FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

SEP in Informatik

Audit von Firewall-Policys mit Hilfe von Netflow Analysen

Daniel Mentz

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

SEP in Informatik

Audit von Firewall-Policys mit Hilfe von Netflow Analysen

Bearbeiter:	Daniel Mentz
Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Dr. Helmut Reiser Claus Wimmer
Abgabedatum:	23. Januar 2008

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 23. Januar 2008

. . . .

(Unterschrift des Kandidaten)

Das Backbonenetz des MWN besteht aus einem Ring von Routern, welche über die drei Standorte Garching, TUM-Stammgelände und LMU verteilt sind. Alle Subnetze des MWN sind direkt oder indirekt über eine Schnittstelle an einen dieser Router angebunden. Mit Hilfe von Access Control Lists (ACLs) ist es möglich, für jedes VLAN-Interface direkt auf dem Router einen Paketfilter aufzusetzen und so das entsprechende Subnetz zu schützen. Ziel dieses Projekts ist es, die Access Control Lists der VLANs auszulesen und diese mit den NetFlow-Daten des selben Routers zu vergleichen. Es soll also im Nachhinein überprüft werden, ob nur durch die ACLs erlaubter Netzverkehr stattgefunden hat. Dies erfordert das Rekonstruieren des Netzverkehrs aus den NetFlow-Daten. Außerdem muss die Funktion der Firewall nachgebildet werden. Sollten in dem rekonstruierten Netzverkehr Verbindungen auftauchen, die durch die Firewall hätten blockiert werden müssen, so sollen diese Abweichungen protokolliert werden.

Ein Skript in der Programmiersprache Ruby, das die Funktion der Firewall nachbildet, wurde erfolgreich entwickelt. Der Vergleich mit den Netflow-Daten schlug aber leider fehl. Auf Grund der Tatsache, dass es sich um zustandsbehaftete Firewalls handelt, ist es erforderlich, den Netzverkehr zu rekonstruieren. Insbesondere muss ermittelt werden, welcher Kommunikationspartner eine Verbindung aufgebaut hat. Es wurde ein Lösungsvorschlag vorgestellt, der diese Rekonstruktion des Initiators leisten soll. Im Verlauf des Projekts hat sich jedoch herausgestellt, dass das vorgeschlagene Verfahren nur in etwa der Hälfte der Fälle eine richtige Ausgabe produziert. Grund dafür ist die Qualität der NetFlow-Daten. Die Zeitstempel, die dort zu finden sind, weisen eine Granularität von 64 Millisekunden auf, was sich als zu grob herausgestellt hat. Außerdem sind die Daten unvollständig, da einige Flows schlichtweg fehlen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Zum Schutz von eigenen Netzen vor Angriffen aus dem Internet werden häufig Firewalls eingesetzt. Aufgabe dieser Firewalls ist es, den Netzverkehr zu überwachen und potentiell gefährlichen Datenverkehr zu unterbinden [ESH 02]. Konfiguriert werden diese Firewalls mit Hilfe einer Liste von Regeln. Anhand dieser Regeln entscheidet eine Firewall, ob ankommende Datenpakete in das zu schützende Netz passieren dürfen.

In dieser Arbeit soll untersucht werden, inwieweit es möglich ist, aufgezeichnete Verkehrsdaten mit der Konfiguration einer Firewall zu vergleichen, um damit nachträglich zu überprüfen, ob tatsächlich nur legitime Verbindungen existiert haben. Zur Aufzeichnung wird dabei die NetFlow–Technologie[CS 07a] von Cisco verwendet. Router, die NetFlow unterstützen, aggregieren Pakete mit identischen Schlüsseleigenschaften zu sogenannten Flows, welche anschließend auf einem NetFlow Collector gespeichert werden.

Für jeden einzelnen Flow in den NetFlow–Daten ist zu entscheiden, ob er laut Firewall–Konfiguration erlaubt ist. Mit Hilfe einer Analysesoftware, die im Rahmen dieses Projekts entwickelt wird, soll dies automatisiert möglich sein.

Mit dieser Analysesoftware wäre es möglich, Fehlkonfigurationen in einem Netz festzustellen. Es ist zum Beispiel der Fall denkbar, dass durch ein fehlerhaft konfiguriertes Routing bestimmte Datenpakete gar nicht den Weg über die dafür vorgesehene Firewall nehmen, sondern über einen Umweg an der Firewall vorbei gelangen.

2 Einführung NetFlow

Möchte man Verkehrsdaten in IP–Netzen aufzeichnen, hat man verschiedene Möglichkeiten. Die einfachste Möglichkeit ist, jedes einzelne Paket, welches über das Interface eines Routers übertragen wird, zu speichern. Damit hätte man einen sehr detaillierten Mitschnitt des Netzverkehrs und hat für nachträgliche Analysen die beste Ausgangsbasis. Da diese Methode aber extrem hohe Anforderungen an die verfügbare Speicherkapazität stellt, kommt sie in Netzen mit hohem Verkehrsaufkommen kaum zum Einsatz. In einer leicht abgewandelten Version zeichnet man nur einen Teil jedes Pakets auf, also zum Beispiel nur die ersten 60 Bytes. Dieser Ansatz stellt jedoch auch hohe Ansprüche an die Kapazität des verwendeten Datenspeichers. Häufig ist man auch nicht an solch detailreichen Aufzeichnungen interessiert. In vielen Fällen verwendet man die Daten, um gewisse statistische Aussagen treffen zu können. Dabei wird dann nicht mehr jedes einzelne Paket betrachtet, sondern zum Beispiel lediglich die Summe der übertragenen Bytes. Die Daten aus vielen einzelnen Paketen werden somit zu einer einzelnen Information aggregiert.

Aus diesem Grund bietet es sich an, die Informationen gleich bei der Aufzeichnung und nicht erst bei der Auswertung zu aggregieren. Bei diesem Prozess gehen natürlich auch Informationen verloren. Ein Beispiel dafür, wie Informationen aggregiert werden können, ist ein Byte–Zähler eines Routers. Dieser Byte–Zähler summiert die Paketgrößen aller über ein Interface verschickter Pakete auf. Am Ende steht ein einzelner skalarer Wert zur Verfügung. Aus diesem Wert ist hinterher nicht mehr ersichtlich, auf wie viele Pakete sich die aufsummierte Anzahl an Bytes verteilt hat, oder wer Sender und Empfänger der Pakete war.

Bei der Entwicklung von NetFlow[CS 07a] hat sich der Netzausrüster Cisco für einen Mittelweg entschieden. Dieser soll in den folgenden Abschnitten erläutert werden.

2.1 Elemente eines Flows

Router, die NetFlows implementieren, fassen mehrere IP-Pakete zu einem so genannten Flow zusammenzufassen. Die Auswahl der IP-Pakete, die ein Flow umfasst, folgt ganz bestimmten Regeln. IP-Pakete, die zu einem Flow gehören, sind in den folgenden Eigenschaften identisch:

- Protokoll auf Transportschicht. (z.B. TCP oder UDP)
- IP–Adresse des Senders.
- Portnummer des Senders.
- IP–Adresse des Empfängers.
- Portnummer des Empfängers.
- Type of Service Feld im IP–Header.
- Netzschnittstelle des Routers, auf der das Paket empfangen wurde.
- Netzschnittstelle des Routers, über die das Paket weiter versendet wurde.

Nun werden aber nicht alle Pakete, welche in diesen acht Werten identisch sind, zu einem Flow zusammengefasst. Sie können auch auf mehrere Flows verteilt sein.

Jeder Router verwaltet eine Menge von aktiven Flows. Empfängt er ein Paket, das zu einem bereits vorhandenen Flow passt, so rechnet er es diesem Flow zu. Existiert kein passender Flow, so legt der Router einen neuen an. Die folgende Tabelle listet die Werte der Datenstruktur auf, die der Router für jeden Flow speichert.

Wert	Beschreibung
dpkts	Die Anzahl der Pakete, die zu diesem Flow gerechnet werden.
doctets	Die aufsummierten Längen der einzelnen Pakete in Bytes.
first	Der Zeitpunkt an dem das **erste** Pakets eines jeweiligen Flows empfangen wurde. (siehe weiter unten im Text)
last	Der Zeitpunkt an dem das **letzte** Paket eines Flows empfangen wurde. (siehe weiter unten im Text)
srcaddr	Die IP–Adresse des Senders.
dstaddr	Die IP–Adresse des Empfängers.
nexthop	IP–Adresse des Routers, an den die jeweiligen Pakete weitergeleitet werden.
input	Die Netzschnittstelle (als SNMP Schnittstellen–Index), auf dem die jeweiligen Pakete empfangen wurden.
output	Die Netzschnittstelle (als SNMP Schnittstellen–Index), über die die jeweiligen Pakete weiter verschickt wurden.
srcport	Die Portnummer, welche der Sender verwendet. (Nur gültig für TCP und UDP)
dstport	Die Portnummer des Empfängers. (Nur gültig für TCP und UDP)
prot	Das auf der Transportschicht (Schicht 4) verwendete Protokoll. (z.B. 6 für TCP oder 17 für UDP)
tos	Der Wert des Type of Service Felds der IP–Pakete.
tcp_flags	„Die aufsummierten TCP–Flags“. Es handelt sich hierbei um den Wert, der sich ergibt, wenn man die TCP–Flags aller TCP–Segmente mit der logischen Oder–Funktion verknüpft. Der vorhergehende Satz in Anführungszeichen ist deswegen nicht ganz korrekt, weil bei der Berechnung des Wertes für tcp_flags als Verknüpfungsfunktion nicht die Addition, sondern das logische Oder verwendet wird. Ist in diesem Feld zum Beispiel das Bit für das SYN–Flag auf 1 gesetzt, so bedeutet dies nur, dass in mindestens einem TCP–Segment das SYN–Flag gesetzt war. Die Information, in wie vielen bzw. in welchen Paketen dies der Fall war, geht verloren.
src_mask	Netzmaske der Quelladresse
dst_mask	Netzmaske der Zieladresse
src_as	Nummer des autonomen Systems, in dem sich der Sender befindet.
dst_as	Nummer des autonomen Systems, in dem sich der Empfänger befindet.

Die Werte first und last beschreiben Zeitpunkte. Diese werden durch die Anzahl an Millisekunden beschrieben, die seit einem bestimmten zeitlichen Referenzpunkt verstrichen sind. In der Regel wird dafür der Zeitpunkt des Hochfahrens des Routers verwendet. Diese Art der Darstellung ermöglicht es, recht einfach den zeitlichen Abstand zwischen zwei Ereignissen zu berechnen.

Solange der Router diese Datenstruktur für einen aktiven Flow verwaltet, sind die dort gespeicherten Werte noch nicht endgültig. Mit jedem Eintreffen eines passenden Pakets, wird diese Datenstruktur aktualisiert. So wird zum Beispiel die Länge des Pakets auf die Variable `doctets` aufaddiert. Auch die Variable `last` wird jedes Mal aktualisiert und enthält damit immer den Zeitpunkt, zu dem das vorläufig letzte Paket empfangen wurde.

2.2 Export von Flows

Die Flows, die ein Router zu einem bestimmten Zeitpunkt im Speicher hält, werden aktive Flows genannt. Ein aktiver Flow wird aktualisiert, wenn ein Paket auftaucht, welches zu diesem Flow passt. Die Umstände, unter denen ein Flow exportiert wird, werden in Abschnitt 2.5 aufgeführt. Dabei können bis zu 30 Flows zusammengefasst werden, die dann zu einem so genannten flow collector exportiert werden. Dieser Export geschieht meist unter Verwendung von UDP als Schicht–4 Protokoll. Da UDP kein zuverlässiges Protokoll ist, werden die einzelnen Pakete mit Sequenznummern versehen. Damit wird es dem flow collector ermöglicht, festzustellen, ob bei der Übertragung Flows verloren gegangen sind. Allerdings werden diese im Falle eines Verlustes nicht erneut übertragen, sondern sind endgültig verloren. Diese Tatsache gilt es bei der Auswertung zu berücksichtigen.

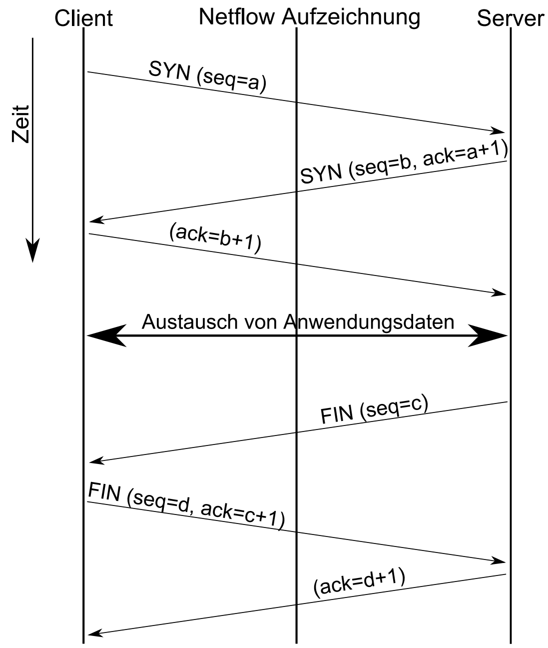

Abbildung 2.1: Der zeitliche Verlauf einer TCP–Verbindung

2.3 Der Zusammenhang zwischen Flows und TCP–Verbindungen

Es ist wichtig zu verstehen, dass Flows lediglich unidirektionale Paketströme beschreiben und **nicht** identisch mit TCP–Verbindungen sind.

Im Folgenden wird untersucht, welche „Spuren“ eine TCP–Verbindung hinterlässt, beziehungsweise welche Eigenschaften die Flows haben, die sie induziert.

2.3.1 TCP–Verbindungsauf- und abbau

Abbildung 2.1 zeigt den Ablauf einer TCP–Verbindung.[Stev 93] Der Client initiiert die Verbindung, in dem er ein Paket mit gesetztem SYN–Flag aussendet. Sofern der Server dem Verbindungswunsch entsprechen möchte, quittiert er den Empfang durch Aussendung eines Pakets mit gesetztem SYN– und gesetztem ACK–Flag. Sobald der Client dieses Paket wiederum mit einem Paket mit gesetztem ACK–Flag quittiert, gilt die Verbindung als aufgebaut. Für einen erfolgreichen Verbindungsaufbau sind demnach drei Pakete notwendig. Daher auch der Name Drei–Wege–Handshake. Natürlich kann es passieren, dass bei der Übertragung Pakete verloren gehen. Diesem Problem begegnen die Kommunikationspartner damit, dass sie die entsprechenden Pakete nochmals senden, sofern die Antwort darauf für eine gewisse Zeit ausgeblieben ist.

Der zeitliche Abstand, zwischen Aussendung eines Pakets und Empfang der Antwort, hängt ab von dem Netz, über das die Pakete versendet werden, und von dem Standort der Kommunikationspartner im Netz. Die Werte können für Verbindungen, die über den ganzen Globus gehen bis zu 500 Millisekunden betragen. In lokalen Netzen kommen häufig Werte unterhalb von einer Millisekunde vor.

Sobald die Verbindung steht, kann der Datenaustausch zwischen Client und Server stattfinden. In welchen zeitlichen Abständen dabei Pakete ausgetauscht werden, hängt von der Anwendung ab. Bei einer Anwendung wie Dateiübertragung (FTP) oder E-Mail (SMTP) wird die Senderate häufig nur durch die Bandbreite des Netzes begrenzt. Hier tauchen Pakete also in recht kurzen Abständen auf. Anders verhält es sich bei Anwendungen mit

Benutzerinteraktivität wie beispielsweise Fernsteuerungen jeglicher Art (z.B. SSH, VNC), Online–Gaming– oder Chat–Anwendungen, aber auch E-Mailanwendungen, die das IMAP–Protokoll verwenden.

Der Verbindungsabbau sieht ähnlich aus wie der Verbindungsaufbau. Die Seite, welche die Verbindung abbauen möchte, sendet ein Paket mit gesetztem FIN–Flag. Der Empfänger bestätigt daraufhin den Empfang des Pakets und somit den Wunsch, die Verbindung abzubauen. Zu beachten ist hierbei, dass TCP die Möglichkeit bietet, jede Richtung der Verbindung einzeln abzubauen. Es kann also die Situation eintreten, dass eine TCP–Verbindung nur noch in eine Richtung geöffnet ist. Zum vollständigen Abbau müssen also beide Kommunikationspartner FIN–Pakete aussenden, welche vom entsprechenden Gegenüber bestätigt werden müssen.

2.4 Gemeinsame Eigenschaften von TCP–Segmenten

Alle TCP–Pakete, die in eine Richtung gesendet werden, also vom Client zum Server oder umgekehrt, sind in den folgenden Eigenschaften identisch:

- IP–Adresse des Senders.
- Portnummer des Senders.
- IP–Adresse des Empfängers.
- Portnummer des Empfängers.
- Type of Service Feld im IP–Header.

Wenn man davon ausgeht, dass sich die Forwarding–Table eines Routers während einer Verbindung nicht ändert, sind alle TCP–Pakete einer Richtung in den Eigenschaften identisch, welche den Schlüssel eines Flows ausmachen. Somit werden all diese Pakete potentiell zu einem Flow zusammen gefasst.

Ähnlich verhält es sich bei UDP. Zwar gibt es bei diesem Protokoll keine Verbindungen in diesem Sinne, jedoch verwenden Clients für mehrere aufeinanderfolgende Anfragen an ein und denselben Server in der Regel identische Kombinationen aus oben genannten Werten. (Portnummern, etc.) Somit werden für jede Kommunikationsbeziehung auf UDP–Basis mindestens zwei Flows entstehen: Ein Flow für die Anfragen an den Server und ein zweiter für die Antworten an den Client.

2.5 Induzierte Flows

Im einfachsten Fall induziert eine TCP–Verbindung also zwei Flows: einen in Hinrichtung und einen in Rückrichtung. Im Idealfall lässt sich durch einen Vergleich der first–Werte [1] der zwei Flows bestimmen, welche Gegenseite die Rolle des Clients und welche die des Servers spielt. Der Client ist derjenige, der das erste Paket aussendet. Dieses SYN–Paket taucht zeitlich vor der Antwort des Servers, also vor dem ersten Paket des Servers, auf. Somit sollte der Wert first des Flows in Richtung Client einen zeitlich späteren Wert aufweisen als der Flow in die entgegengesetzte Richtung.

Nun gibt es aber eine Reihe von Möglichkeiten, warum eine TCP–Verbindung mehr oder weniger als zwei Flows induziert. Werden über eine TCP–Verbindung für eine gewisse Zeit keine Daten übertragen, so tauchen auch keine Pakete mit entsprechenden Portnummern und IP–Adressen auf. Der Router wird dann nach einer gewissen Zeit (in der Regel 5 Minuten) den Flow exportieren. Wenn die TCP–Verbindung nach dieser Zeit wieder aktiv wird, legt der Router einen neuen aktiven Flow an, der, was die Schlüsselfelder (Quell–IP–Adresse, Quellport, Ziel–IP–Adresse, Zielport, etc.) betrifft, mit dem bereits exportierten Flow identisch ist.

Mehrere Flows pro Richtung werden auch dann induziert, wenn über eine TCP–Verbindung längere Zeit (mehr als 32 Minuten) kontinuierlich Daten übertragen werden. Auch dann exportiert der Router vorzeitig den entsprechenden Flow und legt einen neuen Flow an, sobald ihn ein neues Paket erreicht, das eigentlich noch zu dem alten Flow gepasst hätte.

[1] Der Wert first gibt Auskunft über den Zeitpunkt, zu dem das erste Paket eines Flows aufgetreten ist. Siehe auch Seite 3

Auch wenn der Cache des Routers vollläuft, und deshalb Flows vorzeitig exportiert werden müssen, können eine Vielzahl von Flows auftreten, die zu ein und derselben Verbindung gehören.

Wie weiter oben beschrieben, kann auch der Fall eintreten, dass Flows verloren gehen. Das erklärt, warum zu einer TCP–Verbindung nur Flows in eine Richtung auftreten.

Grundsätzlich stellt die Verteilung auf mehrere Flows keine Schwierigkeit dar. Flows mit identischen Schlüsseln werden bei der Analyse einfach gruppiert. Dadurch erhält man dann sogar ein Mehr an Information, da die Daten — zumindest etwas — feiner aufgelöst sind. Möchte man beispielsweise die Datenübertragungsrate eines Datenstroms bestimmen, so kann man im Falle von einem Flow pro Richtung nur einen Durchschnittswert berechnen. Verfügt man hingegen über mehrere Flows in eine Richtung, so kann man für jeden Flow individuell den Durchschnitt berechnen und erhält somit etwas mehr an Information.

3 Szenario

Das Backbonenetz des MWN besteht aus einem Ring von Routern, welche über die drei Standorte Garching, TUM–Stammgelände und LMU verteilt sind (siehe Abbildung 3.1). Alle Subnetze des MWN sind direkt oder indirekt über eine Schnittstelle an einen dieser Router angebunden. Für die Mehrzahl der Subnetze verfügen die Router jedoch über ein dediziertes VLAN–Interface.

Mit Hilfe von Access Lists (ACLs) ist es möglich, für jedes VLAN–Interface direkt auf dem Router einen Paketfilter aufzusetzen und so das entsprechende Subnetz zu schützen. Dies hat den Vorteil, dass die Betreiber der einzelnen Subnetze damit ohne eigene Firewall auskommen. Ziel dieses Projekts ist es, die Access Lists der VLANs auszulesen und diese mit den NetFlow–Daten des selben Routers zu vergleichen. Sollten sich hier Abweichungen ergeben, so sollen diese protokolliert werden. Unter Abweichungen wird hier Datenverkehr verstanden, der laut ACLs eigentlich unterbunden werden sollte. Abbildung 3.2 veranschaulicht dies. [mwn 06]

3.1 Access Lists (ACLs)

Eine Access List besteht aus einer Liste von Regeln, die für jedes Paket abgearbeitet wird. Jede Regel besteht aus einem Muster, welches mit dem Paket verglichen wird, und einer Aktion, die ausgeführt wird, sofern das Paket auf das Muster passt. Die zwei möglichen Aktionen sind „Lasse das Paket passieren“ (permit) und „Verwerfe das Paket“ (deny). Sobald eine Regel gefunden wurde, dessen Muster auf das Paket passt, wird die entsprechende Aktion ausgeführt und die Abarbeitung der Liste beendet. Nachfolgende Regeln werden nicht mehr berücksichtig.

Ein Muster besteht aus verschiedenen Bedingungen, die sich auf einzelne Werte des betrachteten Pakets beziehen. Die folgende Liste gibt eine Übersicht über die Werte, welche mit Mustern verglichen werden können.

- Schicht–4–Protokoll
- IP–Adresse des Senders
- IP–Adresse des Empfängers
- Portnummer auf Senderseite (nur für TCP und UDP)
- Portnummer auf Empfängerseite (nur für TCP und UDP)

Die Muster bestehen entweder aus einem einzelnen Wert oder aus einer Menge von Werten. Die Portnummer eines Pakets kann zum Beispiel mit einer einzelnen Portnummer oder mit einem Intervall von Portnummern verglichen werden.

Die Vergleichswerte für IP–Adressen sind analog entweder einzelne IP–Adressen oder ein Subnetz von IP–Adressen. Dieses Subnetz wird über die Netzadresse inklusive Subnetzmaske beschrieben. Das folgende Beispiel soll dies verdeutlichen:

```
permit tcp 129.187.12.0 0.0.0.255 eq 22 129.187.48.0 0.0.0.255
```

Diese Regel lässt IP–Pakete passieren (permit), welche die folgenden Eigenschaften haben.

- Auf Schicht–4 wird TCP verwendet
- Die Adresse des Senders stammt aus dem Netz `129.187.12.0/24`
- Die Portnummer auf Senderseite ist 22

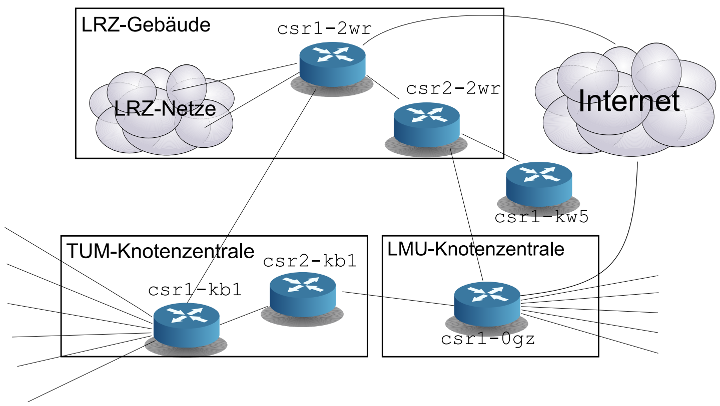

Abbildung 3.1: Die Netzstruktur des MWN

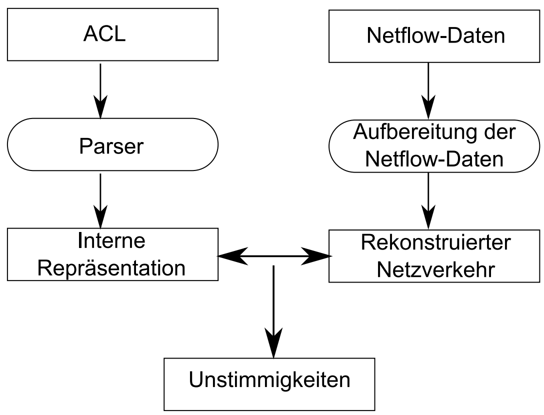

Abbildung 3.2: Die grundlegende Idee des Projekts.

- Die Adresse des Empfängers stammt aus dem Netz `129.187.48.0/24`

Die Portnummer auf Empfängerseite wird in diesem Beispiel nicht eingeschränkt und ist somit beliebig.

Besonders zu beachten ist die etwas eigenwillige Darstellung von Netzmasken in Ciscos ACLs. Diese werden als Einerkomplement der sonst üblichen Darstellung beschrieben. So muss man beispielsweise die Maske `0.0.0.255` anstatt `255.255.255.0` verwenden, um ein Class C Netz zu beschreiben.

Die bisher beschriebenen Regeln sind statische Regeln. Dies bedeutet, dass jedes empfangene Paket für sich betrachtet wird. Mit Cisco IOS Software Release 11.3 wurden sogenannte reflexive ACLs eingeführt [CS 07b]. Diese ACLs ermöglichen es, eingehenden Verkehr, der von einem Rechner innerhalb des zu schützenden Netzes initiiert wurde, zu gestatten. Dafür werden die notwendigen Regeln den ACLs dynamisch hinzugefügt.

Konfiguriert wird diese Funktion durch korrespondierende Regeln in den ACLs für den ein- und ausgehenden Verkehr. Das folgende Beispiel wurde aus der Access List des VLAN 23 extrahiert. VLAN 23 besitzt die zwei Netzadressen `129.187.12.0/24` und `129.187.15.0/24`.

```
permit ip 129.187.12.0 0.0.0.255 any reflect MITARBEITER_REFLEX
```

Das Pendant in der ACL für eingehenden Verkehr sieht folgendermaßen aus:

```
evaluate MITARBEITER_REFLEX
```

Kritisch für dieses Projekt ist, dass sich die ACLs mit dem Empfang von Paketen verändern. Initiiert ein Rechner innerhalb des zu schützenden Netzes eine Verbindung mit einem Rechner außerhalb dieses Netzes, so fügt die Firewall der ACL für eingehenden Verkehr automatisch eine Regel hinzu. Diese neue Regel gewährleistet, dass die Pakete in Gegenrichtung, also die „Antwortpakete“, ungehindert passieren können. Um ein vollständiges Bild der zum Zeitpunkt x gültigen ACLs zu erlangen, muss die Funktion der Firewall praktisch simuliert werden. Es reicht nicht aus, lediglich die durch den Administrator konfigurierten ACLs zu betrachten. Vielmehr muss zusätzlich der aufgezeichnete Netzverkehr untersucht werden, um die dynamisch hinzugefügten Regeln rekonstruieren zu können.

4 Simulation der Paketfilter

Wie in Kapitel 3 beschrieben, sollen die NetFlow–Daten mit den Access Lists verglichen werden. Dies bedeutet, dass die Paketfilter, die durch die Access Lists konfiguriert werden, nachgebildet werden müssen. Das heißt, für jede rekonstruierte Verbindung muss überprüft werden können, ob diese die Firewall hätte passieren dürfen.

Für dieses Projekt wurde ein Skript (Listing A.5) in der Sprache Ruby entwickelt, das genau diese Nachbildung leistet. Dieses Skript stellt eine Funktion zur Verfügung, der folgende Parameter übergeben werden:

- Schicht–4–Protokoll (TCP oder UDP)
- Quell–IP–Adresse
- Quellport
- Ziel–IP–Adresse
- Ziel–Port

Auf der Basis dieser Werte, entscheidet die Funktion, ob ein entsprechendes IP–Paket die Firewall passieren darf. Dabei greift das Skript auf eine MySQL–Datenbank (Listing A.3) zurück, in die zuvor die ACLs importiert wurden (Listing A.4).

Die auf ein gegebenes Paket anzuwendende Aktion („permit“ oder „deny“) soll durch ein einziges SQL–Kommando abgefragt werden können. Die Parameter, die diesem Kommando übergeben werden, sind im wesentlichen die bereits genannten Parameter, die ein IP–Paket ausmachen (Schicht–4–Protokoll, Quell–IP–Adresse, etc.)

MySQL unterstützt die Verarbeitung von IP–Adressen. So steht zum Beispiel die Funktion `INET_ATON()` zur Verfügung, die eine IP–Adresse in „dotted quad“–Notation in einen Integer umwandelt. Beim Abarbeiten der einzelnen Regeln einer Access Control List muss aber häufig geprüft werden, ob eine IP–Adresse in einem bestimmten Subnetz liegt. So soll beispielsweise überprüft werden, ob die Adresse `192.168.55.62` im Subnetz `192.168.10.0/24` liegt. Einen solchen Vergleich von IP–Adressen mit Netzadressen in CIDR–Notation unterstützt MySQL nicht, weshalb hier auf eine andere Lösung ausgewichen wurde: Für jedes Subnetz wird die kleinste und größte IP–Adresse gespeichert. Um bei dem Beispiel des Subnetzes `192.168.10.0/24` zu bleiben, werden in diesem Fall die Adressen `192.168.10.0` und `192.168.10.255` gespeichert. Alle Adressen, die in diesem Intervall liegen, gehören auch zu dem Subnetz `192.168.10.0/24`.

Ähnliche Schwierigkeiten sind bei der Abbildung von Bedingungen bezüglich Portnummern aufgetreten. Ciscos ACLs erlauben folgende Arten von Bedingungen:

- Die Portnummer muss gleich einem bestimmten Wert sein. (z.B. „eq 21“)
- Die Portnummer muss kleiner als ein bestimmter Wert sein. (z.B. „lt 1024“)
- Die Portnummer muss größer als ein bestimmter Wert sein. (z.B. „gt 1024“)
- Die Portnummer muss ungleich einem bestimmten Wert sein. (z.B. „neq 22“)
- Die Portnummer muss in einem gegebenen Intervall liegen. (z.B. „range 5000 6000“)
- Die Portnummer kann beliebig kein. (Port wird nicht spezifiziert)

Um einfache SQL–Abfragen zu ermöglichen, werden all diese verschiedenen Arten von Bedingungen zu einem Intervall konvertiert. Die folgende Tabelle soll dies veranschaulichen:

ACL–Syntax	Intervall
eq x	$[x; x]$
lt x	$[0; x - 1]$
gt x	$[x + 1; 65535]$
neq x	$[0; x - 1] \cup [x + 1; 65535]$
range x y	$[x; y]$
(Port nicht spezifiziert)	$[0; 65535]$

Durch diese Abbildung können Anfragen nun einfacher durchgeführt werden. Die sechs verschiedenen Arten von Bedingungen wurden auf eine reduziert: Es muss nur noch überprüft werden, ob die Portnummer in dem beschriebenen Intervall liegt. Die entsprechende Abfrage lässt sich somit viel einfacher in SQL formulieren.

Die folgende Tabelle listet die einzelnen Spalten der Sicht [1]der Datenbank auf, die die Regeln bzw. ACEs (Access Control Entries) widerspiegelt. Auf dieser Sicht werden die Abfragen durchgeführt.

Spalte	Beschreibung
firewalls.name	Name der Firewall (sofern mehrere Firewalls betrachtet werden)
acl_name	Name der Access Control List
rule_no	Fortlaufende Nummer des Access Control Entries
type	„permit“ oder „deny“
protocol	z.B. TCP oder UDP
srcStartIP	*(siehe Text)*
srcEndIP	*(siehe Text)*
srcMinPort	*(siehe Text)*
srcMaxPort	*(siehe Text)*
dstStartIP	*(siehe Text)*
dstEndIP	*(siehe Text)*
dstMinPort	*(siehe Text)*
dstMaxPort	*(siehe Text)*

Eine Regel wird auf ein IP–Paket angewendet, wenn folgende Bedingungen erfüllt sind:

- Das verwendete Schicht–4–Protokoll entspricht dem Wert in der Spalte `protocol`
- Die IP–Adresse des Senders liegt im Intervall $[srcStartIP; srcEndIP]$
- Die IP–Adresse des Empfängers liegt im Intervall $[dstStartIP; dstEndIP]$
- Die Portnummer des Senders liegt im Interval $[srcMinPort; srcMaxPort]$
- Die Portnummer des Empfängers liegt im Interval $[dstMinPort; dstMaxPort]$
- `rule_no` besitzt den kleinsten Wert aller passender Access Control Entries

Mit Hilfe der folgenden Abfrage wird überprüft, ob ein TCP–Paket vom Endpunkt `212.201.100.133:80` zum Endpunkt `129.187.12.43:59536` passieren darf.

```
SELECT type FROM rules WHERE name='csr1-2wr' AND acl_name='input'
  AND srcStartIP <= INET_ATON('212.201.100.133')
  AND srcEndIP >= INET_ATON('212.201.100.133')

  AND dstStartIP <= INET_ATON('129.187.12.43')
  AND dstEndIP >= INET_ATON('129.187.12.43')

  AND protocol='tcp'

  AND srcMinPort <= 80 AND srcMaxPort >= 80
  AND dstMinPort <= 59536 AND dstMaxPort >= 59536

  ORDER BY rule_no LIMIT 1
```

[1]Unter dem Begriff Sicht (engl. View) wird hier eine logische Relation verstanden.

Interessant sind die Schlüsselwörter `ORDER BY` und `LIMIT` in der letzten Zeile. Diese sorgen dafür, dass alle passenden Regeln nach der fortlaufenden Nummer der Regel sortiert werden und aus dieser Liste nur die erste Regel ausgewählt wird. Damit wird die Funktion der Firewall exakt nachgebildet: Die jeweils erste Regel, die auf ein Paket passt, wird ausgeführt. Alle nachfolgenden Regeln werden ignoriert. Das Ergebnis, das diese Anfrage liefert, ist entweder „permit“ oder „deny“.

5 Analyse der NetFlow–Daten vom Münchner Wissenschaftsnetz

Für diese Arbeit wurden vom Leibniz–Rechenzentrum (LRZ) NetFlow–Daten des Münchner Wissenschaftsnetzes zur Verfügung gestellt. Diese wurden von dem Router `csr1-2wr` im NetFlow–Format der Version 5 exportiert und an den folgenden Tagen aufgezeichnet:

- Donnerstag, 26. Juli 2007
- Samstag, 1. Dezember 2007 bis Dienstag 18. Dezember 2007

Vor der Weiterverabeitung sollten diese Daten zuerst auf ihre Qualität hin untersucht werden. Von besonderem Interesse war die Vollständigkeit der Aufzeichnung sowie die Qualität der Zeitstempel.

5.1 flow–tools

Flow–tools ist eine Sammlung von Werkzeugen, die NetFlow–Daten aufzeichnen, verarbeiten, filtern und verschiedene Formen von Berichten erstellen können. Das LRZ verwendet zur Aufzeichnung `flow-capture`, das ebenfalls aus dieser Werkzeugsammlung stammt. Somit hat es sich angeboten, auch für dieses Projekt die flow–tools zu verwenden, da so die Kompatibilität mit den zur Verfügung stehenden Daten gewährleistet ist.

Das wichtigste Werkzeug für diese Arbeit bildete `flow-export`. Es ermöglicht die Konvertierung der durch `flow-capture` aufgezeichneten Daten in ein CSV [1]–Format. Nutzt man zur Auswertung der Daten eine Skriptsprache, so lässt sich dieses CSV–Format erheblich einfacher importieren als das von `flow-capture` verwendete binäre und propreitäre Format.

`flow-header` liest die Metadaten einer NetFlow–Datei aus. Zu diesen Metadaten gehört zum Beispiel der Anfangs– und Endzeitpunkt der Aufzeichnung, sowie die Anzahl der aufgezeichneten Flows. Interessant für die Arbeit ist vor allem die Anzahl der verlorengegangenen Flows. Wie weiter oben beschrieben, ist es dem NetFlow–Collector möglich, mit Hilfe von Sequenznummern in den NetFlow–UDP–Datagrammen zu erkennen, ob und wie viele Flows aufgrund von Übertragungsfehlern verloren gegangen sind.

5.2 Zeitstempel

Wie auf Seite 3 beschrieben verwendet NetFlow Zeitstempel, welche in Millisekunden angegeben werden, um den Zeitpunkt des ersten und des letzten Pakets eines Flows zu beschreiben. Nun ist es aber bei Computersystemen häufig so, dass die verwendete Hardwareuhr keine entsprechend hohe Auflösung bietet sondern wesentlich langsamer tickt und somit gröbere Zeitwerte liefert. Die Software rechnet diese groben Werte dann in eine feinere Skala um und gaukelt dem Anwender eine feinere Auflösung vor. Dies führt dazu, dass die Werte der Zeitstempel nicht monoton mit fortschreitender Zeit ansteigen, sondern eine Stufenfunktion aufweisen. Die Zeitwerte müssen deshalb bei der Analyse mit einer gewissen Vorsicht verwendet werden.

Um herauszufinden, welche Granularität die Zeitwerte in den NetFlow– Daten besitzen, wurde wie folgt vorgegangen: Die Daten wurden in Blöcke unterteilt, die jeweils einen Intervall von fünf Minuten Aufzeichnungsdauer widerspiegeln. Aus diesen Blöcken wurden alle first–Zeitstempel der NetFlow–Records extrahiert und untersucht, in welche Restklasse bezüglich modulo 64 diese fallen. Nimmt man an, dass die Ereignisse alle zu einem zufälligen Zeitpunkt auftreten, so müssten die Zeitstempel auf die Restklassen annähernd gleichverteilt

[1]Comma–Separated Values

sein. Die Zahl 64 ist tatsächlich ein Erfahrungswert. Das beschriebene Experiment wurde auch mit anderen Werten durchgeführt. Jedoch brachten nur die Zahl 64 und Vielfache von 64 das beobachtete Ergebnis.

Als Beispiel wurden die ersten fünf Minuten nach Mitternacht am 26.07.07 gewählt. Aus diesem Zeitintervall wurden alle 969934 Zeitstempel untersucht, die den Zeitpunkt des ersten Pakets eines jeden Flows bestimmen. Abbildung 5.1 zeigt die Verteilung der Zeitstempel auf die Restklassen. Auffällig ist hierbei, dass lediglich sechs Restklassen eine Häufigkeit von mehr als Null aufweisen. Alle anderen Restklassen tauchen gar nicht auf. Dadurch wird deutlich, dass die Zeitwerte eine Auflösung von etwa 64 Millisekunden besitzen. Diese Tatsache muss bei der Verwendung für Analysezwecke berücksichtigt werden.

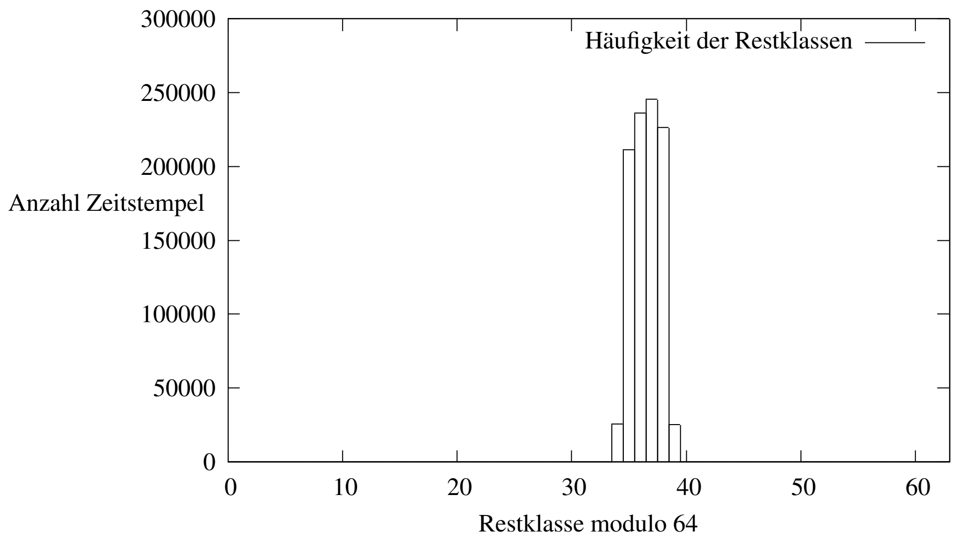

Abbildung 5.1: Verteilung der Zeitstempel auf die Restklassen modulo 64.

Dieses Experiment wurde auch mit anderen Zeitintervallen wiederholt und ergab dabei identische Ergebnisse: Alle Zeitwerte fallen in sechs benachbarte Restklassen. Lediglich die Zahlenwerte der Restklassen änderten sich. Dies lässt vermuten, dass die Auflösung nicht exakt 64 Millisekunden beträgt, sondern davon leicht abweicht.

Sortiert man die Zeitstempel und sieht man sich die einzelnen Zahlenwerte an, so kann man einzelne „Bursts“ von etwa 6 Zeitstempeln erkennen. So tauchen in einer Aufzeichnung zum Beispiel die Werte 50,51,52,53,54 und 55 auf. Danach herrscht für 59 Millisekunden Stille bevor die Werte 114,115,116,117,118 und 119 auftreten. Diese „Bursts“ treten also periodisch mit einer Periodendauer von 64 Millisekunden auf.

Aufgrund dieser Beobachtungen werden im weiteren Verlauf der Arbeit alle Zeitstempel auf ein Vielfaches von 64 Millisekunden gerundet.

5.3 Verlorengegangene Flows

Mit Hilfe des oben erwähnten Werkzeuges `flow-header` ist es möglich, aus den Metadaten der NetFlow–Daten die Anzahl an verlorengegangenen Flows zu extrahieren. Im Durchschnitt ging etwa jeder 2000ste Flow verloren. Um zu überprüfen, ob dieser Anteil eventuell von der Tageszeit oder dem Wochentag abhängig ist, wurden die Daten wieder in Blöcke unterteilt, die jeweils eine Stunde Aufzeichnungsdauer widerspiegeln. Dabei ließ sich aber kein Trend ablesen. Die Verlustrate ist weitgehend unabhängig von Tageszeit und Wochentag.

5.4 TCP–Flags

Da die zur Verfügung stehenden NetFlow–Daten in der NetFlow Version 5 vorliegen, sollten diese ein Feld für die TCP–Flags besitzen. Wie auf Seite 3 beschrieben, enthält dieses Feld einen Wert, der durch die ODER–Verknüpfung der TCP–Flags aller zu diesem Flow gehörenden Pakete gebildet wird. Mit Hilfe dieser Information könnte man zum Beispiel feststellen, ob ein gegebener Flow den ersten Flow einer TCP–Verbindung darstellt. Diese könnte man an einem vorhandenen SYN–Flag feststellen. Weiterhin wäre ein gesetztes RST–Flag ein Hinweis darauf, dass ein Verbindungsaufbau abgewiesen wurde.

Das LRZ setzt jedoch im Backbone–Bereich Router der Serie Cisco Catalyst 6500 ein. Da diese Modelle die Aufzeichnung von TCP–Flags nicht unterstützen, enthält das entsprechende Feld immer den Wert 0. [CS 07c]

5.5 Verlust des jeweils ersten Pakets eines Flows

Eine weitere Besonderheit der Cisco Catalyst 6500 Router ist das Multilayer Switching. Das erste Paket eines Flows wird durch die Multilayer Switch Feature Card (MSFC) verarbeitet. Diese trifft die Entscheidung, an welches Netz das Paket weitergeleitet wird und sendet es anschließend an die Supervisor Engine zurück. Damit alle weiteren Pakete des selben Flows schneller vermittelt werden können, wird direkt auf dem Switch ein Layer–3–Flow im Multilayer–Switching–Cache eingerichtet. Dies hat den Vorteil, dass alle folgenden Pakete nicht den Umweg über die Multilayer Switch Feature Card nehmen müssen, sondern direkt vom Switch vermittelt werden können.

Die Aufzeichnung der NetFlow–Daten geschieht allerdings nur auf dem Switch. Alle Pakete, die durch die MSFC weitergeleitet werden, also die jeweils ersten Pakete der Flows, werden nicht berücksichtigt. [CS 05]

Durch diesen Umstand enthalten die zur Verfügung stehenden Daten einen systematischen Fehler. Die Summe der übertragenen Bytes und die Anzahl der übertragenen Pakete ist nicht korrekt. Die Anzahl der Pakete müsste also um die Zahl eins korrigiert werden. Die Summe der in Wahrheit übertragenen Bytes lässt sich wohl nicht mehr rekonstruieren, da die Größe des ersten Pakets variieren kann. Die Größe eines TCP–SYN–Pakets hängt zum Beispiel von den verwendeten TCP–Optionen ab.

Hinzu kommt, dass mancher Verkehr in den NetFlow–Daten vollkommen unsichtbar bleibt. Kommunikationsbeziehungen, die lediglich aus einem Paket in jede Richtung bestehen, werden in den NetFlow–Daten nicht auftauchen. Eine solche Kommunikationsbeziehung könnte zum Beispiel die Anfrage an einen DNS–Server sein. Erfolgt diese Anfrage über UDP, so besteht sie lediglich aus einem Paket in jede Richtung: Der Anfrage und der Antwort.

6 Rekonstruktion der TCP–Verbindungen

Die betrachteten Firewalls bzw. Paketfilter der Backbone–Router im MWN setzen die Technik der „Stateful Packet Inspection“ ein. Das bedeutet, dass die empfangenen Pakete nicht unabhängig von einander betrachtet werden. Die Entscheidung, was mit einem bestimmten Paket passiert, wird nicht alleine aufgrund des Inhalts des Pakets gefällt. Vielmehr spielt es eine erhebliche Rolle, welche Pakete zuvor über die Leitung gegangen sind.

Trennt eine Firewall beispielsweise ein zu schützendes Netz vom Internet, so darf ein Paket aus dem Internet in der Regel nur dann in das zu schützende Netz passieren, wenn zuvor ein Rechner aus dem zu schützenden Netz dieses Paket angefordert hat. Hier wird deutlich, dass ein Paket immer im Zusammenhang mit den zuvor weitergeleiteten Paketen gesehen werden muss.

6.1 Resultierende Anforderungen an die Analysesoftware

Dadurch, dass eine Firewall Pakete im Kontext der vorhergehenden Pakete betrachtet, muss eine Analyse–Software dies ebenfalls tun. Wie beschrieben, fasst ein Flow Pakete, die in die gleiche Richtung gesendet wurden, zusammen. Von daher reicht es nicht aus, jeden Flow isoliert von den anderen Flows zu betrachten. Genauso wie eine Firewall Pakete in zwei entgegengesetzte Richtungen korreliert, muss eine Analyse–Software auch die daraus resultierenden Flows in Zusammenhang bringen.

Wir wollen bei dem Beispiel bleiben, in dem eine Firewall ein mit dem Internet verbundenes Netz schützen soll. Betrachtet man einen Flow in Richtung des zu schützenden Netzes, so kann man alleine auf Grundlage der Attribute dieses Flows keine Aussage darüber treffen, ob dieser legitim ist oder nicht. Legitim wäre er zum Beispiel dann, wenn er durch eine TCP–Verbindung induziert wurde, welche ein Client aus dem zu schützenden Netz aktiv aufgebaut hat. Mit „aktiv“ ist in diesem Zusammenhang gemeint, dass der Client im zu schützenden Netz die Verbindung initiiert hat. Von daher muss eine Analyse–Software diesen Flow mit anderen relevanten Flows in Verbindung setzen.

6.2 Eigenschaften der Rekonstruktion

Diese Arbeit konzentriert sich in erster Linie auf TCP–Verbindungen. Es ist also Aufgabe, alle Flows, welche zu einer TCP–Verbindung gehören, zusammenzufassen und die TCP–Verbindung daraus zu rekonstruieren. Für unsere Zwecke muss diese Rekonstruktion aber nicht vollständig sein. Bestimmte Eigenschaften bzw. Charakteristika sind für unsere Zwecke **nicht** interessant. Dazu gehören zum Beispiel:

- Die Dauer der Verbindung
- Die Anzahl der übertragenen Pakete und Bytes
- Die erzielte Übertragungsgeschwindigkeit
- Der Inhalt der Verbindung
- Welche Seite die Verbindung **ab**gebaut hat

Von großem Interesse ist allerdings die Antwort auf die Frage, welche Seite die Verbindung aktiv aufgebaut, also initiiert hat. Diese wesentliche Eigenschaft gibt in den meisten Fällen darüber Auskunft, ob eine Verbindung laut Firewall–Konfiguration legitim war oder nicht. Viele Firewalls sind mit einem TCP–Automaten ausgestattet und sind so konfiguriert, dass sie lediglich Verbindungen zulassen, welche durch Rechner in dem zu schützenden Netz initiiert wurden.

6.3 Vorstellung des Verfahrens zur Rekonstruktion

Im Folgenden soll eine Idee vorgestellt werden, wie man aus den NetFlow–Daten den initiierenden Rechner bestimmen kann. Danach wird darüber berichtet, wie gut sich das beschriebene Verfahren in der Praxis bewährt hat.

Zwei TCP–Segmente gehören genau dann zu ein und derselben TCP–Verbindung, wenn sie sich in den Werten

- IP–Adresse des Senders,
- Portnummer, die auf der Senderseite verwendet wird,
- IP–Adresse des Empfängers,
- Portnummer, die auf der Empfängerseite verwendet wird,

identisch sind. Sie gehören auch dann zusammen, wenn die genannten Werte gleich sind, sofern man bei einem Paket die Rolle von Sender und Empfänger vertauscht. Also die Kombination aus IP–Adresse und Portnummer des Senders des einen TCP–Segments identisch ist mit der Kombination aus IP–Adresse und Portnummer des Empfängers des anderen TCP–Segments.

Das beschriebene Vier–Tupel bleibt während einer TCP–Verbindung konstant. Der Rechner, der die Verbindung aufbaut entscheidet noch vor Aussendung des ersten Pakets über die entsprechenden Werte, welche sich im Verlauf nicht ändern. Legt man diese Tatsache zu Grunde, kann man recht einfach die Pakete einer TCP–Verbindung isolieren. Dabei kann jedoch ein Problem auftauchen: Theoretisch kann ein Rechner die Kombination aus Portnummern und IP–Adressen für mehrere zeitliche getrennte TCP–Verbindungen verwenden. In der Regel werden für die eigene Portnummer bei ausgehenden Verbindungen immer neue Werte gewählt. Jedoch ist der Bereich, aus dem diese Werte gewählt werden, beschränkt. Beispielsweise unter Linux enthält dieser Bereich 28233 [1] verschiedene Werte. Spätestens, wenn dieser Nummernvorat erschöpft ist, müssen Portnummern wiederverwendet werden.

Anders als im vorhergehenden Absatz beschrieben, gibt es doch eine Situation, in der sich die Werte des Vier–Tupels ändern können: Befindet sich zwischen den zwei Kommunikationspartnern ein Router, der Network Address Translation (NAT) durchführt, so ändert dieser sozusagen „on-the-fly“ IP–Adresse und möglicherweise auch die Portnummer. Für die Rekonstruktion der TCP–Verbindungen ist dies jedoch unproblematisch, da die Werte konsequent verändert werden. Betrachtet man lediglich die NetFlow–Daten, kann man diese Manipulation ignorieren. Möchte man die rekonstruierten Verbindungen jedoch mit Konfigurationen von Firewalls korrelieren, die auf der anderen Seite des NAT–Routers stehen, so muss man eine Konvertierung vornehmen.

Was für TCP–Segmente gilt, soll nun auf Flows angewendet werden: Zur Rekonstruktion der TCP–Verbindungen werden Flows nach dem oben beschriebenen Schema gruppiert. Als Ergebnis dieser Gruppierung erhält man für jede Verbindung zwei Listen von Flows, für jede Richtung eine. Diese beiden Listen werden nun nach den Werten des first Attributs sortiert. Wie weiter oben beschrieben enthält der Wert first einen Zeitstempel, welcher den Zeitpunkt des Empfangs des jeweils ersten Pakets eines Flows beschreibt. Damit werden die Flows also in die zeitlich korrekte Reihenfolge gebracht. Nun vergleicht man die jeweils ersten Flows beider Richtungen. Anhand der erwähnten Zeitstempel kann man ablesen, welcher Flow früher angefangen hat. Daraus ergibt sich dann, welche Seite das erste Paket versendet hat beziehungsweise welche Seite den Verbindungsaufbau initiiert hat.

[1] Der Wert 28233 ergibt sich aus dem Intervall, aus dem Linux lokale Portnummern vergibt. Dieses Intervall wird über die Datei `/proc/sys/net/ipv4/ip_local_port_range` konfiguriert und für Rechner, die über mehr als 128 Megabytes Hauptspeicher verfügen, auf [32768; 61000] voreingestellt. Siehe auch `http://www-didc.lbl.gov/TCP-tuning/ip-sysctl-2.6.txt`

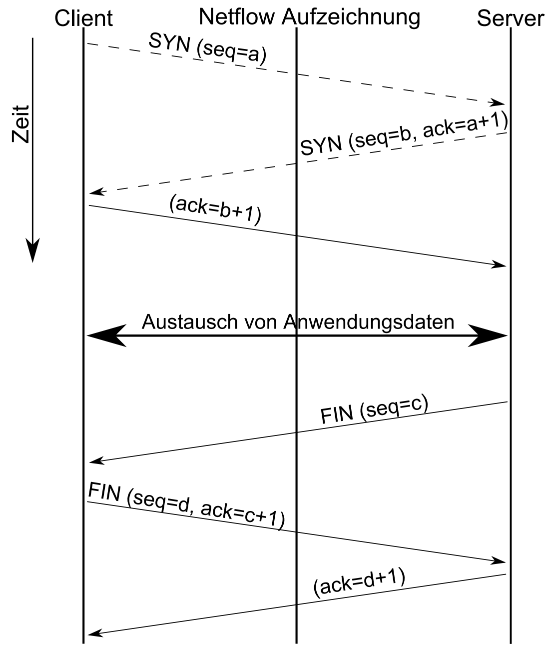

Abbildung 6.1: Der Nachrichtenaustausch in einer TCP–Verbindung.

6.4 Grenzen des Verfahrens

Nun gibt es eine Reihe von Bedenken, warum das beschriebene Verfahren nicht das gewünschte Ergebnis liefern könnte.

Wie berichtet, haben die verwendeten Zeitstempel eine recht grobe Auflösung von 64 Millisekunden. Angesichts der Tatsache, dass der zeitliche Abstand zwischen SYN– und SYN/ACK–Paketen in schnellen Netzen bei wenigen Millisekunden liegen kann, erscheint die Auflösung der NetFlow–Daten als zu grob. Es wird erwartet, dass viele Flows, die eigentlich nacheinander anfangen müssten, scheinbar zur selben Zeit beginnen. Lediglich bei Rechnern, die topologisch gesehen weit von einander entfernt sind, kann der Initiator richtig bestimmt werden.

Ein weiteres Problem stellen die in Abschnitt 5.3 beschriebenen verlorengegangenen Flows dar. Für den Fall, dass just der erste Flow, der Pakete vom Initiator in Richtung des antwortenden Rechners beschreibt, verloren geht, wird fälschlicherweise der antwortende Rechner als Initiator identifiziert, es sei denn, der erste Flow in Gegenrichtung geht ebenfalls verloren.

Als nächstes sollen die Auswirkungen der Tatsache diskutiert werden, dass in den vorliegenden NetFlow–Daten das jeweils erste Paket eines Flows nicht erfasst wird.

Im Regelfall hat der Verlust keine negativen Konsequenzen, da er konsequent für beide Richtungen erfolgt. Abbildung 6.1 zeigt ein Beispiel. Die beiden gestrichelt gezeichneten Pakete gehen verloren. Dazu gehören das SYN–Paket, welches der Client versendet und das SYN/ACK–Paket, das der Server als Antwort darauf verschickt. Somit ist das erste Paket, welches in den NetFlow–Daten aufgezeichnet wird, das ACK–Paket vom Client. Von daher wird der Client korrekt als Initiator bestimmt.

Abbildung 6.2 zeigt ein anderes Beispiel auf. Hier geht das dritte Paket, also das ACK–Paket, welches der Client an den Server schickt, verloren. Dies geschieht noch **bevor** es den Router erreicht, welcher die Flows aufzeichnet. Der Server wartet daraufhin vergeblich auf das ACK–Paket, welches eine Bestätigung seines zuvor ausgesendeten SYN/ACK–Pakets darstellen würde. Nach einer gewissen Zeit, entscheidet sich der Server das SYN/ACK–Paket erneut auszusenden. Dies führt zu der Situation, dass der Router, welcher sich zwischen den zwei Kommunikationspartner befindet, zuerst zwei Pakete vom Server sieht, bevor er das zweite vom

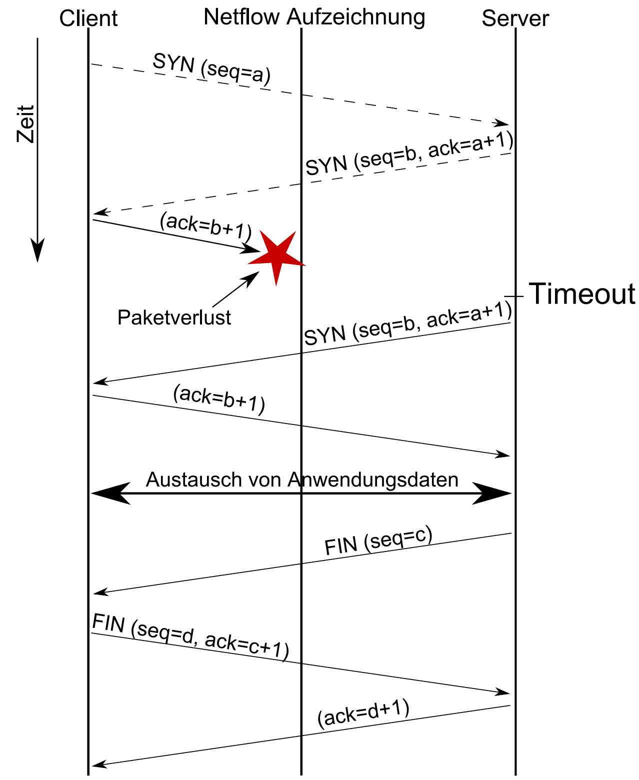

Abbildung 6.2: Beispiel einer TCP–Verbindung mit Paketverlust während des Drei–Wege–Handshakes.

Client sieht. Sieht man sich nun die induzierten NetFlow-Daten dieser Kommunikation an, so scheint es, als hätte zuerst der Server ein Paket ausgesendet. Es wird daher irrtümlich angenommen, dass der Server die Kommunikation initiiert hat.

An dieser Stelle sei ein kleiner Exkurs zu UDP erlaubt. Hier stellt sich ein ganz ähnliches Problem. Damit der richtige Rechner als Client bestimmt wird, muss dieser zwei Pakete ausgesendet haben, noch bevor der Server das zweite Paket verschickt hat. Problematisch ist zum Beispiel der Fall, wenn der Client eine Anfrage in Form eines einzigen UDP–Pakets an den Server aussendet. Fällt die Antwort darauf recht groß aus, weswegen das zurückgesendete IP–Paket fragmentiert werden muss, dann tritt genau der beschriebene Fall ein.

Auch bei recht einfachen Anfragen treten Schwierigkeiten auf. Besteht die Kommunikation aus lediglich einem Paket in jede Richtung, wie es zum Beispiel bei den meisten DNS–Abfragen der Fall ist, dann taucht die Kommunikationsbeziehung in den NetFlow–Daten überhaupt nicht auf.

6.5 Alternative Möglichkeiten

Um wieder auf TCP–Verbindungen zurück zukommen, sei noch eine andere Methode erwähnt, um die Richtung einer TCP–Verbindung zu bestimmen. Das Betriebssystem Linux wählt beispielsweise für ausgehende Verbindungen in der Regel eine lokale Portnummer im Bereich zwischen 32768 und 61000[2]. Häufig verbinden sich lokale Anwendungen mit einem Dienst auf Serverseite, welcher eine Portnummer verwendet, die aus der Menge der so genannten well–known Ports stammt. Nun könnte man sich die auf beiden Seiten verwendeten Portnummern ansehen und diese in die Kategorien „well– known“ und „not–well–known“ einteilen. Stammt

[2]Dieses Intervall wird über die Datei `/proc/sys/net/ipv4/ip_local_port_range` konfiguriert und für Rechner, die über mehr als 128 Megabytes Hauptspeicher verfügen, auf [32768; 61000] voreingestellt. Siehe auch `http://www-didc.lbl.gov/TCP-tuning/ip-sysctl-2.6.txt`

die Portnummer der einen Seite aus der Gruppe der „well– known“ Ports und die Portnummer der anderen Seite aus der Menge der „not–well–known“ Ports, dann nimmt man an, dass der Verbindungsaufbau von letzterer Seite ausging. Es können jedoch Probleme auftreten, wenn die Portnummern beider Seiten aus der gleichen Menge sind. Dann ist eine Unterscheidung nicht möglich. Ein Angreifer könnte zum Beispiel bewusst einen well–known Port als lokalen Port für seinen Angriff nutzen, um bewusst diese Situation zu provozieren. Nichts hält einen davon ab, Port 80 (HTTP) als lokalen Port für seine ausgehenden Verbindungen zu wählen. [3]

[3] Unter dem Betriebsystem UNIX muss man der Benutzer root sein, um Portnummern unter 1024 zu verwenden.

7 Implementierung und Evaluation des Verfahrens zur Rekonstruktion

Die im vorhergehenden Abschnitt beschriebene Methode zur Rekonstruktion der TCP–Verbindungen soll nun evaluiert werden. Dazu bedarf es einer Referenz, die definitiv über die Richtung einer Verbindung Auskunft gibt. Für jede ermittelte Verbindung soll überprüft werden können, ob diese richtig rekonstruiert wurde. Das wichtigste Kriterium dabei ist, ob die Rollen von Client und Server den Kommunikationspartnern richtig zugewiesen wurden.

7.1 Filterung der Datenbasis

Die zur Verfügung stehenden NetFlow–Daten werden zuerst auf solche Flows gefiltert, die Port 80 als Quell– oder Zielport besitzen. Für diese Flows ist es einfach festzustellen, welche Seite welche Rolle wahrnimmt, da Port 80 ein well–known Port darstellt. Dabei liegt natürlich die Annahme zu Grunde, dass in dem betrachteten Zeitintervall kein Rechner **aktiv** eine Verbindung unter Verwendung von Port 80 als lokale Portnummer aufgebaut hat. Diese Annahme erscheint sinnvoll, da kein Betriebssystem von sich aus diese Portnummer verwendet. Nur wenn eine Anwendung dem Betriebssystem explizit diese Portnummer mitteilt, wird diese für ausgehende Verbindungen verwendet. Eine Anwendung, die so vorgeht, verstößt entweder gegen die Spezifikation von HTTP oder missbraucht Port 80.

Im Rahmen dieser Arbeit wurde ein Skript (Listing A.1) in der Programmiersprache Ruby entwickelt, welches die im vorhergehenden Abschnitt beschriebene Methode implementiert und statistische Daten über die Zuverlässigkeit der Methode ausgibt.

Die Vorfilterung der Daten erfolgt mit Hilfe des Werkzeuges `flow-nfilter`. Dieses lässt sich mit Hilfe einer Textdatei konfigurieren. Dort werden so genannte Primitive definiert. Ein solches Primitiv ist eine Menge von IP–Protokoll–Nummern, Portnummern, Netzschnittstellen–IDs oder anderen Parametern. Diese Primitive werden bei der Definition von Filtern verwendet. Ein Filter besteht aus einer Reihe von Bedingungen, welche erfüllt sein müssen, damit ein Paket den Filter passieren kann. Eine Bedingung besteht aus einem Parameter, der verglichen werden soll und einem Primitiv, das die erlaubten Werte definiert. Ein solcher Parameter kann zum Beispiel das IP–Protokoll, Quell– oder Ziel–IP–Adresse oder Quell– oder Zielport sein.

Das folgende Beispiel definiert ein Primitiv, das die zwei Portnummern 80 und 443 umfasst. Durch das Schlüsselwort `type` wird der Typ defniert. Mit `permit` werden die einzelnen Portnummern festgelegt.

```
filter-primitive http
  type ip-port
  permit 80
  permit 443
```

Möchte man ein Primitiv definieren, das alle IP–Adressen im Subnetz `129.187.15.0/24` beschreibt, so verwendet man die folgende Definition.

```
filter-primitive mynet
  type ip-address-mask
  permit 129.187.15.0/24
```

Diese zwei Primitive kann man nun zu einem Filter zusammenfügen. Der nun folgende Beispielfilter könnte in Worten wie folgt beschrieben werden: „Wähle nur solche Flows aus, dessen Zielport 80 oder 443 ist und dessen Zieladresse im Netz `129.187.15.0/24` liegt.“

```
filter-definition mynet
  match ip-destination-port http
  match ip-destination-address mynet
```

Aufeinanderfolgende Bedingungen werden durch ein logisches UND miteinander verknüpft. Somit müssen beide Bedingungen erfüllt sein, damit das betrachtete Paket den Filter passieren darf. Der Zielport muss aus der Menge $\{80, 443\}$ stammen und die Zieladresse muss im Netz `129.187.15.0/24` liegen.

Die Daten aus dem MWN werden zusätzlich auf Flows gefiltert, an denen Rechner in einem festgelegten VLAN beteiligt sind. Dies dient lediglich der Datenreduzierung. Ohne diese Filterung wäre die Datenmenge schlicht zu groß, und das Ruby–Skript bräuchte zu lange, um diese Datenmenge zu bearbeiten.

7.2 Verarbeitung durch das Ruby–Skript

Die Ausgabe von `flow-nfilter` geschieht in einem binären Format. Damit diese Daten einfacher eingelesen werden können, werden sie zuerst mit Hilfe des Programms `flow-export` in ein auf ASCII basierendes CSV–Format umgewandelt. In diesem Format bildet jede Zeile einen Flow. Die einzelnen Attribute eines Flows sind durch Kommata von einander getrennt.

Die im folgenden beschriebene Auswertung kann in fünf Schritte gegliedert werden:

1. Parsen der Eingabe und Erzeugung von Instanzen der Klasse `Flow` für jeden einzelnen Flow. Im selben Moment werden die Flows zu Verbindungen gruppiert.
2. Aussortierung von Verbindungen, die zu Aufzeichnungsbeginn bereits existierten.
3. Alle Flows einer Verbindung werden chronologisch sortiert.
4. Ermittlung des ersten Flows in jede Richtung.
5. Berechnung der zeitlichen Differenz zwischen erstem Flow in Rückrichtung und erstem Flow in Hinrichtung.

In dem Ruby–Skript (Listing A.1) wird jeder Flow durch eine Instanz der Klasse `Flow` dargestellt. Jede Zeile aus der Ausgabe von `flow-export` wird einzeln dem Konstruktor dieser Klasse übergeben. Der Konstruktor verarbeitet diese Zeichenkette. Die einzelnen Attribute des Flows stehen danach als Eigenschaften des neuen Objekts zur Verfügung.

Jeder Instanz der Klasse `Flow` wird ein so genannter `FlowKey` zugeordnet. Ein wichtiges Merkmal dieses `FlowKeys` ist, dass er für entgegengesetzte Flows identisch ist. Dies soll an einem kurzen Beispiel erläutert werden. Der Flow von IP–Adresse `1.2.3.4` Port 47 nach IP–Adresse `5.6.7.8` Port 11 besitzt den gleichen `FlowKey` wie der Flow von IP–Adresse `5.6.7.8` Port 11 nach `1.2.3.4` Port 47, also der Flow in die Gegenrichtung. Erreicht wird dies dadurch, dass die beiden Kombinationen aus IP–Adresse und Portnummer für den `FlowKey` nach der IP–Adresse sortiert werden. Damit besitzen die genannten Flows ein und denselben Schlüssel.

Direkt beim Einlesen der einzelnen Flows, werden diese in eine Hashtabelle eingefügt. Als Schlüssel wird dabei der beschriebene `FlowKey` verwendet. Damit unter einem Schlüssel mehrere Flows gespeichert werden können, sind die Werte der Hashtabelle Arrays von Flows. Sobald der Einleseprozess abgeschlossen ist, hat man eine Hashtabelle zur Verfügung, welche `FlowKeys` auf Arrays von Flows abbildet. Diese Arrays bilden die gesuchte Gruppierung von Flows. Es wird angenommen, dass alle Flows in einem Array zu einer einzigen TCP–Verbindung gehören.

Im nächsten Schritt werden Verbindungen aussortiert, von denen angenommen wird, dass sie vor Beginn der NetFlow–Aufzeichnung begonnen haben. Es wird angenommen, dass jede TCP–Verbindung mindestens alle 32 Minuten einen Flow induziert. Wie auf Seite **??** beschrieben, exportiert ein Router einen Flow spätestens nachdem dieser 32 Minuten aktiv war. Unter der Voraussetzung, dass über jede TCP–Verbindung kontinuierlich Daten versendet werden, kann man also annehmen, dass eine TCP–Verbindung neu aufgebaut wurde, sofern seit 32 Minuten kein Flow mit gleichem Schlüssel aufgetaucht ist. Flows die innerhalb der ersten 32 Minuten der Aufzeichnung angelegt wurden, werden ignoriert. Auch werden alle nachfolgenden Flows, die zur gleichen Verbindung gehören ignoriert. Diese TCP–Verbindungen wurden möglicherweise vor Beginn der Aufzeichnung aufgebaut. Der richtige Initiator kann deshalb nicht mehr festgestellt werden.

Betrachtet man TCP–Verbindungen, über die das IMAP–Protokoll verwendet wird, so ist die oben gemachte Annahme falsch. E–Mail–Clients, welche IMAP verwenden, halten die TCP–Verbindung in der Regel so lange offen, so lange die Anwendung läuft. Daten werden nur dann ausgetauscht, wenn eine neue E–Mail eintrifft, oder der Benutzer mit der Anwendung interagiert. Was die korrekte Bestimmung des Initiators betrifft, steht man hier leider auf verlorenem Posten.

Im weiteren Verlauf der Analyse werden alle Flows einer Verbindung unabhängig von der Richtung chronologisch sortiert. Als Schlüssel dient dabei der Zeitstempel, welcher in dem Attribut `first` eines jeden Flows zu finden ist. Danach wird pro Richtung der zeitlich erste Flow ermittelt. Dies setzt natürlich voraus, dass überhaupt Flows in beide Richtungen existieren. Es wurden Verbindungen beobachtet, bei denen dies nicht der Fall ist, die also lediglich aus einem Flow bestehen. Diese Fälle werden gesondert behandelt. Dabei wird festgehalten, ob dieser „einsame“ Flow Pakete beschreibt, welche vom Client zum Server oder in Gegenrichtung geflossen sind.

Die folgende Verbindung zwischen den Endpunkten `129.187.12.5:59536` und `195.189.236.165:80` dient als Beispiel für solch einen „einsamen“ Flow.

```
129.187.12.5:59536 - 195.189.236.165:80
  first appearance 3585 seconds after start of recording
  Bad single flow
  <= prot: 6 first 158489708 last 158604204 dpkts 48 doctets 63999
```

Hier sieht man, wie ein Rechner von Port 80 aus 63999 Bytes an einen anderen Rechner sendet. Dies erscheint seltsam, da der Empfänger zuvor kein einziges Paket in entgegengesetzter Richtung verschickt hat, womit er die Daten hätte anfordern können. Hier ging mit hoher Sicherheit ein Flow verloren, da kein HTTP–Server von sich aus, unaufgefordert Daten verschickt.

Für die Verbindungen, die entgegengesetzte Flows besitzen, wird die Differenz $\Delta = t_r - t_f$ gebildet. t_f und t_r beschreiben die Zeitpunkte, zu denen das erste Paket in Hinrichtung beziehungsweise in Rückrichtung aufgetreten ist.[1] Diese Information lässt sich den first–Zeitstempeln der jeweiligen ersten Flows pro Richtung entnehmen. Mit Hinrichtung ist in diesem Zusammenhang die Richtung vom Client zum Server gemeint, mit Rückrichtung die entgegengesetzte Richtung. Da die Verbindungen zuvor nach einem festen, well–known Port gefiltert wurden, kann man anhand dieses well–known Port die Richtung eines Flows bestimmen. Es wird davon ausgegangen, dass der initiierende Host zuerst ein Paket aussendet, weshalb die Differenz Δ immer einen positiven Wert ergeben sollte. Da wie in Abschnitt 5.2 beschrieben, die Auflösung der Zeitstempel lediglich 64 Millisekunden beträgt, wird die Differenz Δ auf ein Vielfaches von 64 Millisekunden gerundet.

Sofern das Ruby–Skript entsprechend konfiguriert wurde, gibt es für jede Verbindung eine Zusammenfassung aus. Als Beispiel soll hier eine Verbindung dienen, die aus insgesamt sechs Flows besteht:

```
129.187.12.43:1956 - 212.201.100.133:80
  first appearance 49822 seconds after start of recording
  Difference: 64
  Rounded Difference: 64
  => prot: 6 first 168855243 last 168940811 dpkts 287 doctets 33892
  <= prot: 6 first 168855307 last 168855371 dpkts 19 doctets 25431
  <= prot: 6 first 168862350 last 168940622 dpkts 496 doctets 686966
  <= prot: 6 first 169022852 last 171104644 dpkts 4499 doctets 6153433
  => prot: 6 first 169022852 last 171104580 dpkts 2362 doctets 213824
```

Die erste Zeile enthält die Adressen beider Kommunikationspartner. Welcher der Kommunikationspartner die Rolle des Clients einnimmt, kann an den Portnummern abgelesen werden. Die sechs Flows, welche in diesem Beispiel aufgelistet werden, sind chronologisch sortiert. Der Pfeil => bzw. <= gibt an, ob der Flow vom erstgenannten zum zweitgenannten Rechner fließt oder umgekehrt. Wie man hier erkennen kann, erfolgt der chronologisch erste Flow in Hinrichtung, also vom Client zum Server. Die Differenz Δ zum ersten Flow in

[1] Bei den, für dieses Projekt vorliegenden Daten, handelt es sich allerdings um das zweite Paket eines jeden Flows. (siehe Abschnitt 5.5)

Rückrichtung beträgt genau 64 Millisekunden. In diesem Beispiel würde der Initiator also richtig bestimmt werden.

Das Ruby–Skript speichert die Häufigkeiten aller auftretenden Differenzen Δ und gibt am Ende der Analyse eine Verteilungsfunktion aus, welche mit Hilfe des Werkzeuges gnuplot in eine Grafik umgewandelt wurde. Wie bereits beschrieben, werden alle Verbindungen, die Flows in lediglich eine Richtung besitzen, besonders registriert.

8 Ergebnisse der Evaluation

Als wichtigstes Beispiel dieser Arbeit, dient eine Aufzeichnung vom Donnerstag, den 26. Juli 2007. Von allen aufgezeichneten Flows werden nur solche betrachtet, die Teil einer HTTP–Verbindung sind, und an denen ein Rechner im VLAN 23 beteiligt ist. Das VLAN 23 lässt sich an Hand der Interface ID 115 erkennen.

Die folgende Tabelle listet die Ergebnisse der Analyse auf:

Verbindungen insgesamt	76418
Differenz Δ in Millisekunden	Häufigkeit
$<$ -64ms	253 (0,3%)
-64ms	1242 (1,6%)
0ms	38503 (50,3%)
64ms	19352 (25,3%)
128ms	6111 (8,0%)
$>$ 128ms	3306 (4,3%)
nur Hinrichtung	6145(8,0%)
nur Rückrichtung	1506(2,0%)

Verwendet man die beschriebene Methode zur Ermittlung des Initiators, läge man lediglich in 34914 von 76418 Fällen richtig ($\Delta > 0$ plus die „nur Hinrichtung“–Fälle). Das macht einen Anteil von gerade einmal 45,7% aus. In 50,3% der Fälle könnte man gar keine Aussage treffen ($\Delta = 0$) und in 3,9% der Fälle ($\Delta < 0$ plus die „nur Rückrichtung“–Fälle) träfe man eine fehlerhafte Aussage.

In Abbildung 8.1 ist die Verteilungsfunktion der Differenzen Δ dargestellt. An den Stellen, an denen die Verteilungsfunktion einen Sprung macht, befindet sich eine Häufung der Verbindungen. Je größer der Sprung ist, desto mehr Verbindungen weisen die Differenz auf, welche man auf der x–Achse ablesen kann.

Um die beschriebenen Ergebnisse zu unterstützen, wurde das Experiment unter Verwendung von anderen Parametern wiederholt. Abbildung 8.2 zeigt die Verteilungsfunktion des Experiments unter Verwendung der Portnummer 22 (SSH). Datum und VLAN wurden nicht verändert.

An dieser Verteilungsfunktion ist auffällig, dass ein recht hoher Anteil an Verbindungen existiert, bei denen der Verbindungsaufbau recht schnell erfolgte. Dies bedeutet, dass das erste Paket in Rückrichtung dem ersten Paket in Hinrichtung zeitlich recht schnell folgte, was vermuten lässt, dass die beteiligten Rechner topologisch nicht weit voneinander entfernt sind. Es wird vermutet, dass SSH beim LRZ häufig benutzt wird, um Rechner im Münchner Wissenschaftsnetz zu administrieren. Seltener kommt es vor, dass sich Mitarbeiter mit weiter entfernten Rechner über SSH verbinden.

Auch für Port 143 (imap) wurde das Experiment wiederholt. Datum und VLAN blieben wieder gleich. Abbildung 8.3 zeigt das Ergebnis. Auffällig ist hier der große Anteil an Verbindungen mit einer negativen Differenz Δ. Hier kommt vermutlich das auf Seite 23 beschriebene Problem ins Spiel. Eine erhebliche Anzahl an IMAP–Verbindungen wurden wohl vor Beginn der Aufzeichnung aufgebaut. Da über IMAP–Verbindungen nur sporadisch Daten gesendet werden, werden diese Verbindungen auch nicht als bereits bestehende Verbindungen aussortiert (siehe Seite 22). Sendet der Server nun plötzlich eine Nachricht an den Client, weil zum Beispiel eine neue E–Mail eingegangen ist, dann wird der Server irrtümlicherweise als Initiator einer bereits bestehenden Verbindung bestimmt.

Für den Durchlauf, dessen Ergebnis man in Abbildung 8.4 sehen kann, wurde Sonntag, der 1. Dezember 2007 gewählt, um mögliche Abhängigkeiten von Wochentagen auszuschließen. Qualitativ deckt sich das Ergebnis aber mit den bisherigen Ergebnissen.

Um auch Abhängigkeiten vom betrachteten VLAN auszuschließen, wurde der Versuch noch ein letztes Mal für ein anderes Interface wiederholt. Als Datum wurde wieder der 26.07.2007 und als Portnummer 80 gewählt.

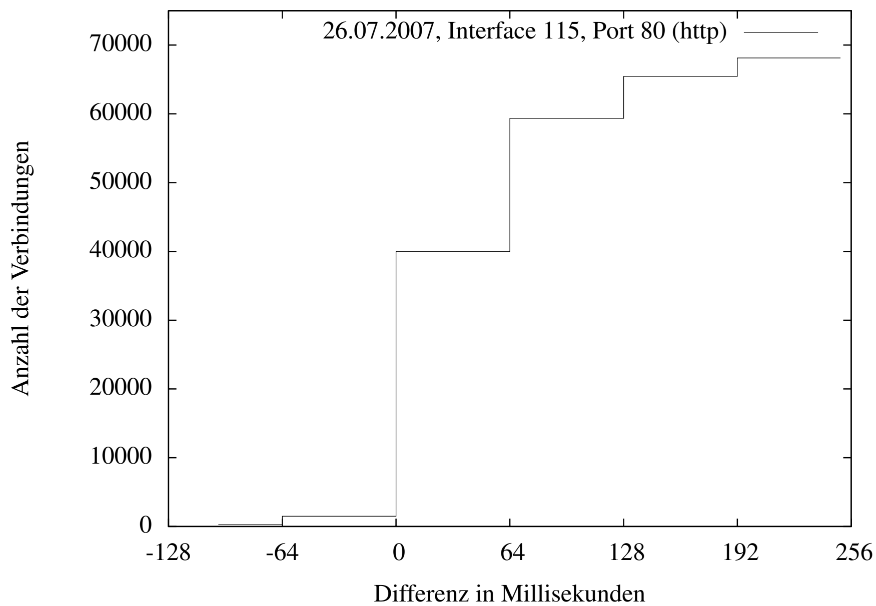

Abbildung 8.1: Verteilungsfunktion der Differenzen Δ. 26.07.2007, Interface 115, Port 80 (http)

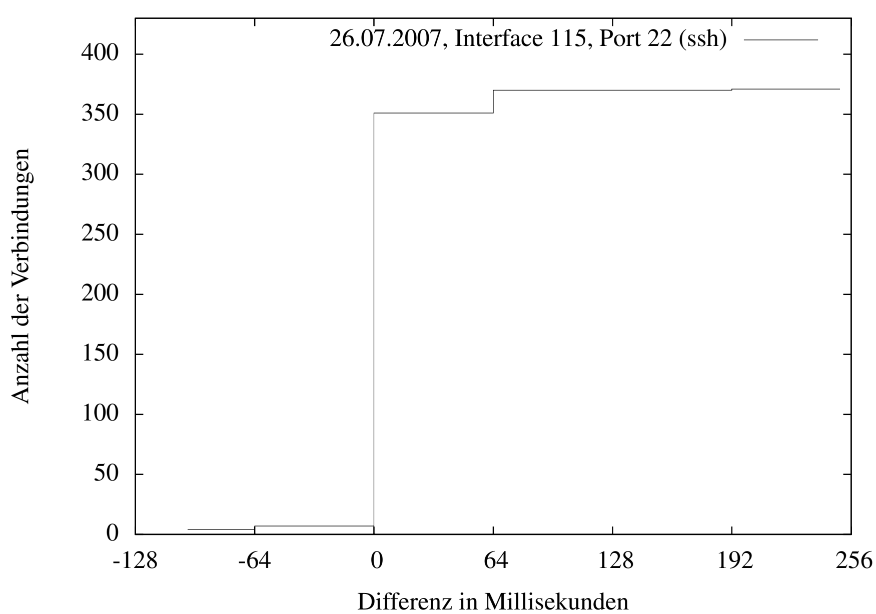

Abbildung 8.2: Verteilungsfunktion der Differenzen Δ. 26.07.2007, Interface 115, Port 22 (ssh)

Das Ergebnis ist in Abbildung 8.5 zu sehen und unterscheidet sich qualitativ nicht von den restlichen Ergebnissen.

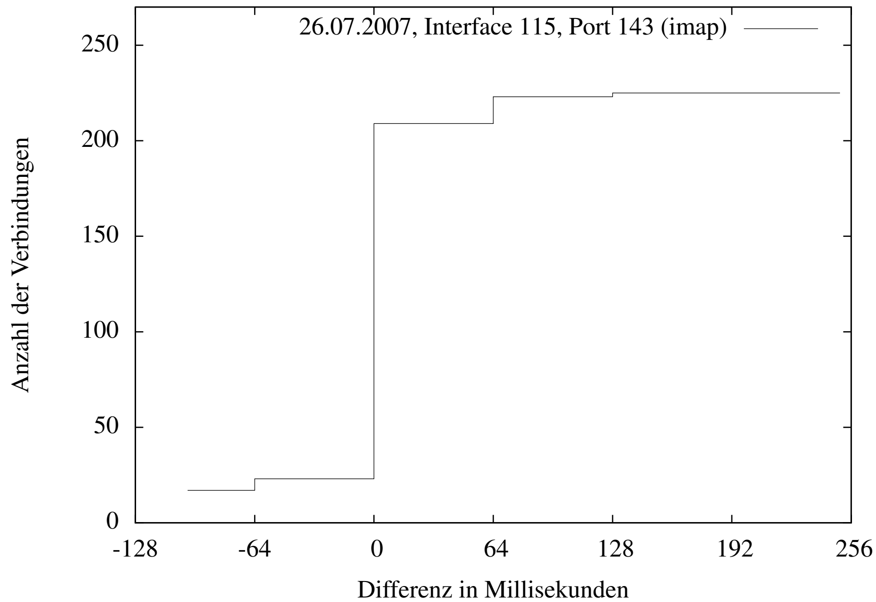

Abbildung 8.3: Verteilungsfunktion der Differenzen Δ. 26.07.2007, Interface 115, Port 143 (imap)

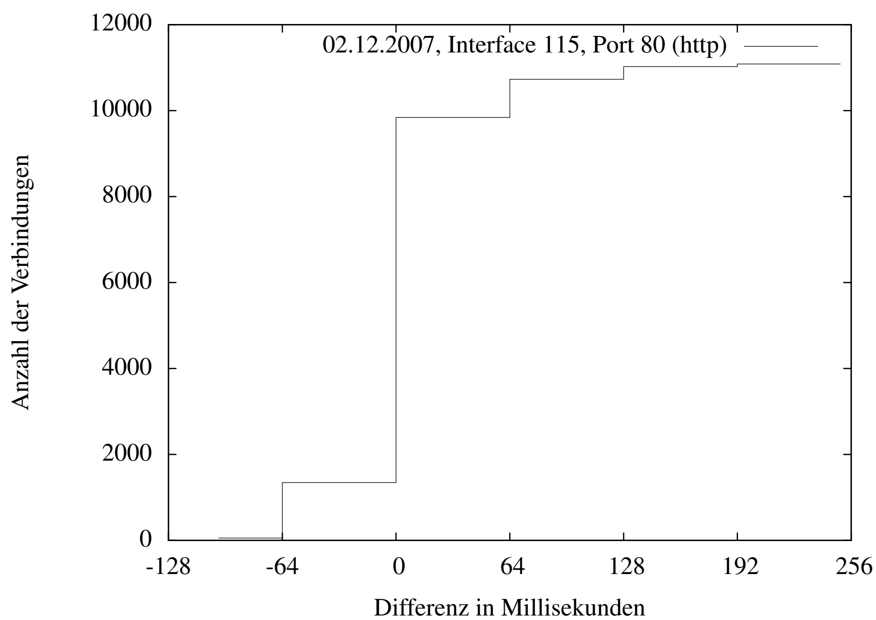

Abbildung 8.4: Verteilungsfunktion der Differenzen Δ. 02.12.2007, Interface 115, Port 80 (http)

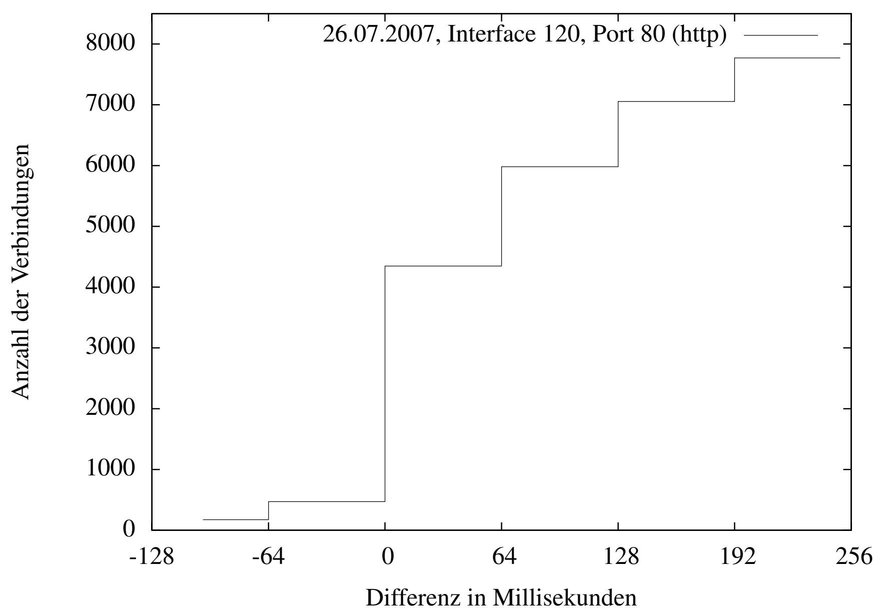

Abbildung 8.5: Verteilungsfunktion der Differenzen Δ. 26.07.2007, Interface 120, Port 80 (http)

9 Bewertung der Ergebnisse

Leider ist es in diesem Projekt nicht gelungen, aus den NetFlow–Daten die notwendigen Informationen zu extrahieren, um die Daten mit der Konfiguration einer Firewall zu korrelieren. Für eine zuverlässig Analyse bedarf es einer Datenbasis, in der alle TCP–Verbindungen enthalten sind. Da es sich um eine Sicherheitsanwendung handelt, kann es nicht toleriert werden, dass in etwa die Hälfte der TCP–Verbindungen unsichtbar bleibt. Angriffe in dieser nicht sichtbaren Menge würden unerkannt bleiben. Zudem liefert das Verfahren zur Rekonstruktion von TCP–Verbindungen eine erhebliche Zahl an falschen Ergebnissen. Dies würde dazu führen, dass die Analysesoftware eine große Zahl an Fehlalarmen produzieren würde, was das Ergebnis wiederum wertlos machen würde.

Anja Feldmann und Robin Sommer haben sich ebenfalls der Rekonstruktion von TCP–Verbindungen aus NetFlow–Daten gewidmet[SoFe 02]. Allerdings haben sie mit anderen Voraussetzungen gearbeitet. In den NetFlow–Daten, welche Feldmann und Sommer verwendet haben, waren TCP–Flags vorhanden. Außerdem hatten sie weder mit verlorengegangenen Flows noch mit der Tatsache, dass das erste Paket eines jeden Flows fehlt, zu kämpfen. (siehe Abschnitt 5.3 und 5.5) Zur Bestimmung von Client und Server einer Verbindung, haben sie sich einer erweiterten Menge an well–known Ports bedient. Wie auf Seite 19 begründet wurde, ist diese Methode für eine Sicherheitsanwendung nicht brauchbar. Ein Angreifer könnte mit entsprechendem Wissen, die Analysesoftware austricksen. Feldmann und Sommer berichten, dass sie in 99,5% der Fälle den Initiator richtig bestimmt haben. Im Umkehrschluss bedeutet dies, dass für 0,5% der Verbindungen der Initiator falsch bestimmt wurde. Der Benutzer der Analysesoftware müsste in diesen Fällen per Hand nachbessern, was zusätzlichen Arbeitsaufwand für ihn bedeuten würde.

10 Zusammenfassung und Ausblick

Wie im vorhergehenden Abschnitt dargelegt, reichen die NetFlow–Daten nicht aus, um nicht legitime Verbindungen aufzudecken. Schuld daran sind hauptsächlich die große Granularität der Zeitstempel sowie unvollständige Daten. Eine alternative Möglichkeit zur Aufzeichnung des Netzwerkverkehrs ist der Einsatz eines Sniffers, welcher alle IP–Pakete vollständig aufzeichnet. Sofern bei der Aufzeichnung keine Pakete verloren gehen, wäre eine vollständige Rekonstruktion des Datenverkehrs im Netz möglich. Diese Rekonstruktion könnte man dann mit der Firewall–Konfiguration abgleichen.

A Quellcode

Listing A.1: Ruby–Skript zur Auswertung der Netflow–Daten

```
#!/usr/bin/ruby

# Ueber die Standardeingabe wird die Ausgabe von
# flow-export -f 2
# erwartet.

# Aufruf:
# ./parse.rb <Portnummer> <Ausgabedatei fuer CDF>

# Beispiel:
# $ flow-cat -g <netflow files> | \
#     flow-nfilter -f filter.cfg -F ifx_portx -v port=<portnummer> -v if=<interface> | \
#     flow-export -f 2 | ./parse.rb <portnummer> <Ausgabedatei fuer CDF>

require 'ipaddr'
require 'resolv'

# Wenn die Konstante OUTPUT das Symbol :each_connection
# enthaelt, werden Informationen ueber jede einzelne
# Verbindung ausgegeben.

OUTPUT = [:each_connection]
# OUTPUT = []

port = Integer(ARGV[0])
datafile = ARGV[1]

# Die Klasse Flow repraesentiert einen einzelnen Flow.
# Die Attribute des Flows werden auf Eigenschaften
# der jeweiligen Objekte abgebildet.

# So laesst sich zum Beispiel ueber f.dpkts herausfinden,
# wieviele Bytes der Flow f umfasst.

class Flow
    # semantic of this array
    # [:symbol_name, type, Do we need this attribute?]
    FIELDS = [
        [:unix_secs,  :numeric, false],
        [:unix_nsecs, :numeric, false],
        [:sysuptime,  :numeric, true ],
        [:exaddr,     :ipaddr,  false],
        [:dpkts,      :numeric, true],
        [:doctets,    :numeric, true],
        [:first,      :numeric, true],
        [:last,       :numeric, true],
        [:engine_type,:numeric, false],
        [:engine_id,  :numeric, false],
        [:srcaddr,    :ipaddr,  true],
        [:dstaddr,    :ipaddr,  true],
```

```
        [:nexthop,    :ipaddr,  false],
        [:input,      :numeric, true],
        [:output,     :numeric, true],
        [:srcport,    :numeric, true],
        [:dstport,    :numeric, true],
        [:prot,       :numeric, true],
        [:tos,        :numeric, false],
        [:tcp_flags,  :numeric, false],
        [:src_mask,   :numeric, false],
        [:dst_mask,   :numeric, false],
        [:src_as,     :numeric, false],
        [:dst_as,     :numeric, false]
        ]
    FIELDS2 = FIELDS.map { |f| ["@#{f[0]}".to_sym,f[1],f[2]] }

    # "Getter" fuer Attribute definieren.
    attr_reader *(FIELDS.select { |f| f[2] }.map { |f| f[0] } )

    def initialize(line)
        # Durch Kommata abgetrennte Zeichenketten werden in ein Array
        # umgewandelt. Beispiel:
        # aus "a,b,c" wird ["a","b","c"]
        attributes = line.split(',')
        # Das Array kuerzen, sollte es zu gross sein
        attributes.pop while attributes.length > FIELDS2.length
        # Einzelne Werte werden den Eigenschaften zugewiesen
        [FIELDS2,attributes].transpose.each do |field,value|
            name,type,interesting = field
            next unless interesting
            value = if type == :ipaddr then
                    # Wenn der Wert eine IP-Adresse ist, dann
                    # entsprechendes Objekt erzeugen.
                    IPAddr.new(value)
                else
                    # Ist der Wert ein Integer, dann entsprechend
                    # konvertieren.
                    Integer(value)
                end
            # Wert wird der Objektvariable zugewiesen.
            instance_variable_set(name,value)
        end
    end
    def getFlowKey()
        FlowKey.new(@srcaddr,@srcport,@dstaddr,@dstport)
    end
    def formatAddress(addr)
        r = addr.to_s
        begin
            r << " (#{Resolv.getname(r)})"
        rescue Resolv::ResolvError
        end
        return r
    end
    def to_s()
        "#{formatAddress(srcaddr)} port #{srcport} => " +
        "#{formatAddress(dstaddr)} port #{dstport}"
    end
end

# Objekte der Klasse FlowKey dienen als Hash-Werte fuer
# die Flows. Die FlowKeys von zwei entgegengerichteten Flows
```

```ruby
# sind identisch.

class FlowKey
    attr_reader :addr1,:port1,:addr2,:port2
    def initialize(addr1,port1,addr2,port2)
        # Sortiere IP-Adressen und Portnummern nach IP-Adressen
        if addr1.to_i() > addr2.to_i()
            addr2,addr1 = addr1,addr2
            port2,port1 = port1,port2
        end
        @addr1,@port1 = addr1,port1
        @addr2,@port2 = addr2,port2
    end
    def hash
        # Bilde Hashwert fuer diese Instanz
        @addr1.to_i.hash + @port1.hash +
        @addr2.to_i.hash + @port2.hash
    end
    def ==(b)
        @addr1 == b.addr1 and
        @port1 == b.port1 and
        @addr2 == b.addr2 and
        @port2 == b.port2
    end
    def eql?(b); self == b; end
    def to_s
        "#{addr1}:#{port1} - #{addr2}:#{port2}"
    end
end

# Eine Verbindung besteht aus mehreren Flows
# Ausserdem wird festgehalten, wann das erste
# Paket dieses Flows aufgetaucht ist.
class Connection
    def initialize
        @flows = Array.new
    end
    attr_accessor :flows
    attr_accessor :first_appearance
end

connections = Hash.new

start_time = nil
end_time = nil

# Verbindungen, die in den ersten 32 Minuten
# auftauchen, werden ignoriert.

SKIP_TIME = 32 * 60 * 1000 # seconds
# SKIP_TIME = 2 * 60 * 1000 # seconds

# Lese Flows ein
$stdin.each_with_index do |l,i|
    # Ignoriere Zeilen, die mit # anfangen.
    next if l[0] == ?#
    # Loesche LF am Ende der Zeile
    l.chomp!
    # Erstelle neue Instanz, die diesen Flow repraesentiert.
    f = Flow.new(l)
    start_time = f.sysuptime if start_time == nil
```

```
    end_time = f.sysuptime
    # Gibt es bereits eine Verbindung, zu der dieser Flow passt?
    if !connections.has_key?(f.getFlowKey())
        # Falls nicht: Erstelle eine neue Verbindung.
        connections[f.getFlowKey()] = Connection.new
        connections[f.getFlowKey()].first_appearance = f.first
    end
    # Flow zur Verbindung hinzufuegen.
    connections[f.getFlowKey()].flows << f
end

# Loesche Verbindungen, die bereits in den ersten 32 Minuten existiert haben.
before = connections.length
connections.delete_if { |key,value| value.first_appearance - start_time < SKIP_TIME }
after = connections.length
puts "Deleted␣#{before␣-␣after}␣early␣connections"
before = after
connections.delete_if { |key,value| end_time - value.first_appearance < SKIP_TIME }
after = connections.length
puts "Deleted␣#{before␣-␣after}␣late␣connections"

# Erstelle Funktion der Haeufigkeiten der Anzahl von Flows pro Verbindung.
number_of_flows = Hash.new(0)
connections.each do |key,conn|
    number_of_flows[conn.flows.length] = number_of_flows[conn.flows.length] + 1
end

number_of_flows.sort.each do |length,frequency|
    puts "Number␣of␣flows:␣#{length}␣frequency:␣#{frequency}"
end

# Erstelle Funktion der Haeufigkeiten der Differenzen
differences = Hash.new(0)
above_interval = 0
below_interval = 0

TRESHOLD = 100
INTERVAL = (-100..250)

# Anzahl Verbindungen, die nur Flows in Hinrichtung umfassen
good_single_flows = 0
# Anzahl Verbindungen, die nur Flows in Rueckrichtung umfassen
bad_single_flows = 0

# Iteriere ueber alle Verbindungen.
connections.each do |key,conn|

    if OUTPUT.include?(:each_connection)
        puts key
        puts "␣␣first␣appearance␣#{(conn.first_appearance␣-␣start_time)/1000}␣" +
            "seconds␣after␣start␣of␣recording"
    end

    # Sortiere Flows nach first-Zeitstempel
    conn.flows.sort! { |a,b| a.first <=> b.first }

    # Suche den jeweils ersten Flow in Vorwaerts- und Rueckwaertsrichtung
    first_forward = nil
    first_backward = nil
    conn.flows.each { |f|
        first_forward = f if !first_forward and f.dstport == port
```

```
        first_backward = f if !first_backward and f.srcport == port
        break if first_forward and first_backward
    }
    if first_forward and first_backward
        # Berechne die Differenz
        difference = first_backward.first - first_forward.first
        # Differenz auf ein Vielfaches von 64 runden.
        rounded_difference = (difference.to_f / 64).round.to_i * 64
        puts "␣␣Difference:␣#{difference}" if OUTPUT.include?(:each_connection)
        puts "␣␣Rounded␣Difference:␣#{rounded_difference}" if OUTPUT.include?(:each_connection)
        # Unterschreitet die Differenz den Schwellenwert
        if difference < -TRESHOLD
            puts "␣␣!!␣Problematic␣flow." if OUTPUT.include?(:each_connection)
        end
        # Funktion der Hauefigkeiten aufbauen
        if INTERVAL.include?(rounded_difference)
            differences[rounded_difference] = differences[rounded_difference] + 1
        elsif difference < INTERVAL.begin
            below_interval = below_interval + 1
        else
            above_interval = above_interval + 1
        end
    elsif first_forward
        # Es existieren nur Flows in Hinrichtung
        puts "␣␣Good␣single␣flow" if OUTPUT.include?(:each_connection)
        good_single_flows = good_single_flows + 1
    elsif first_backward
        # Es existieren nur Flows in Rueckrichtung
        puts "␣␣Bad␣single␣flow" if OUTPUT.include?(:each_connection)
        bad_single_flows = bad_single_flows + 1
    end
    if OUTPUT.include?(:each_connection)
        # Alle Flows auflisten, die zu dieser Verbindung gehoeren.
        conn.flows.each { |f|
            direction = if key.addr1 == f.srcaddr then '=>' else '<=' end
            puts "␣␣#{direction}␣prot:␣#{f.prot}␣first␣#{f.first}␣last␣#{f.last}␣" +
                "dpkts␣#{f.dpkts}␣doctets␣#{f.doctets}␣input␣#{f.input}␣output␣#{f.output}"
        }
    end
end

# Schreibe Verteilungsfunktion in Datendatei
# fuer gnuplot
if datafile
    sum = below_interval
    File.open(datafile,"w+") { |file|
        INTERVAL.each do |i|
            sum = sum + differences[i]
            file.puts "#{i}␣#{sum}"
        end
    }
end

# Gebe statistische Daten aus

puts "<-64ms␣#{below_interval}"
-1.upto(2) { |i|
    puts "#{i*64}␣#{differences[i*64]}"
}
sum = 0
```

```
129.upto(INTERVAL.end) { |i|
    sum = sum + differences[i]
}

puts ">128ms␣#{sum+above_interval}"

puts "below_interval␣#{below_interval}"
puts "above_interval␣#{above_interval}"
puts "good_single_flows␣#{good_single_flows}"
puts "bad_single_flows␣#{bad_single_flows}"
puts "total␣number␣of␣connections␣#{connections.length}"
```

Listing A.2: Konfigurationsdatei für flow-nfilter

```
# Auruf durch

# flow-nfilter -f filter.cfg -F ifx_portx -v port=$p -v if=$if
# $p und $if bitte durch Portnummer bzw. SNMP-Interface-ID ersetzen.

# Nur TCP-Verbindungen
filter-primitive tcp
  type ip-protocol
  permit 6

# Primitiv zur Filterung auf ein bestimmtes Interface
filter-primitive ifx
  type ifindex
  permit @if

# Primitiv zur Filterung auf einen bestimmten Port
filter-primitive portx
  type ip-port
  permit @port

# Diese Filterdefinition kombiniert
# die Primitiven.
filter-definition ifx_portx
  match ip-destination-port portx
  match output-interface ifx
  match ip-protocol tcp
  or
  match ip-destination-port portx
  match input-interface ifx
  match ip-protocol tcp
  or
  match ip-source-port portx
  match output-interface ifx
  match ip-protocol tcp
  or
  match ip-source-port portx
  match input-interface ifx
  match ip-protocol tcp
```

Listing A.3: Datenbankschema zur Repräsentierung der ACLs

```
-- phpMyAdmin SQL Dump
-- version 2.8.0.3-Debian-1
-- http://www.phpmyadmin.net
--
-- Host: localhost
-- Generation Time: Jun 18, 2006 at 10:51 PM
```

```
-- Server version: 5.0.22
-- PHP Version: 5.1.2-1+b1
--
-- Database: `fw_config`
--

-- --------------------------------------------------------
-- object groups can be of one out of four different types:
-- * protocol (f.e. tcp,udp,icmp,esp)
-- * network (individual hosts and prefixes. e.g. 128.2.5.0/24)
-- * service (= tcp or udp port number, for example: www,ftp,ssh)
-- * icmp_type (f.e. echo, echo reply)

-- There is a row in object_groups for every object group, no matter of what
-- type the object group is. Depending on the type of the object group there
-- are rows for every object in the object group in one of the tables
-- icmp_objects, network_objects, protocol_objects or service_objects.

-- Access Control Lists

-- There exists a row in acls for every Access Control List. The individual
-- access control entries are stored in the table aces.  Access control entries
-- consist of references to object groups. If an ACE specifies a host, network
-- etc. explicitly, an artificial object group has to be created by the import
-- program. To give an example, consider the following ACE:

-- access-list OUTSIDE extended permit tcp host 209.165.201.8 host 209.165.200.225 eq www

-- Instead of referencing an object group this ACE specifies a host explicitly.
-- Therefore, an artificial object group of type network has to be created. By
-- convention, this object group will be called _host_209_165_201_8 and
-- contains only one object which is the host 209.165.201.8

-- The same applies to ports/services. To import the ACE which was given above
-- as an example an object group has to be created for the service www.

CREATE TABLE `firewalls` (
  `fwid` int(10) unsigned NOT NULL,
  `name` varchar(64) collate ascii_bin NOT NULL,
  PRIMARY KEY (`fwid`)
) ENGINE=MyISAM DEFAULT CHARSET=ascii COLLATE=ascii_bin;


--
-- Table structure for table `access_groups`
--

-- access_groups associate ACLs with interfaces

CREATE TABLE `access_groups` (
  `fwid` int(10) unsigned NOT NULL,
  `acl_name` varchar(64) collate armscii8_bin NOT NULL,
  `direction` enum('in','out') collate armscii8_bin NOT NULL,
  `interface` varchar(64) collate armscii8_bin NOT NULL,
  PRIMARY KEY  (`fwid`,`acl_name`,`direction`,`interface`)
) ENGINE=MyISAM DEFAULT CHARSET=armscii8 COLLATE=armscii8_bin;

-- --------------------------------------------------------

--
-- Table structure for table `aces`
```

```
--

-- 'og' stands for object group and is a foreign key into
-- object_groups
-- ace stands for access control entry

CREATE TABLE `aces` (
  `fwid` int(10) unsigned NOT NULL,
  `acl_name` varchar(64) collate ascii_bin NOT NULL,
  `rule_no` int(10) unsigned NOT NULL,
  `type` enum('permit','deny') collate ascii_bin NOT NULL,
  `protocol` varchar(16) collate ascii_bin NOT NULL,
  `srcIPog` varchar(64) collate ascii_bin NOT NULL,
  `srcSRVog` varchar(64) collate ascii_bin,
  `dstIPog` varchar(64) collate ascii_bin NOT NULL,
  `dstSRVog` varchar(64) collate ascii_bin,
  `icmptypeog` varchar(64) collate ascii_bin,
 PRIMARY KEY  (`fwid`,`acl_name`,`rule_no`)
) ENGINE=MyISAM DEFAULT CHARSET=ascii COLLATE=ascii_bin;

-- --------------------------------------------------------

--
-- Table structure for table `icmp_objects`
--

CREATE TABLE `icmp_objects` (
  `fwid` int(10) unsigned NOT NULL,
  `name` varchar(64) collate ascii_bin NOT NULL,
  `icmp_type` tinyint(3) unsigned NOT NULL,
 PRIMARY KEY  (`fwid`,`name`,`icmp_type`)
) ENGINE=MyISAM DEFAULT CHARSET=ascii COLLATE=ascii_bin;

-- --------------------------------------------------------

--
-- Table structure for table `network_objects`
--

CREATE TABLE `network_objects` (
  `fwid` int(10) unsigned NOT NULL,
  `name` varchar(64) collate ascii_bin NOT NULL,
  `startIP` int(10) unsigned NOT NULL,
  `endIP` int(10) unsigned NOT NULL,
 PRIMARY KEY  (`fwid`,`name`,`startIP`,`endIP`)
) ENGINE=MyISAM DEFAULT CHARSET=ascii COLLATE=ascii_bin;

-- --------------------------------------------------------

--
-- Table structure for table `object_groups`
--

CREATE TABLE `object_groups` (
  `fwid` int(10) unsigned NOT NULL,
  `name` varchar(64) collate ascii_bin NOT NULL,
  `description` varchar(255) collate ascii_bin default NULL,
  `type` enum('protocol','network','service','icmp-type') collate ascii_bin NOT NULL,
  `protocol` enum('tcp','udp','tcp-udp') collate ascii_bin default NULL,
 PRIMARY KEY  (`fwid`,`name`)
) ENGINE=MyISAM DEFAULT CHARSET=ascii COLLATE=ascii_bin;
```

```sql
-- --------------------------------------------------------

--
-- Table structure for table `port_objects`
--

CREATE TABLE `port_objects` (
  `fwid` int(10) unsigned NOT NULL,
  `name` varchar(64) collate ascii_bin NOT NULL,
  `minport` smallint(5) unsigned NOT NULL,
  `maxport` smallint(5) unsigned NOT NULL,
  PRIMARY KEY  (`fwid`,`name`,`minport`,`maxport`)
) ENGINE=MyISAM DEFAULT CHARSET=ascii COLLATE=ascii_bin;

-- --------------------------------------------------------

--
-- Table structure for table `protocol_objects`
--

CREATE TABLE `protocol_objects` (
  `fwid` int(10) unsigned NOT NULL,
  `name` varchar(64) collate ascii_bin NOT NULL,
  `protocol` tinyint(4) unsigned NOT NULL,
  PRIMARY KEY  (`fwid`,`name`,`protocol`)
) ENGINE=MyISAM DEFAULT CHARSET=ascii COLLATE=ascii_bin;


CREATE VIEW rules AS
        SELECT firewalls.name,acl_name,rule_no,type,protocol,
                src.startIP AS srcStartIP,src.endIP AS srcEndIP,
                srcports.minport AS srcMinPort,srcports.maxport AS srcMaxPort,
                dst.startIP AS dstStartIP,dst.endIP as dstEndIP,
                dstports.minport AS dstMinPort,dstports.maxport AS dstMaxPort,
                icmp_objects.icmp_type AS icmp_type
        FROM aces
                LEFT OUTER JOIN firewalls ON
                        firewalls.fwid = aces.fwid
                LEFT OUTER JOIN network_objects src ON
                        src.fwid = aces.fwid AND src.name = srcIPog
                LEFT OUTER JOIN network_objects dst ON
                        dst.fwid = aces.fwid AND dst.name = dstIPog
                LEFT OUTER JOIN port_objects srcports ON
                        srcports.fwid = aces.fwid AND srcports.name = srcSRVog
                LEFT OUTER JOIN port_objects dstports ON
                        dstports.fwid = aces.fwid AND dstports.name = dstSRVog
                LEFT OUTER JOIN icmp_objects ON
                        icmp_objects.fwid = aces.fwid AND icmp_objects.name = icmptypeog;
```

Listing A.4: Ruby–Skript für den Import von ACLs in eine relationale Datenbank

```ruby
require 'ipaddr'

# Map names of IP protocols to numbers. Compare to /etc/protocols
PROTOCOLS = {
 'ah' => 51, 'eigrp' => 88, 'esp' => 50, 'gre' => 47, 'icmp' => 1, 'igmp' => 2,
'igrp' => 9, 'ip' => 0, 'ipinip' => 4, 'nos' => 94, 'ospf' => 89, 'pcp' => 108,
'snp' => 109, 'tcp' => 6, 'udp' => 17, 'pim' => 103 }

# Map service names to port numbers. Compare to /etc/services
```

```
PORTS = { 'aol'=>5190, 'bgp'=>179, 'biff'=>512, 'bootpc'=>68, 'bootps'=>67, 'chargen'=>19,
'citrix-ica'=>1494, 'cmd'=>514, 'ctiqbe'=>2748, 'daytime'=>13, 'discard'=>9, 'domain'=>53,
'dnsix'=>195, 'echo'=>7, 'exec'=>512, 'finger'=>79, 'ftp'=>21, 'ftp-data'=>20, 'gopher'=>70,
'https'=>443, 'h323'=>1720, 'hostname'=>101, 'ident'=>113, 'imap4'=>143, 'irc'=>194,
'isakmp'=>500, 'kerberos'=>750, 'klogin'=>543, 'kshell'=>544, 'ldap'=>389, 'ldaps'=>636,
'lpd'=>515, 'login'=>513, 'lotusnotes'=>1352, 'mobile-ip'=>434, 'nameserver'=>42,
'netbios-ns'=>137, 'netbios-dgm'=>138, 'netbios-ssn'=>139, 'nntp'=>119, 'ntp'=>123,
'pcanywhere-status'=>5632, 'pcanywhere-data'=>5631, 'pim-auto-rp'=>496, 'pop2'=>109,
'pop3'=>110,
'pptp'=>1723, 'radius'=>1645, 'radius-acct'=>1646, 'rip'=>520, 'secureid-udp'=>5510,
'smtp'=>25, 'snmp'=>161, 'snmptrap'=>162, 'sqlnet'=>1521, 'ssh'=>22, 'sunrpc'=>111,
'syslog'=>514, 'tacacs'=>49, 'talk'=>517, 'telnet'=>23, 'tftp'=>69, 'time'=>37,
'uucp'=>540, 'who'=>513, 'whois'=>43, 'www'=>80, 'xdmcp'=>177}

ICMPTYPES = { 'echo-reply'=>0, 'unreachable'=>3, 'source-quench'=>4, 'redirect'=>5,
'alternate-address'=>6, 'echo'=>8, 'router-advertisement'=>9, 'router-solicitation'=>10,
'time-exceeded'=>11, 'parameter-problem'=>12, 'timestamp-request'=>13, 'timestamp-reply'=>14,
'information-request'=>15, 'information-reply'=>16, 'mask-request'=>17, 'mask-reply'=>18,
'conversion-error'=>31, 'mobile-redirect'=>32}

class ParseError < StandardError
end

# This class is a wrapper class around IO
# If reading a file line by line it provides
# the ability to "push back" a line to the input
# stream. The pushed back line will be returned
# at the next call to gets.
class UngetReader
    def initialize(s)
        @s = s
        @buf = nil
    end
    def gets
        tmp = @buf
        @buf = nil
        # If there's a line in the buffer, return this line
        # or else call gets of the IO class
        tmp or @s.gets
    end
    def ungets(line)
        # Save the line in the buffer. It will be returned
        # on the next call to gets
        @buf = line
    end
    def lineno
        # Get line number
        # If buffer is not empty, substract 1 from lineno
        @s.lineno - if @buf then 1 else 0 end
    end
end

# Super class for object groups
class ObjectGroup
end

# This class stores all the information of an object group
# of type network. Here's an example of an object group:
#
# object-group network sample
#  network-object 128.61.127.0 255.255.255.224
```

```
#  network-object 128.61.5.0 255.255.255.0
#  network-object host 127.2.5.3
#
#  If the object group references other object groups
#  (nested object groups) those references will be
#  resolved and the contents of the referenced object group
#  will be copied in the referencing object group
class NetworkObjectGroup < ObjectGroup
    OBJECTCOMMAND = 'network-object'
    OBJECTTYPE = 'network'
    # ALLONES is the netmask for host addresses
    ALLONES = IPAddr.new '255.255.255.255',Socket::AF_INET
    def initialize(name,params=nil)
        super(name)
    end
    def process_object(tokens)
        # This function will be called for every occurence
        # of 'network-object' in the configuration file
        # tokens will be an array of strings:
        # Example for a network: ['128.61.127.0','255.255.255.224']
        # Example for a single host: ['host','127.2.5.3']
        # compare with example above
        raise ParseError,
            "expecting two tokens after network-object" if
            tokens.length < 2
        if tokens[0] == 'host'
            r = IPRange.new IPAddr.new(tokens[1],Socket::AF_INET),ALLONES
        else
            r = IPRange.new IPAddr.new(tokens[0],Socket::AF_INET),
                IPAddr.new(tokens[1],Socket::AF_INET)
        end
        @objects |= [r]
        # Remove tokens we have read from array of tokens
        tokens.shift
        tokens.shift
    end
    # copy another object group into this object group
    def insert_og(og)
        @objects |= og.objects
    end
    def sqlinsert(fwid)
        # Generate SQL statements for this network object group
        s = String.new
        @objects.each { |r|
            min = r.address.to_i
            max = (r.address | (~r.mask)).to_i
            s += "INSERT INTO network_objects VALUES (#{fwid},'#{@name}',#{min},#{max});\n"
        }
        super(fwid,'NULL') + s
    end
    class IPRange
        # An IP range consists of an address and a netmask
        # Single hosts are mapped to an IP range where the netmask is 255.255.255.255
        attr_reader :address, :mask
        def initialize(address,mask)
            @address = address
            @mask = mask
        end
        def eql?(o)
            @address == o.address && @mask == o.mask
        end
```

```
        def hash
            @address.to_i + @mask.to_i
        end
        def inspect
            "#{@address.to_s}/#{@mask.to_s}"
        end
        def to_s
            inspect
        end
    end
    def NetworkObjectGroup.create_default_groups
        # There’s one default object group which is called any and includes all IP addresses
        og = new(’__any’)
        og.process_object([’0.0.0.0’,’0.0.0.0’])
        [og]
    end
    def NetworkObjectGroup.get_group_for_literal(tokens, fwconfig)
        raise ParseError,
            "expecting␣two␣tokens␣for␣definition␣of␣src/dst␣IP" if
            tokens.length < 2
        if tokens[0] == ’host’
            name = "__host_" + tokens[1].tr(’.’,’_’)
        else
            name = "__net_#{tokens[0].tr(’.’,’_’)}___#{tokens[1].tr(’.’,’_’)}"
        end
        if !og = fwconfig.get_og(NetworkObjectGroup,name)
            og = new name
            og.process_object(tokens)
            fwconfig.add_og(og)
        else
            2.times { tokens.shift }
        end
        return name
    end
end

# This class stores all the information of an object group
# of type service (ports). Here’s an example of such an object group:
#
# object-group service buzzcard-port-tcp tcp
#   port-object range 9000 9005
#   port-object eq telnet
#
#  If the object group references other object groups
#  (nested object groups) those references will be
#  resolved and the contents of the referenced object group
#  will be copied in the referencing object group
class ServiceObjectGroup < ObjectGroup
    OBJECTCOMMAND = ’port-object’
    OBJECTTYPE = ’service’
    attr_reader :protocol
    def initialize(name,params)
        super(name)
        # service object groups can be of three different types
        @protocol = case params[0]
                when ’udp’ then :udp
                when ’tcp’ then :tcp
                when ’tcp-udp’ then :tcp_udp
                else
                    raise ParseError,
                    "expecting␣either␣’udp’,’tcp’␣or␣’tcp-udp’␣" +
```

```
                    "as type of service object group"
            end
end
# Parse definitions of port ranges:
# Examples:
# "eq 21"
# "eq telnet"
# "lt 1024"
# "range 20 30"
def process_object(tokens)
    case tokens[0]
    when 'eq'
        r = [PortRange.new(tokens[1],tokens[1])]
    when 'lt'
        r = [PortRange.new(0,tokens[1].to_i-1)]
    when 'gt'
        r = [PortRange.new(tokens[1].to_i+1,65535)]
    when 'neq'
        r = [PortRane.new(0,tokens[1].to_i-1),
            PortRange.new(tokens[1].to_i+1,65535)]
    when 'range'
        r = [PortRange.new(tokens[1],tokens[2])]
        tokens.shift
    else
        raise ParseError,
            "expecting either 'eq', 'lt', 'gt', 'neq' or 'range' " +
            "for definition of port range"
    end
    2.times {tokens.shift}
    @objects |= r
end
# copy another object group into this object group
def insert_og(og)
    raise ParseError, "object group has different type" if og.protocol != @protocol
    @objects |= og.objects
end
def sqlinsert(fwid)
    s = String.new
    @objects.each { |r|
        s += "INSERT INTO port_objects VALUES (#{fwid},'#{@name}',#{r.min},#{r.max});\n"
    }
    super(fwid,"'#{if protocol == :tcp_udp then 'tcp-udp' else protocol end}'") + s
end
class PortRange
    attr_reader :min, :max
    def initialize(min,max)
        @min = PortRange.port2num(min)
        @max = PortRange.port2num(max)
    end
    def PortRange.port2num(port)
        if port.kind_of?(Integer) then return port end
        # Is port a decimal port number
        if port =~ /\A\d{1,5}\z/ and (0..65535).include?(port.to_i)
            port.to_i
        else
            # if not, it's the name of a service
            PORTS[port] or raise ParseError,"unknown service name #{port}"
        end
    end
    def eql?(o)
        @min == o.min && @max == o.max
```

```
        end
        def hash
            @min + @max
        end
        def inspect
            "from␣#{min}␣to␣#{max}"
        end
        def to_s
            inspect
        end
    end
    def ServiceObjectGroup.create_default_groups
        any_tcp = new('__any_tcp',['tcp'])
        any_tcp.process_object(['range','0','65535'])
        any_udp = new('__any_udp',['udp'])
        any_udp.process_object(['range','0','65535'])
        [any_tcp,any_udp]
    end
    def ServiceObjectGroup.get_group_for_literal(tokens,protocolno,fwconfig)
        # protocol is either TCP or UDP
        protocol = PROTOCOLS.index(protocolno)
        if ['lt','gt','eq','neq'].include? tokens[0] then
            port1 = PortRange.port2num(tokens[1])
            name = "__#{protocol}_#{tokens[0]}_#{port1}"
            cleanup = 2
        elsif tokens[0] == 'range' then
            port1 = PortRange.port2num(tokens[1])
            port2 = PortRange.port2num(tokens[2])
            name = "__#{protocol}_range_#{port1}_#{port2}"
            cleanup = 3
        else
            return nil
        end
        # check if pseudo object group already exists
        if !og = fwconfig.get_og(ServiceObjectGroup,name)
            # if not, create it
            og = new name,[protocol]
            og.process_object(tokens)
            fwconfig.add_og(og)
        else
            cleanup.times {tokens.shift}
        end
        return name
    end
end

class ICMPTypeObjectGroup < ObjectGroup
    OBJECTCOMMAND = 'icmp-object'
    OBJECTTYPE = 'icmp-type'
    def initialize(name,params=nil)
        super(name)
    end
    def process_object(tokens)
        raise ParseError, "expecting␣ICMP␣type" if tokens.length != 1
        raise ParseError, "unknown␣ICMP␣type␣'#{$tokens[0]}'" if
            !type = ICMPTYPES[tokens[0]]
        @objects |= [type]
        # Remove tokens we have read from array of tokens
        tokens.shift
    end
    # copy another object group into this object group
```

```ruby
    def insert_og(og)
        @objects |= og.objects
    end
    def sqlinsert(fwid)
        # Generate SQL statements for this network object group
        s = String.new
        @objects.each { |r|
            s += "INSERT INTO icmp_objects VALUES (#{fwid},'#{@name}',#{r});\n"
        }
        super(fwid,'NULL') + s
    end
    def ICMPTypeObjectGroup.create_default_groups
        Array.new
    end
    def ICMPTypeObjectGroup.get_group_for_literal(tokens, fwconfig)
        raise ParseError, "expecting icmp type" if tokens.length != 1
        name = "__#{tokens[0]}"
        if !og = fwconfig.get_og(ICMPTypeObjectGroup,name)
            og = new name
            og.process_object(tokens)
            fwconfig.add_og(og)
        else
            tokens.shift
        end
        return name
    end
end

# Super class for NetworkObjectGroup, ServiceObjectGroup and ICMPTypeObjectGroup
class ObjectGroup
    TYPES = {
        'network' => NetworkObjectGroup,
        'service' => ServiceObjectGroup,
        'icmp-type' => ICMPTypeObjectGroup,
        'protocol' => :ignore
    }
    OBJECTCOMMAND = '<click-here-and-type-object-type>-object';
    attr_reader :name,:objects,:description
    protected :objects
    def initialize(name)
        @name = name
        @objects = Array.new
    end
    def parse(r,fwconfig)
        while line = r.gets
            tokens = line.split
            case tokens[0]
            when 'description'
                @description = line[/\s*description (.*)/,1]
            when self.class::OBJECTCOMMAND
                tokens.shift
                process_object(tokens)
            when 'group-object'
                insert_og fwconfig.get_og(self.class,tokens[1])
            else
                r.ungets line
                break
            end
        end
    end
    def print
```

```
        puts '---------'
        puts "Name of Group: #{@name}"
        puts "Description: #{@description}" if @description
        puts "#{@protocol}" if self.kind_of?(ServiceObjectGroup)
        puts @objects
        puts '---------'
    end
    def sqlinsert(fwid,protocol)
        desc = if @description then @description.gsub("'","\\\\'") else nil end
        "INSERT INTO object_groups VALUES (#{fwid},'#{@name}'," +
            if desc then "'#{desc}'" else "NULL" end + "," +
            "'#{self.class::OBJECTTYPE}',#{protocol});\n"
    end
    def ObjectGroup.parse(line,r,fwconfig)
        tokens = line.split
        tokens.shift
        raise ParseError,
            "unknown type of object group: #{tokens[0]}" if !t = TYPES[tokens[0]]
        return if t == :ignore
        tokens.shift
        name = tokens[0]
        tokens.shift
        og = t.new(name,tokens)
        og.parse(r,fwconfig)
        fwconfig.add_og og
    end
end

# class for Access Control Lists
class ACL
    attr_reader :name
    def initialize(name)
        @name = name
        @aces = Array.new
    end
    # add Access Control Entry to ACL
    def add_ace(ace)
        @aces << ace
    end
    def to_s
        @aces.join($/)
    end
    def sqlinsert(fwid)
        s = String.new
        i = 1
        @aces.each { |ace|
            s += ace.sqlinsert(fwid,name,i)
            i = i + 1
        }
        return s
    end

end

# Class for Access Contol Entries
class ACE
    TYPEOFRULES = { 'permit' => :permit, 'deny' => :deny }
    def initialize(fwconfig)
        @type_of_rule = nil
        # fwconfig contains a reference to the object of FirewallConfig
        @fwconfig = fwconfig
```

```
end
def ACE.parse(line,r,fwconfig)
    tokens = line.split
    tokens.shift
    raise ParseError, "expecting name of ACL" if !name_of_acl = tokens[0]
    tokens.shift
    ace = new(fwconfig)
    return unless ace.process_ace(tokens)
    # Check if there's already an ACL with this name
    if (!acl = fwconfig.get_acl(name_of_acl))
        # if not, create the ACL object
        fwconfig.add_acl(acl = ACL.new(name_of_acl))
    end
    # Add ACE to ACL
    acl.add_ace(ace)
end
def parse_protocol(tokens)
    # Is protocol a decimal number?
    p = if tokens[0] =~ /\A\d{1,3}\z/ and (0..255).include?(tokens[0].to_i)
        tokens[0].to_i
    else
        # if not, it's the name of a service
        PROTOCOLS[tokens[0]] or raise ParseError,"unknown protocol name #{tokens[0]}"
    end
    tokens.shift
    return p
end
def parse_endpoint(tokens)
    case tokens[0]
    when'object-group'
        name = tokens[1]
        raise ParseError,
            "network object group #{name} unknown" if
                !@fwconfig.get_og(NetworkObjectGroup,name)
        tokens.shift
        tokens.shift
        return name
    when 'any'
        tokens.shift
        return '__any'
    else
        NetworkObjectGroup.get_group_for_literal(tokens,@fwconfig)
    end
end
def parse_service(tokens)
    case tokens[0]
    when 'object-group'
        name = tokens[1]
        if og = @fwconfig.get_og(ServiceObjectGroup,name)
            raise ParseError, "service object group is of a different type" if
                (og.protocol == :tcp and @protocol != 6) or
                (og.protocol == :udp and @protocol != 17)
            2.times {tokens.shift}
            name
        else nil end
    else
        ServiceObjectGroup.get_group_for_literal(tokens,@protocol,@fwconfig)
    end
end
def parse_icmptype(tokens)
    return nil if tokens.length == 0
```

```
        case tokens[0]
        when 'object-group'
            name = tokens[1]
            if og = @fwconfig.get_og(ICMPTypeObjectGroup,name)
                2.times {tokens.shift}
                name
            else
                raise ParseError, "icmp-type␣object␣group␣#{name}␣does␣not␣exist."
            end
        else
            ICMPTypeObjectGroup.get_group_for_literal(tokens,@fwconfig)
        end
    end
    def process_ace(tokens)
        return false if tokens[0] != 'extended'
        tokens.shift
        raise ParseError, "expecting␣'deny'␣or␣'permit'" if
            !@type_of_rule = TYPEOFRULES[tokens[0]]
        tokens.shift
        @protocol = parse_protocol(tokens)
        @src_og = parse_endpoint(tokens)
        if @protocol == 6 or @protocol == 17
            @src_service = parse_service(tokens) || "__any_#{PROTOCOLS.index(@protocol)}"
        end
        @dst_og = parse_endpoint(tokens)
        if @protocol == 6 or @protocol == 17
            @dst_service = parse_service(tokens) || "__any_#{PROTOCOLS.index(@protocol)}"
        end
        if @protocol == 1
            @icmptype_og = parse_icmptype(tokens)
        end
        raise ParseError, "unparsed␣tokens␣at␣the␣end␣of␣ACE" if
            tokens.length!=0 and tokens[0]!='log'
        return true
    end
    def inspect
        "ACE␣#{@type_of_rule}␣protocol␣#{PROTOCOLS.index(@protocol)}" +
            "␣src␣#{@src_og}#{if␣@src_service␣then␣':'␣+␣@src_service␣end}" +
            "␣dst␣#{@dst_og}#{if␣@dst_service␣then␣':'␣+␣@dst_service␣end}"
    end
    def to_s
        inspect
    end
    def sqlinsert(fwid,name_of_acl,rule_no)
        "INSERT␣INTO␣aces␣VALUES␣(#{fwid},'#{name_of_acl}',#{rule_no}," +
        "'#{@type_of_rule}','#{PROTOCOLS.index(@protocol)||@protocol}'," +
        "'#{@src_og}'," + if @src_service then "'#{@src_service}'" else "NULL" end + "," +
        "'#{@dst_og}'," + if @dst_service then "'#{@dst_service}'" else "NULL" end + "," +
        if @icmptype_og then "'#{@icmptype_og}'" else "NULL" end + ");\n"
    end
end

class AccessGroup
    def process_access_group(tokens,fwconfig)
        raise ParseError, "expecting␣4␣tokens␣after␣'access-group'" if tokens.length != 4
        raise ParseError, "expecting␣keyword␣'interface'" if tokens[2]!='interface'
        @name_of_acl,@direction,dummy,@interface = tokens
        raise ParseError,
            "direction␣must␣be␣either␣'in'␣or␣'out'" if !['in','out'].include?(@direction)
        raise ParseError,
            "unknown␣ACL␣#{@name_of_acl}" if !fwconfig.get_acl(@name_of_acl)
```

```
    end
    def AccessGroup.parse(line,r,fwconfig)
        ag = new
        tokens = line.split
        tokens.shift
        ag.process_access_group(tokens,fwconfig)
        fwconfig.add_ag(ag)
    end
    def print
        puts "ACL #{@name_of_acl} direction #{@direction} interface #{@interface}"
    end
    def sqlinsert(fwid)
        "INSERT INTO access_groups VALUES (#{fwid},'#{@name_of_acl}'," +
            "'#{@direction}','#{@interface}');\n"
    end
end

class FirewallConfig
    COMMANDS = {
        'object-group' => ObjectGroup.method(:parse),
        'access-list' => ACE.method(:parse),
        'access-group' => AccessGroup.method(:parse)
    }
    def parse(input)
        r = UngetReader.new(input)
        while line = r.gets
            line.chomp!
            line.strip!
            next if line =~ /^[!:]/
            command = line.split[0]
            begin
                if c = COMMANDS[command] then c.call(line,r,self) end
            rescue ParseError,ArgumentError
                $stderr.puts "Parse error on line #{r.lineno}: " + $!
                raise
            end
        end
    end
    def initialize(name,id)
        @name = name
        @id = id
        @ogs = {
            NetworkObjectGroup => Hash.new,
            ServiceObjectGroup => Hash.new,
            ICMPTypeObjectGroup => Hash.new
        }
        @ogs.each_key do |c|
            c.create_default_groups.each do |og|
                add_og og
            end
        end
        @acls = Hash.new
        @access_groups = Array.new
    end
    def add_og(og)
        raise ParseError, "Object group #{og.name} already exists" if
            @ogs[og.class][og.name]
        @ogs[og.class][og.name] = og
    end
    def get_og(type, name)
        @ogs[type][name]
```

```
    end
    def add_acl(acl)
        raise ParseError, "ACL␣#{acl.name}␣already␣exists" if @acls[acl.name]
        @acls[acl.name] = acl
    end
    def get_acl(name)
        @acls[name]
    end
    def add_ag(access_group)
        @access_groups << access_group
    end
    def print
        puts "********␣Network␣object␣groups:"
        @ogs[NetworkObjectGroup].each_value { |og| og.print }
        puts "********␣Service␣object␣groups:"
        @ogs[ServiceObjectGroup].each_value { |og| og.print }
        puts "********␣ICMP␣Type␣object␣groups:"
        @ogs[ICMPTypeObjectGroup].each_value { |og| og.print }
        puts "********␣ACLs"
        @acls.each { |name,acl|
            puts "+++++++␣ACL:␣#{name}"
            puts acl
        }
        puts "********␣Access␣Groups"
        @access_groups.each { |ag| ag.print }
    end
    def sqlinsert
        nogs = String.new
        sogs = String.new
        itogs = String.new
        acls = String.new
        access_groups = String.new
        @ogs[NetworkObjectGroup].each_value { |og| nogs+=og.sqlinsert(@id) }
        @ogs[ServiceObjectGroup].each_value { |og| sogs+=og.sqlinsert(@id) }
        @ogs[ICMPTypeObjectGroup].each_value { |og| itogs+=og.sqlinsert(@id) }
        @acls.each_value { |acl|
            acls += acl.sqlinsert(@id) }
        @access_groups.each { |ag| access_groups += ag.sqlinsert(@id) }
        "--␣Firewall:\n" +
        "INSERT␣INTO␣firewalls␣VALUES␣(#{@id},'#{@name}');\n" +
        "--␣Network␣object␣groups:\n" + nogs +
        "--␣Service␣object␣groups:\n" + sogs +
        "--␣ICMP␣Type␣object␣groups:\n" + itogs +
        "--␣ACLs:\n" + acls +
        "--␣Access␣Groups:\n" + access_groups
    end
end
```

Listing A.5: Ruby–Skript zur Abfrage der Datenbank

```
#!/usr/bin/ruby
require 'mysql'
require 'ipaddr'
require 'FirewallConfig.rb'

# Create DB connection
$db = Mysql.new('localhost','fwconfig',nil,'fwconfig')
$db.query_with_result = false

# Check the database to find out if a given connection is allowed or not
def dbquery(fwname,acl,protocol,src,srcport,dst,dstport)
```

```
        r = $db.query( q =
                "select type from rules where name='#{fwname}' " +
                "AND acl_name='#{acl}' AND srcStartIP <= #{src.to_i} "+
                "AND srcEndIP >= #{src.to_i} AND dstStartIP <= #{dst.to_i} " +
                "AND dstEndIP >= #{dst.to_i} " +
                "AND ( (protocol = 'ip') " +
                "OR (protocol='#{protocol}' AND srcMinPort <= #{srcport} " +
                "AND srcMaxPort >= #{srcport} AND dstMinPort <= #{dstport} " +
                "AND dstMaxPort >= #{dstport}) ) " +
                " ORDER BY rule_no LIMIT 1")
        row = r.fetch_row
        if row
                type_of_rule = row[0]
                return type_of_rule.to_sym
        else
                return nil
        end
end

$db.set_server_option(Mysql::OPTION_MULTI_STATEMENTS_ON)
$db.query_with_result = true

result = dbquery('name_of_firewall','inbound','tcp',IPAddr.new('212.201.100.133'),80,IPAddr.new(

$db.close
```

Literaturverzeichnis

[CS 05] CISCO SYSTEMS, INC.: *NetFlow Accounting on a Catalyst 6500 SUP1*, 2005. http://www.cisco.com/warp/public/477/netflow-22268.html, Document ID: 22268.

[CS 07a] CISCO SYSTEMS, INC.: *Cisco IOS NetFlow - Introduction*, 2007. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.

[CS 07b] CISCO SYSTEMS, INC.: *Configuring IP Access Lists*, 2007. http://www.cisco.com/warp/public/707/confaccesslists.pdf, Document ID: 23602.

[CS 07c] CISCO SYSTEMS, INC.: *NetFlow Services Solutions Guide*, 2007. http://www.cisco.com/univercd/cc/td/doc/cisintwk/intsolns/netflsol/nfwhite.htm.

[ESH 02] ELAM, GUNILLA, TOMAS STEPHANSON und NIKLAS HANBERGER: *Warriors of the Net*, 2002. http://www.warriorsofthe.net/.

[mwn 06] *Das Münchner Wissenschaftsnetz (MWN). Konzepte, Dienste, Infrastrukturen, Management*, 2006. http://www.lrz-muenchen.de/services/netz/mwn-netzkonzept/mwn-netzkonzept.pdf.

[SoFe 02] SOMMER, ROBIN und ANJA FELDMANN: *NetFlow: Information Loss or Win?* In: *Proceedings of ACM SIGCOMM Internet Measurement Workshop (IMW) 2002*. ACM Press, 2002, citeseer.ist.psu.edu/article/sommer02netflow.html .

[Stev 93] STEVENS, W. RICHARD: *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.