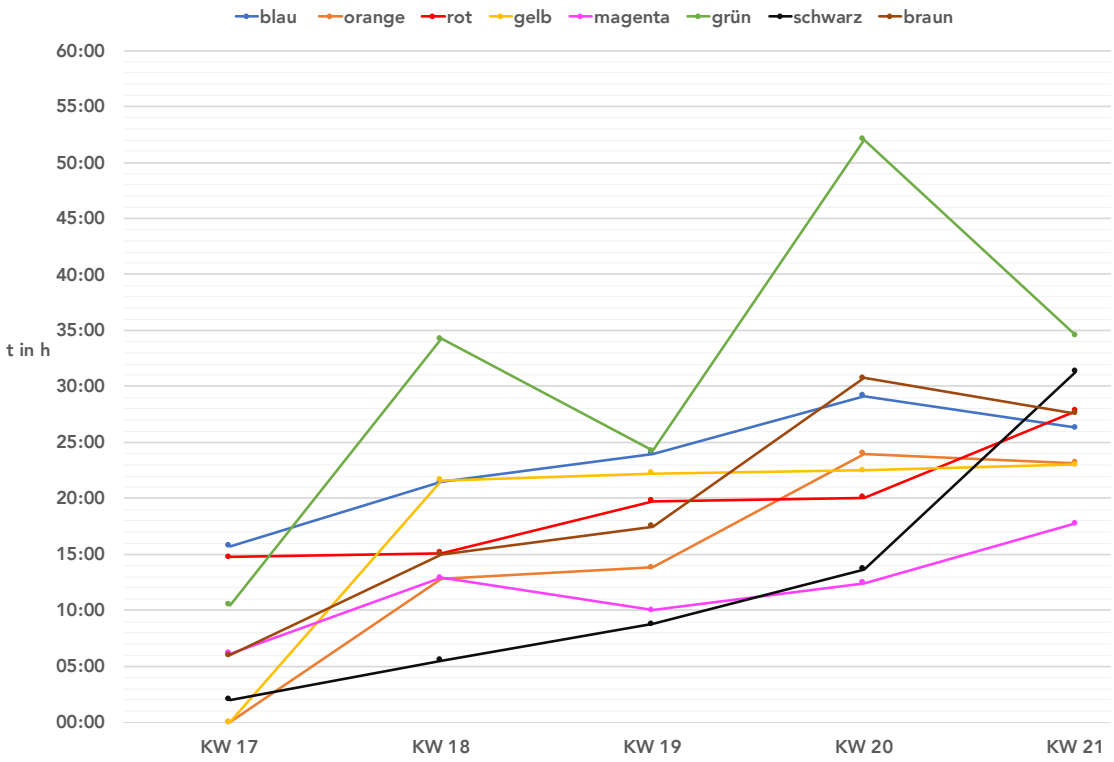


Abbildung 5.3: Planungs- und Entwurfsphase (KW 17-21): Zeitaufwand der einzelnen Teammitglieder

TEAMMITGLIED	KW 17	KW 18	KW 19	KW 20	KW 21	∑
Grün	10h 30min	34h 14min	24h 15min	52h 1min	34h 32min	155h 32min
Blau	15h 45min	21h 29min	24h	29h 8min	26h 19min	116h 41min
Gelb	0h	21h 38min	22h 15min	22h 30min	23h	89h 23min
Rot	14h 44min	15h 8min	19h 44min	20h 3min	27h 47min	97h 26min
Braun	6h	15h	17h 30min	30h 45min	27h 35min	96h 50min
Magenta	6h 10min	12h 53min	10h	12h 25min	17h 44min	59h 12min
Orange	0h	12h 49min	13h 50min	24h	23h 10min	71h 49min
Schwarz	2h	5h 31min	8h 47min	13h 40min	31h 20min	61h 18min
∑	55h 9min	138h 42min	140h 21min	204h 32min	211h 27min	750h 11min

Abbildung 5.4: Planungs- und Entwurfsphase (KW 17-21): Tabelle zum Zeitaufwand der einzelnen Teammitglieder

5.2 Implementierungsphase

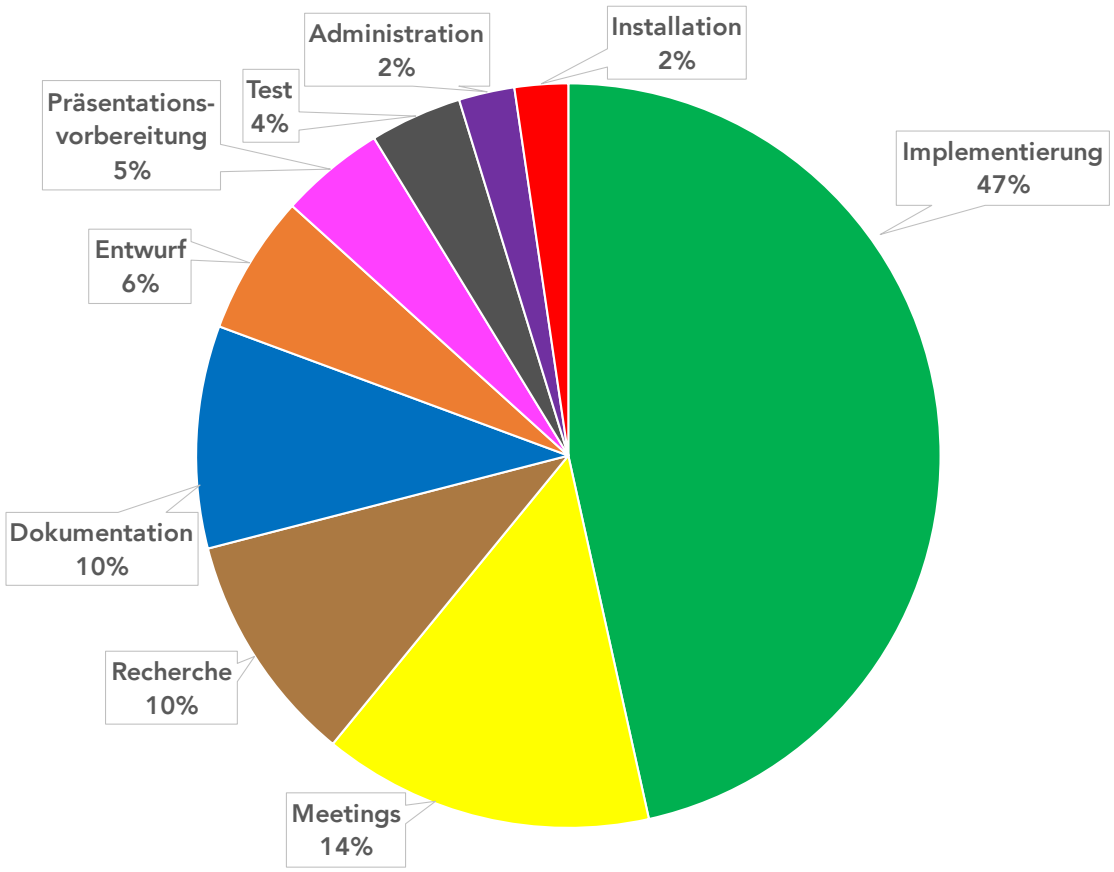


Abbildung 5.5: Implementierungsphase (KW 22-24): Anteile der Aufgabenkategorien an der insgesamt aufgebrachten Zeit

Wie in Abbildung 5.5 zu sehen ist, wurde fast die Hälfte der Zeit dieser Projektphase für die **Implementierung** verwendet. Dieser Anteil ist durchaus angemessen, weil in der Implementierungsphase der größte Teil des Systems programmiert werden soll.

Meetings stellen mit 14% die zweithäufigste Kategorie dar. Diese Größe bedarf nicht zwangsläufig einer Änderung, da die Meetings in dieser Phase lediglich wichtige Themen umfasst haben und so kurz wie möglich gehalten wurden. Außerdem bleibt zu erwähnen, dass in zwei Meetings ein Code-Review durchgeführt wurde, wodurch alle Teammitglieder voneinander lernen konnten und das System insgesamt sowie von anderen entwickelte Komponenten besser verstanden werden konnten.

Die Kategorien **Recherche** und **Dokumentation** nahmen beide ein Zehntel der insgesamt aufgebrachten Zeit in Anspruch und gehen somit mit einem angemessenen Anteil in das Projekt ein.

Die 6%, die dem **Entwurf** zugeordnet werden können, sind relativ wenig. Möglicherweise wurde für diese Kategorie aufgebrachte Zeit auch in andere Kategorien wie Dokumentation oder

Implementierung gebucht.

Positiv zu bewerten ist, dass die restlichen Kategorien, also **Präsentationsvorbereitung**, **Test**, **Administration** und **Installation**, höchstens 5% der Zeit in Anspruch genommen haben. Trotzdem ist es wichtig, dass die den Tests zugeordnete Zeit in der kommenden Validierungsphase sprunghaft ansteigt. In der Implementierungsphase hat es sich dabei vor allem um Unit-Tests gehandelt, die für die gesamte Validierung nicht ausreichen werden.

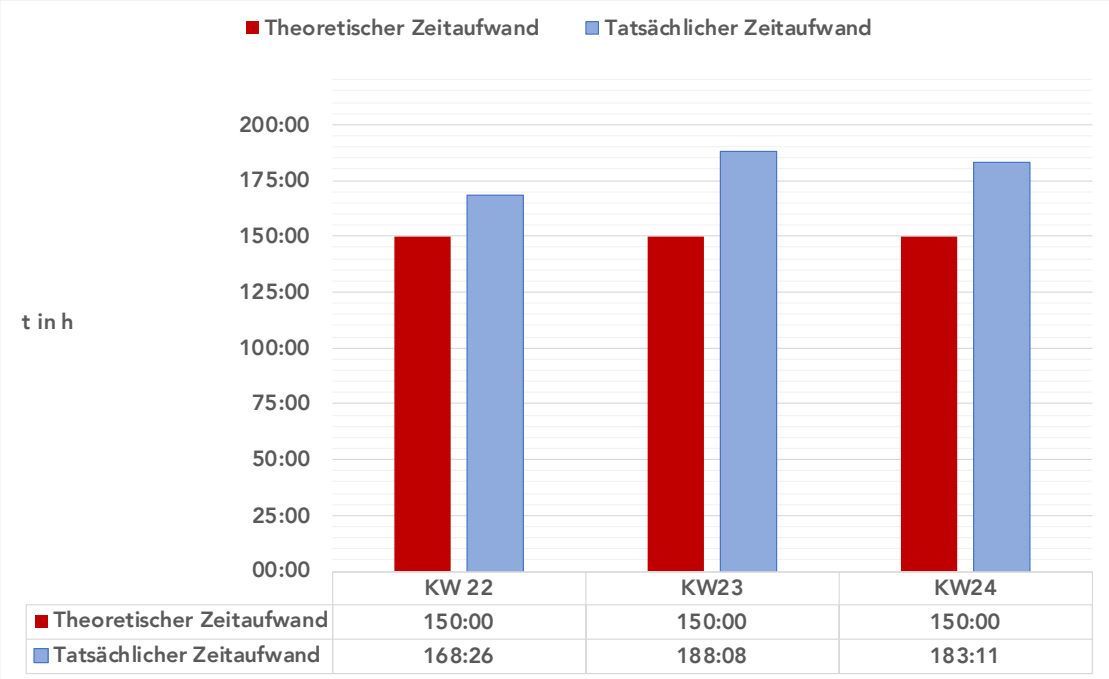


Abbildung 5.6: Implementierungsphase (KW 22-24): Vergleich des theoretischen Aufwands mit dem tatsächlichen Aufwand

Abbildung 5.6 macht deutlich, dass in jeder einzelnen Woche der Implementierungsphase (KW 22 - 24) der tatsächliche Zeitaufwand über dem theoretischen Zeitaufwand lag. Die vorgegebenen 150h Stunden Zeitaufwand pro Woche wurden in der Kalenderwoche 23 sogar um über 38h überschritten. Wenn man diese 38h durch die Anzahl der Teammitglieder teilt, die bei acht liegt, wird deutlich, dass in dieser Woche im Durchschnitt jedes Teammitglied 4h und 45min mehr Zeit für das Softwareprojekt aufgewendet hat als vorgegeben.

Aus Abbildung 5.7 lässt sich grundsätzlich kein starker Anstieg der Zeit, die jeder einzelne Beteiligte für das Softwareprojekt aufgewendet hat, erkennen. Bei sechs der acht Teammitglieder liegt ihr persönliches Maximum in diesem Zeitraum in KW 23. Hier liegt auch der Punkt, mit dem größten Wert in der Implementierungsphase. Der dazugehörige Wert beträgt 32h 30min (vgl. Abb. 5.8). Bei einigen Teammitglieder kann erkannt werden, dass die empfohlene Zeit von 15 bzw. 20 Stunden in einzelnen Fällen unterschritten wird. Üblicherweise wird auf die gesamte Zeit bezogen der Soll-Wert allerdings erfüllt, was man auch in Abbildung 5.6 erkennen kann.

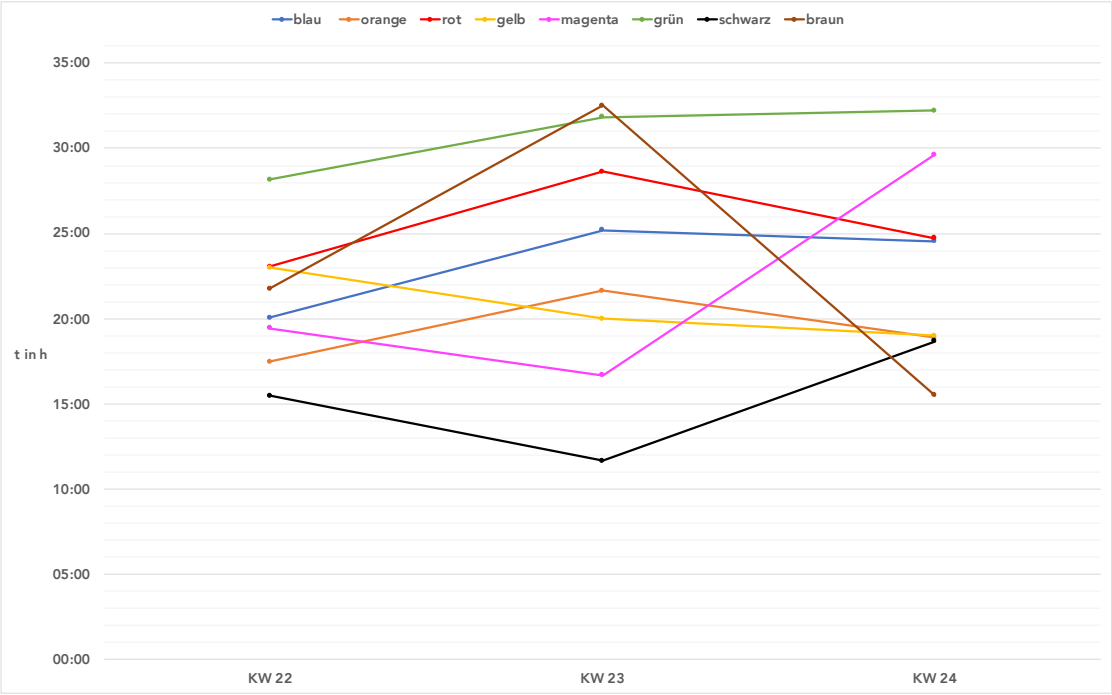


Abbildung 5.7: Implementierungsphase (KW 22-24): Zeitaufwand der einzelnen Teammitglieder

TEAMMITGLIED	KW 22	KW 23	KW 24	∑
Grün	28h 10min	31h 51min	32h 13min	92h 14min
Blau	20h 3min	25h 11min	24h 33min	69h 47min
Gelb	23h	20h	19h	62h
Rot	23h 4min	28h 38min	24h 44min	76h 26min
Braun	21h 45min	32h 30min	15h 30min	69h 45min
Magenta	19h 27min	16h 40min	29h 38min	65h 45min
Orange	17h 28min	21h 38min	18h 52min	57h 58min
Schwarz	15h 29min	11h 40min	18h 41min	45h 50min
∑	168h 26min	188h 8min	180h 41min	539h 45min

Abbildung 5.8: Implementierungsphase (KW 22-24): Tabelle mit den erfassten Zeiten

5.3 Vergleich der Projektphasen

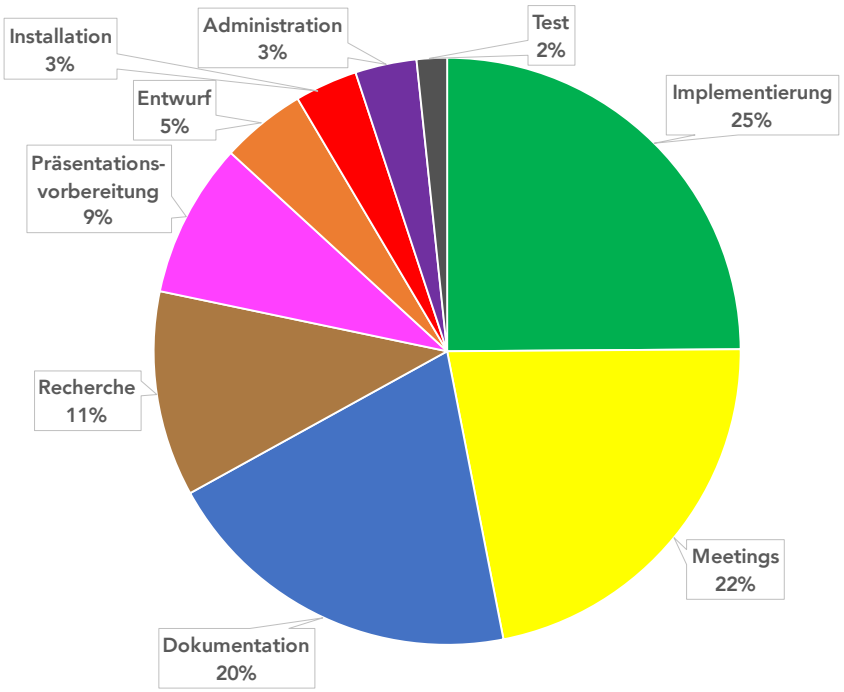


Abbildung 5.9: Planungs- und Entwurfsphase und Implementierungsphase (KW 17-24): Anteile der Aufgabenkategorien an der insgesamt aufgebrachten Zeit

In Abbildung 5.6 wird die Verteilung der Kategorien bezogen auf den gesamten Projektzeitraum dargestellt.

Bei einem Vergleich der Anteile der Kategorien an der insgesamt aufgebrachten Zeit zwischen Planungs- und Entwurfsphase und Implementierungsphase fällt als erstes auf, dass sich die für die **Implementierung** aufgebrachte Zeit vervielfacht hat. Dieser sprunghafte Anstieg wurde auch prognostiziert, weil die Implementierung in der ersten Phase lediglich einen ersten Prototyp umfasste und der zentrale Bestandteil der jetzt abgeschlossenen Phase ist.

Weiterhin ist der Anteil der **Meetings** von 29 auf 14% gesunken. Da dieser Anteil im ersten Abschnitt des Softwareprojekts viel zu hoch war und schließlich auf einen adäquaten Wert gesenkt werden konnte, kann geschlussfolgert werden, dass die richtigen Maßnahmen getroffen wurden. So wurden Diskussionen, die nicht alle Teammitglieder betroffen haben, von den wöchentlichen Treffen in kleinere Gruppen verlagert und dann in der entsprechenden Kategorie themenbezogen gebucht.

Die **Dokumentation** hat während der Implementierungsphase ebenfalls weniger Zeit in Anspruch genommen als in der Planungs- und Entwurfsphase. Das umfassende Reviewdokument der ersten Phase, bestehend aus dem Pflichtenheft und einer Entwurfsdokumentation, stellte im ersten Monat des Projekts eine der Hauptaufgaben dar, während ihm in dieser Phase lediglich eine mittelhohe Priorität zugeordnet wurde.

Der **Rechercheaufwand** ist leicht gesunken, da sich zu Beginn des Projekts besonders viel

Wissen angeeignet werden musste, welches im Laufe des Projekts dann zunehmen angewendet werden musste. Da fast alle Teammitglieder zu Beginn des Projekts recht unerfahren waren, scheinen die 11% der im gesamten Projektzeitraum aufgebrachten Zeit ein geeigneter Anteil zu sein.

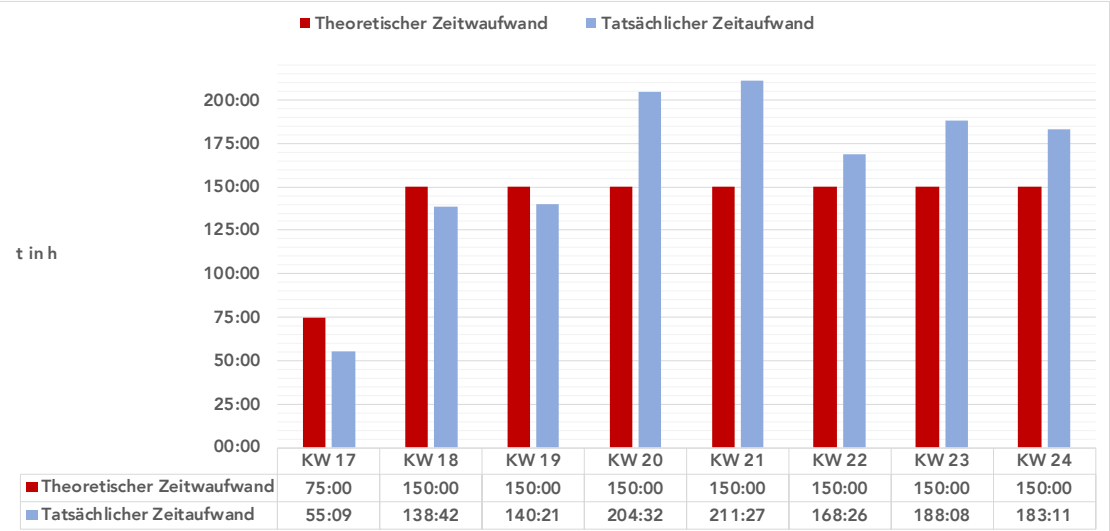
Insgesamt wurden 9% der Zeit für die Vorbereitung von **Präsentationen** verwendet, was ebenfalls positiv zu bewerten ist. An dieser Stelle muss angemerkt werden, dass die Zeiterfassung der Implementierungsphase lediglich die Zeiten bis KW 24 umfasst, wodurch die letzten drei Tage vor der Präsentation nicht abgedeckt wurden. In dieser Zeit wird vermutlich noch eine nicht zu vernachlässigende Menge an Zeit für die Vorbereitung des zweiten Reviews investiert werden. Aus diesem Grund und weil jedes Teammitglied in der ersten Woche des Projekts eine Präsentation zu einem bestimmten Thema ausgearbeitet hat, ist der Wert von 11% auf 5% zurückgegangen.

Der **Entwurf** umfasste insgesamt 5% der Zeit und wurde in beiden Phasen tendenziell zu wenig beziehungsweise in anderen, ähnlichen Kategorien erfasst.

Für die **Installation** musste sogar in der ersten Phase nur 5% der Zeit verwendet werden. In der Implementierungsphase sank dieser Anteil auf 2%. Da fast alle benötigten Komponenten bereits installiert sein sollten, lässt sich vermuten, dass der Anteil dieser Kategorie in der letzten Projektphase weiter gegen null geht.

Der **Administrationsaufwand** konnte schon in der ersten Phase des Projekts gering gehalten werden und ging auch in der Implementierungsphase in einem geeigneten Maß ein.

In der zweiten Phase hat sich der **Testaufwand** von 2 auf 4% verdoppelt. Das zeigt, dass die Projektverlauf zunehmende Bedeutung des Testens erkannt wurde. Wichtig ist allerdings, dass beim Testen in der nächsten Phase ein sprunghafter Anstieg ähnlich wie bei der Implementierung zwischen erster und zweiter Phase zu erkennen ist.



	KW 17	KW 18	KW 19	KW 20	KW 21	KW 22	KW 23	KW 24
Theoretischer Zeitwaufwand	75:00	150:00	150:00	150:00	150:00	150:00	150:00	150:00
Tatsächlicher Zeitaufwand	55:09	138:42	140:21	204:32	211:27	168:26	188:08	183:11

Abbildung 5.10: Planungs- und Entwurfsphase und Implementierungsphase (KW 17-24): Vergleich des theoretischen Aufwands mit dem tatsächlichen Aufwand

Folgendes lässt sich in Abbildung 5.10 erkennen: Während in den ersten drei Wochen des Softwareprojekts (KW 17-18) der tatsächliche Zeitaufwand noch leicht unter dem theoretischen Wert

liegt, ändert sich dies in den darauffolgenden Wochen dahingehend, dass der tatsächliche Zeitaufwand den theoretischen (zum Teil stark) übersteigt. Die Feststellung, dass mehr Zeit aufgewendet wird bzw. werden muss als vorgegeben, stimmt mit der Aufwandsschätzung aus dem ersten Reviewdokument überein. Denn wenn die theoretischen Werte aller bisherigen Wochen (KW 17-24) aufaddiert werden, ergibt sich ein Wert von 1289h 56min (vgl. Abb. 5.12). Diesem stehen 1125h entgegen:

$$1 \cdot 75h + 7 \cdot 150h = 1125h$$

In der ersten Aufwandssschätzung schätzten wir den Aufwand des gesamten Projekts auf 2107h. Würden wir in den kommenden Wochen (KW 25-29) „lediglich“ den theoretische berechnete Zeit brauchen, wären das zu den bisher benötigten 1289h 56min noch 675h, also insgesamt 1964h 56min. Dieser Wert kommt dem Schätzwert aus der Aufwandsschätzung sehr nahe.

$$4 \cdot 150h + 1 \cdot 75h = 675h$$

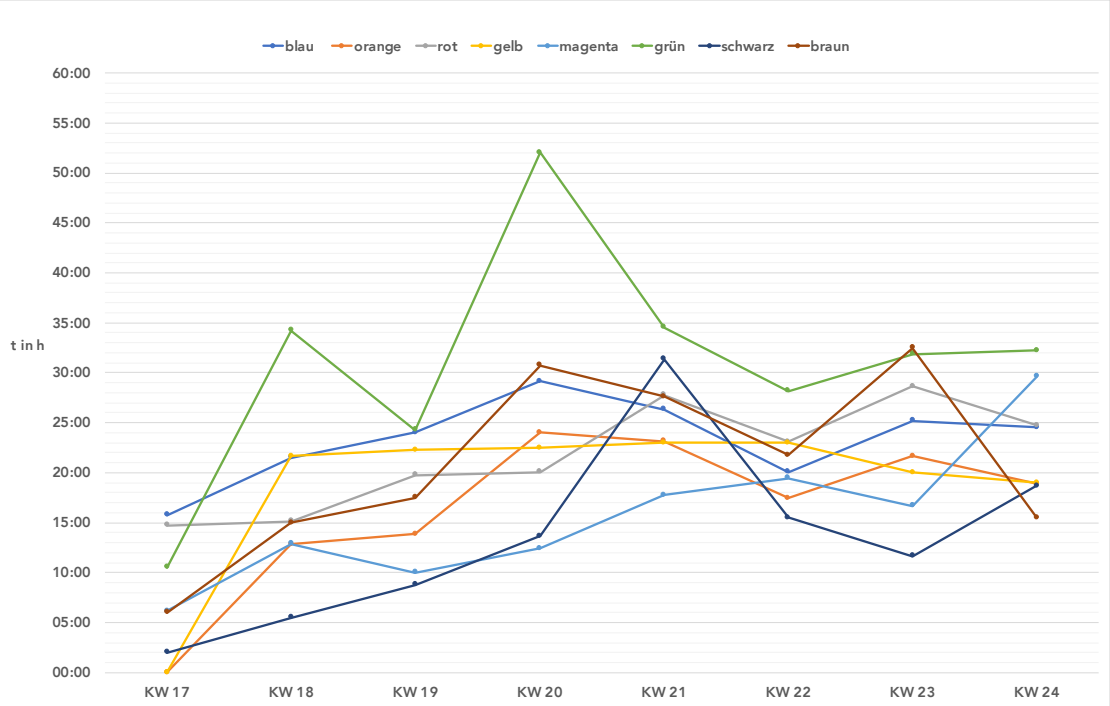


Abbildung 5.11: Planungs- und Entwurfsphase und Implementierungsphase (KW 17-24): Zeitaufwand der einzelnen Teammitglieder

Die Abbildung 5.11 zeigt den Zeitaufwand der einzelnen Teammitglieder in den Kalenderwochen 17 bis 24. In den Kalenderwochen 17 bis 21 (Planungs- und Entwurfsphase) ist grundsätzlich ein Anstieg bei den meisten Teammitgliedern zu sehen. In den folgenden Wochen (Implementierungsphase) ist dieser nicht mehr so stark zu erkennen. Auch nimmt in den Kalenderwochen 22 bis 24 der Abstand zwischen den einzelnen Punkte in der Abb. 5.11 ab. Das heißt, dass der Zeitaufwand der einzelnen Teammitglieder sich nicht mehr so stark unterscheidet wie das zum Beispiel in KW 20 der Fall ist. So liegt etwa die Spannweite in KW 20 bei über 39h, während in

KW 22 diese bei unter 13h liegt.

$$R_{KW20} = 52h\ 1min - 12h\ 25min = 39h\ 36min$$

$$R_{KW22} = 28h\ 10min - 15h\ 19min = 12h\ 51min$$

Der maximale Wert an Wochenstunden pro Person bleibt in KW 20 mit über 52h (vgl. Abb. 5.12).

TEAMMITGLIED	KW 17	KW 18	KW 19	KW 20	KW 21	KW 22	KW 23	KW 24	∑
Grün	10h 30min	34h 14min	24h 15min	52h 1min	34h 32min	28h 10min	31h 51min	32h 13min	247h 46min
Blau	15h 45min	21h 29min	24h	29h 8min	26h 19min	20h 3min	25h 11min	24h 33min	186h 28min
Gelb	0h	21h 38min	22h 15min	22h 30min	23h	23h	20h	19h	151h 23min
Rot	14h 44min	15h 8min	19h 44min	20h 3min	27h 47min	23h 4min	28h 38min	24h 44min	173h 52min
Braun	6h	15h	17h 30min	30h 45min	27h 35min	21h 45min	32h 30min	15h 30min	166h 35min
Magenta	6h 10min	12h 53min	10h	12h 25min	17h 44min	19h 27min	16h 40min	29h 38min	124h 57min
Orange	0h	12h 49min	13h 50min	24h	23h 10min	17h 28min	21h 38min	18h 52min	131h 47min
Schwarz	2h	5h 31min	8h 47min	13h 40min	31h 20min	15h 29min	11h 40min	18h 41min	107h 8min
∑	55h 9min	138h 42min	140h 21min	204h 32min	211h 27min	168h 26min	188h 8min	180h 41min	1289h 56min

Abbildung 5.12: Planungs- und Entwurfsphase und Implementierungsphase (KW 17-24): Tabelle mit den erfassten Zeiten

Kapitel 6

Abkürzungsverzeichnis

DDoS Distributed Denial-of-Service

DoS Denial-of-Service

DRoS Distributed Reflected Denial-of-Service

Gbps Giga bit pro sekunde

KW Kalenderwoche

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.

Abbildungsverzeichnis