Kapitel 1

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.

1.1 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.

1.1.1 Netzwerkaufbau

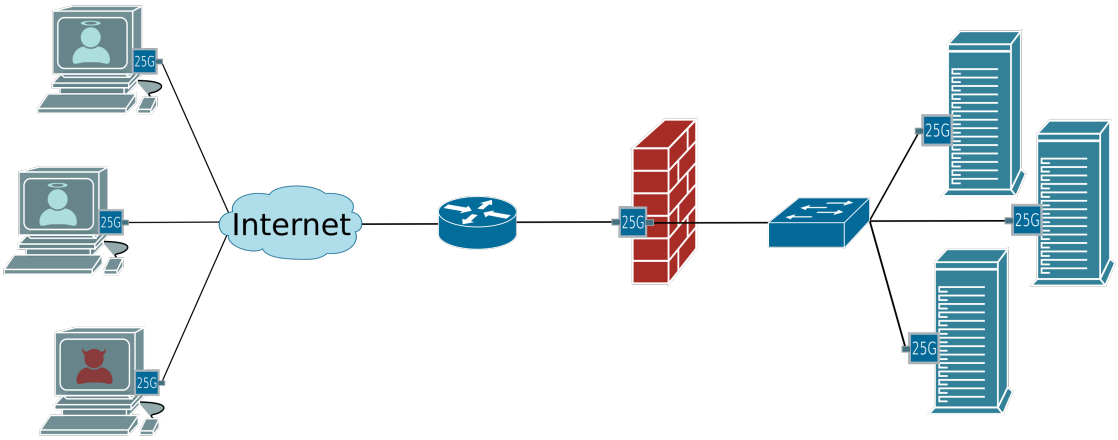

Abbildung 1.1: Realaufbau unter Verwendung eines Angreifers

Die Abbildung 1.1 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 in der Abbildung ist jeweils das Internet zu erkennen. In diesem sind unterschiedliche 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 zwischen dem Angriff eines einzelnen Angreifers, oder einer Menge von einem Angreifer gekaperten und gesteuerten Computer, also eines Botnets, vorzunehmen.

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. Deswegen musste ein alternativer Aufbau gefunden werden, der allerdings vergleichbare Charakteristika aufweisen muss.

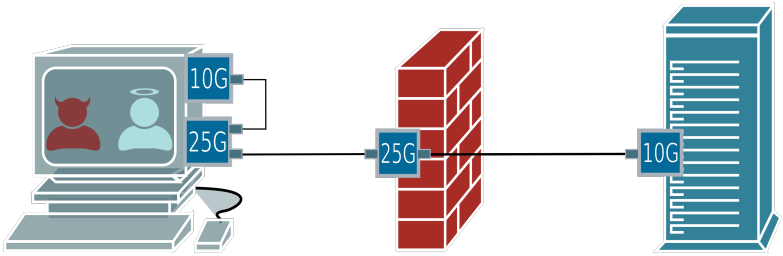

Abbildung 1.2: Versuchsaufbau

Der für das Projekt verwendete Versuchsaufbau untergliedert sich ebenfalls in drei Teile. Auch hier beginnt die Darstellung 1.2 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 1.2 auf der rechten Seite befindet, 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.

1.1.2 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 legitime Pakete zwischen dem internen 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.

1.1.2.1 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 Potenzial, 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 aber 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.

1.1.2.2 Kontrollfluss eines Paketes

In diesem Abschnitt soll veranschaulicht werden, wie genau die Behandlung eines Paketes vom `NicManagement` bis zum `Treatment` erfolgt. Dabei werden die Pakete selbst als Akteure angesehen und 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 1.3 am Ende der Sektion 1.1.2 zu finden.

1.1.2.3 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 1.4: 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 Quell- und Ziel-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.

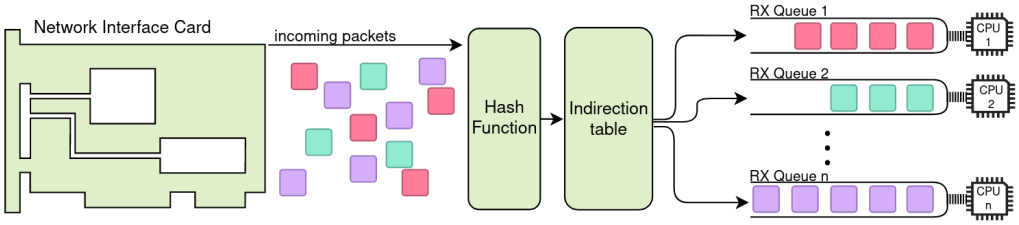

Abbildung 1.4: Beispielhafte Paketverarbeitung mit Receive Side Scaling

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 geholt. In der zu entwickelnden Software geschieht dieses Paket-holen (engl. „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 Verkehrs 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.

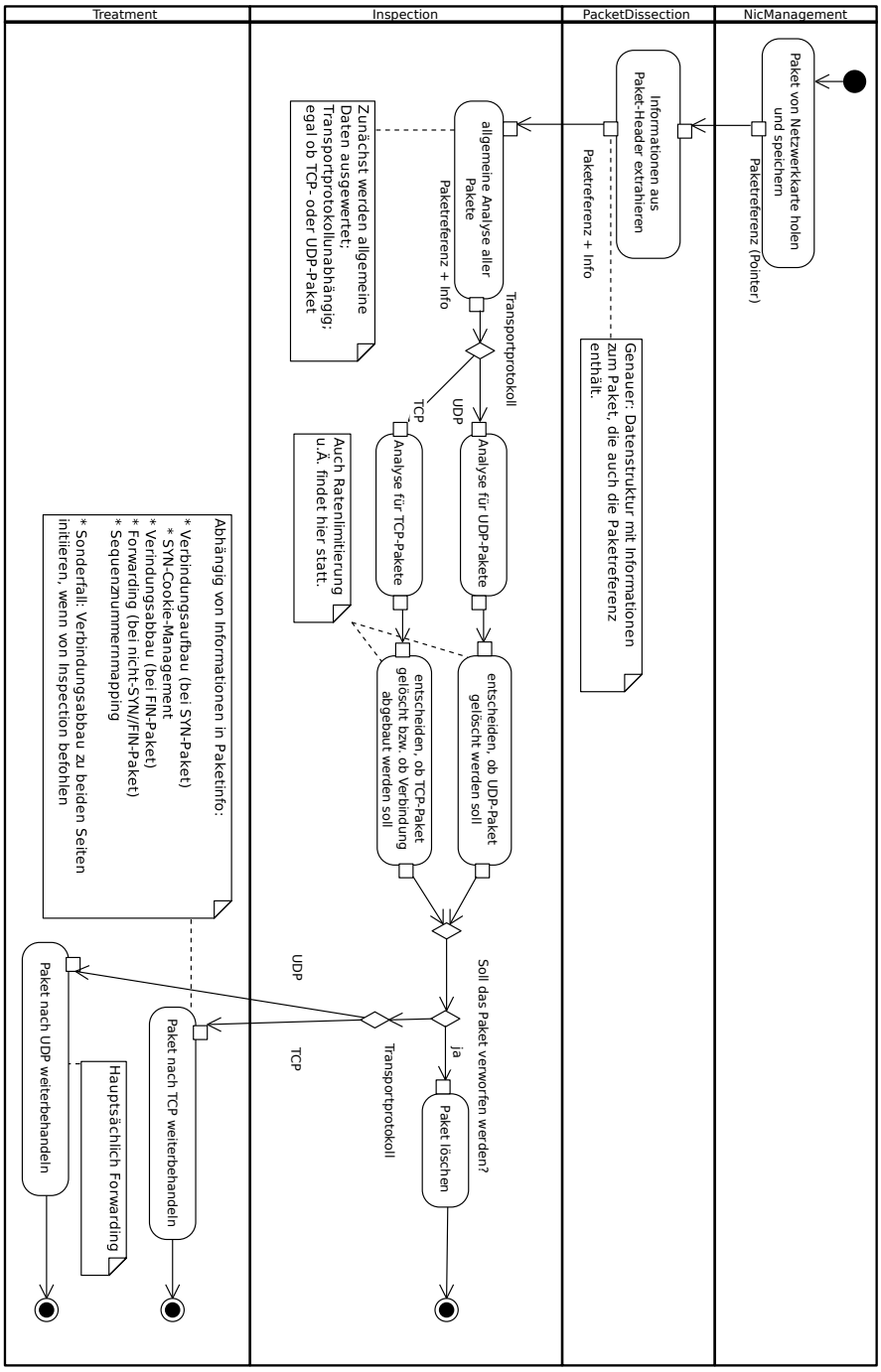

Abbildung 1.3: Schematische Darstellung des Kontrollflusses

1.2 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.

1.2.1 Paketdiagramm

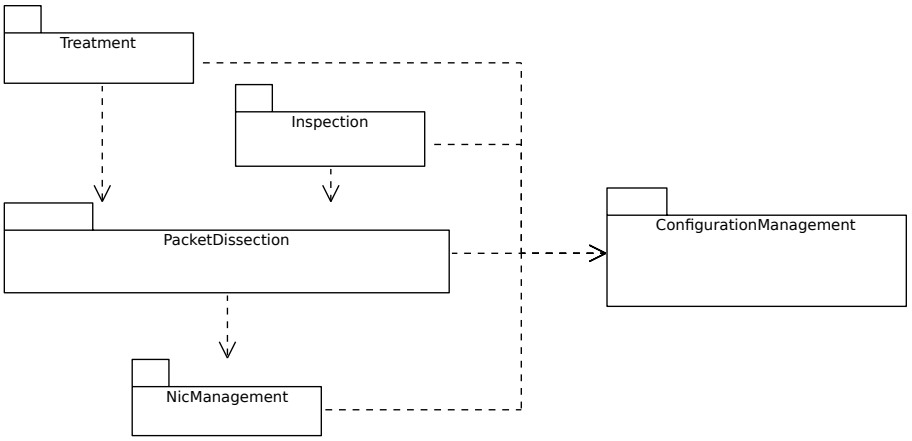

Abbildung 1.5: Paketdiagramm

Grundsätzlich ist es angedacht, wie im Paketdiagramm 1.5 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.

1.2.2 NicManagement

Das `NicManagement` übernimmt, wie im letzten Review-Dokument erwähnt, das Senden, das Pollen und das Löschen von Paketen. Dieses Paket wurde eingeführt, um bestimmte Funktionen und Initialisierungsschritte vom DPDK zu kapseln. Dabei handelt es sich vor allem um folgende Operationen: `rte_eth_rx_burst()` und `rte_eth_tx_burst()` 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.

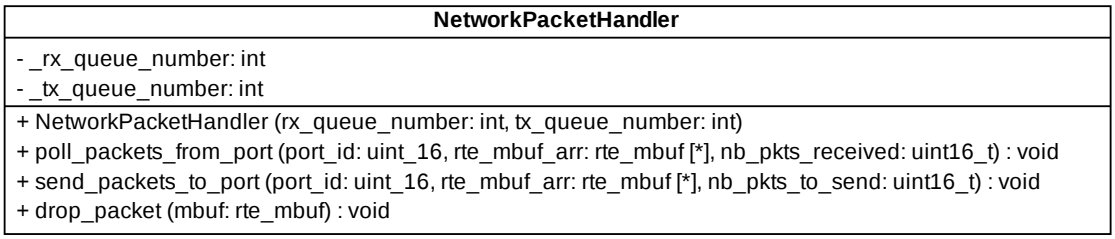

Abbildung 1.6: Klassendiagramm: `NetworkPacketHandler`

Um ein neues Objekt der Klasse `NetworkPacketHandler` zu erzeugen, muss der Konstruktor aufgerufen werden. Diesem müssen zwei Parameter übergeben werden (vgl. Abb. 1.6): Zum einen die ID der RX-Queues (`rx_queue_number`), zum anderen die der TX-Queues(`tx_queue_number`).

Für das Verwerfen von Paketen ist die Methode `drop_packet()` zuständig. Hierbei muss ein Pointer auf das Paket übergeben werden, das verworfen werden soll.

Die Methode `poll_packets_from_port()` holt Pakete von einem spezifischen Port. Dazu werden die ID des Ports, von denen die Pakete geholt werden sollen, ein Array, auf welches die Pointer der geholten mbufs geschrieben werden sollen und die Anzahl der Pakete, die in das Array geholt werden sollen, benötigt. Ähnliche Parameter werden der Methode `send_packets_to_port()` übergeben.

1.2.3 ConfigurationManagement

Das Paket `ConfigurationManagement` kümmert sich um die Initialisierung der Software. Des weiteren 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 grundlegenden Einstellungen der Software enthält. An anderer Stelle kann dann

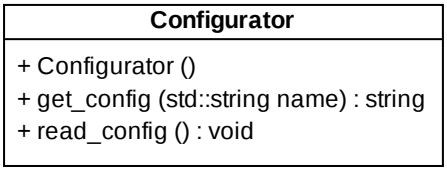

Abbildung 1.7: Klassendiagramm: `Configurator`

über verschiedene Methoden auf Konfigurationsinformationen zugegriffen werden. Abbildung 1.7 zeigt das Klassendiagramm vom `Configurator`.

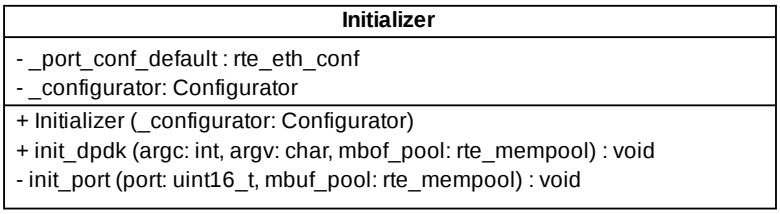

Abbildung 1.8: Klassendiagramm: `Initializer`

Die Klasse `Initializer` dient dazu die für die Bibliothek DPDK notwendigen Voraussetzungen zu schaffen. Das Klassendiagramm befindet sich in Abb. 1.8.

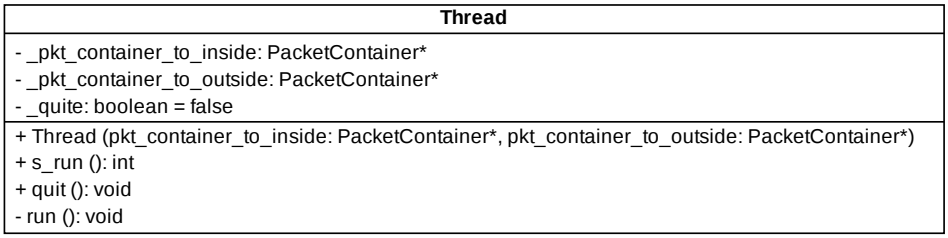

Abbildung 1.9: Klassendiagramm: `Thread`

Die Klasse `Thread` enthält den Ablaufplan für die Workerthreads des Systems (vgl. Abb. 1.9).

1.2.4 PacketDissection

Der Zweck dieses Pakets ist, sämtliche Daten, die `Inspection` 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 Programms 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. Im `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 immer noch 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.

1.2.5 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 Minimum begrenzt werden, was der Auswertungsgeschwindigkeit jedes Inspection-Threads zugute kommt und ein Bottleneck-Problem an der Inspection vorbeugt.

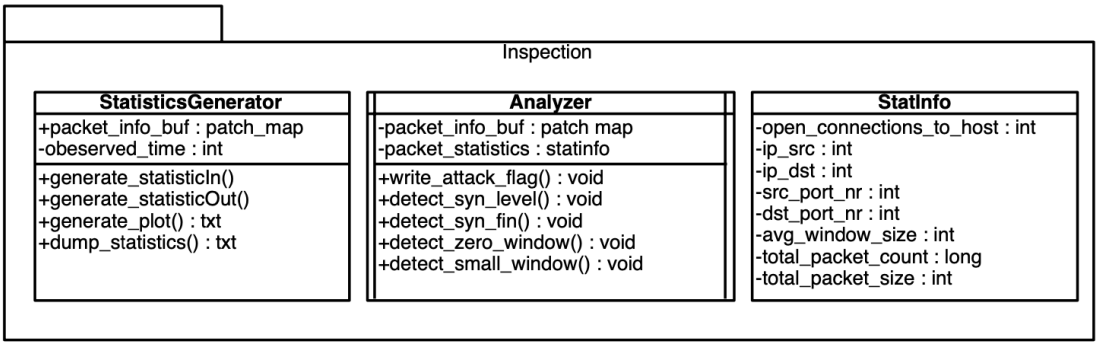

Abbildung 1.10: Altes Klassendiagramm der Inspection aus der Implementierungsphase

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 großteils 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.

In Abb. 1.10 und Abb. 1.11 lässt sich gut erkennen, wie sich die `Inspection` während der Überarbeitung verändert und vereinfacht hat.

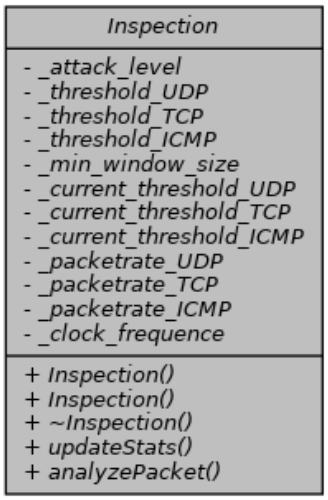

Abbildung 1.11: Aktuelles Klassendiagramm der `Inspection` aus der Planungs- und Entwurfsphase

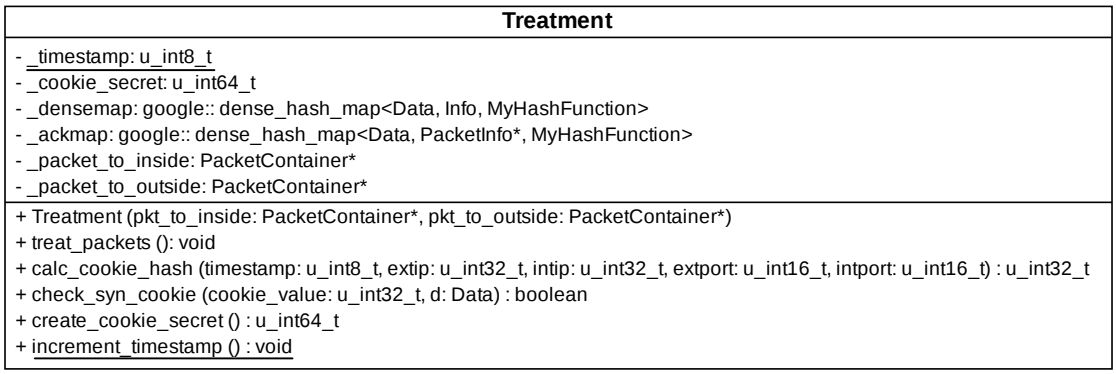

Abbildung 1.12: Klassendiagramm des Treatments aus der Implementierungsphase

1.2.6 Treatment

Abbildung 1.12 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. 1.13).

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 `Inspection` (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 in der `Inspection` alle

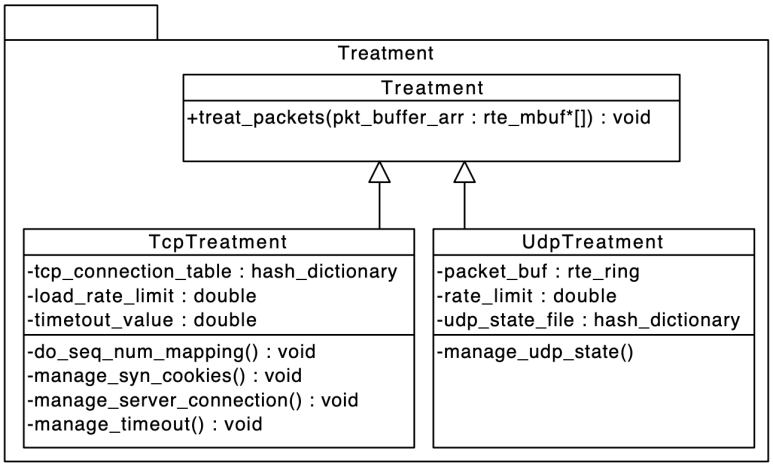

Abbildung 1.13: Altes Paket Treatments mit verschiedenen Klassen aus der Planungs- und Entwurfsphase

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 Umbenennung wird deutlich, dass es sich um eine Google-Densemap handelt und nicht um eine beliebige Map. Die Densemap untergliedert sich in drei Teile: `Data` ist der Key (vgl. Abb. 1.14), die Info ist die Nutzlast (vgl. Abb. 1.15) und der Hashwert aus `Data` ergibt die Position in der Map (vgl. Abb. 1.16). 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 Verbindungsaufbau 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 der `Inspection` 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. Dazu wird der Methode unter anderem ein Pointer auf ein `Data`-Objekt übergeben. Der Aufbau der Klasse `Data` ist in Abb. 1.14 genauer dargestellt. 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 der `Inspection` verschoben. Die Methode `manage_server_connection()` wurde mit der Methode `treat_packets()` konsolidiert, um auch hier Funktionsaufrufe einzusparen.

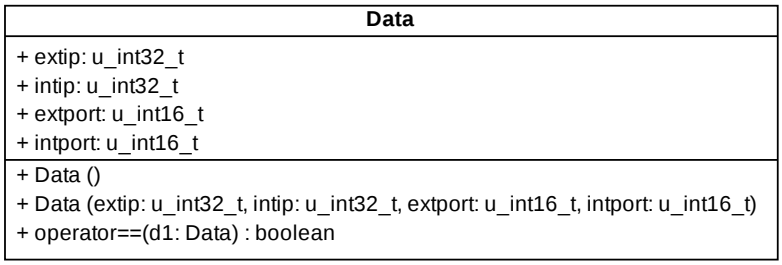

Abbildung 1.14: Klassendiagramm: `Data`

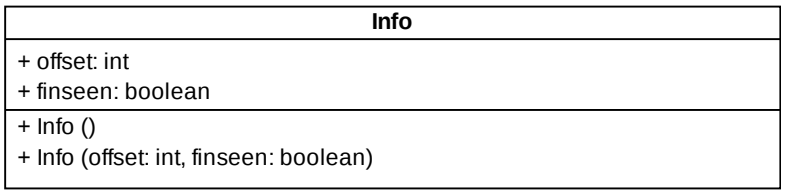

Abbildung 1.15: Klassendiagramm: `Info`

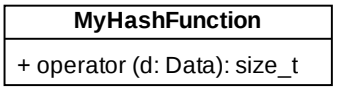

Abbildung 1.16: Klassendiagramm: `MyHashFunction`

Folgende Änderungen ergaben sich während der Validierungsphase:

Gemäß der Bennenungskonventionen wurde das Attribut `_timestamp` in `_s_timestamp` und die Methode `increment_timestamp()` in `s_increment_timestamp()` umbenannt.

Nach intensivem Testen der Realisierung mittels zwei Maps im Treatment stellte sich heraus, dass diese Realisierung aus performancetechnischer Sicht unvorteilhaft war. In Folge dessen wurden die Maps `_ackmap` und `_densemap` konsolidiert, sodass nunmehr insgesamt pro Verbindung ein Einfüge-, Such- und Löschvorgang gespart werden kann.

Die Methode `treat_packets()` wurde aufgeteilt: Die Funktion `treat_packets_to_inside()` bearbeitet diejenige Pakete, die als Ziel die internen Server haben. Die Funktion `treat_packets_to_outside()` bearbeitet diejenige Pakete, die als Ziel die externen Server haben.

Die Methode `create_cookie_secret()` wurde aufgrund der Kapselung in die Klasse `Rand` ausgelangert. Die dortige Methode `get_random_64bit_value()` ist statisch und verfügt über die gleiche Funktionalität.

Um die Kapselung der Klasse `Treatment` zu erhöhen, wurden außerdem alle Methoden bis auf `treat_packets_to_inside()`, `treat_packets_to_outside()` und `s_increment_timestamp()` auf `private` gesetzt. Diese drei sind weiterhin `public`, weil diese in der `main.cpp` und im `Thread` benötigt werden.