Routing Caches in Software Packet Forwarding Devices

Christian Thieme
Tutor: Daniel G. Raumer und Paul Emmerich
Seminar Future Internet SS2015
Chair for Network Architectures and Services
Department of Computer Science, Technische Universität München
E-mail: christian.thieme@tum.de

ABSTRACT
Due to the rise of internet capable devices and the need to route their traffic, it is a challenge for every routing device to process the massive load in an acceptable period of time. Therefore to speed up the process of routing and to avoid the expenses for frequently upgrading Router hardware, the route cache has been invented. The route cache stores recently used routes and prevents time expensive lookups in the full Forwarding Information Base. This paper describes the packet forwarding process implemented in the Linux kernel. While the focus lies on the implemented routing cache in the Linux kernel, which was removed in version 3.7. Within the explanations of the routing cache behaviors, the Garbage Collector is also taken into account in regards of optimization and efficiency. It is a challenge to make the routing cache perform efficiently in every best- and worst-case traffic scenario. Also denial of service attacks specifically directed at the routing cache and the Garbage Collector are addressed as well as the cache hiding problem. Additionally, the open source software Open vSwitch in regards of the routing cache is discussed.

Keywords
Routing Cache, Packet Forwarding, Open vSwitch, Denial of Service, Linux Kernel, Forwarding Information Base, Cache Hiding

1. EINLEITUNG
Der Linux Kernel ist ein inzwischen weit verbreiteter Betriebssystemkernel und wurde 1991 von Linus Torvalds entwickelt [1]. Er ist als Teil von Linux Distributionen als auch in eingebetteten Systemen, wie z.B. Routern zu finden. Der Kernel bietet viele unterschiedliche grundlegende Funktionalitäten. Eine davon ist die Verarbeitung und Weiterleitung von Datenpaketen in Netzwerken. Um diesen Prozess möglichst effizient und zudem kostengünstig gestalten, ist in den Versionen vor 3.7 ein Routing Cache enthalten. Dieser wurde jedoch nach zwei Jahren andauernder Vorbereitung entfernt [17]. Dieses Paper beschäftigt sich damit, grundlegende Funktionen des Routing Prozesses im Linux Kernel zu erläutern und verständlich darzustellen. Es wird größtenteils auf IPv4 in Verbindung mit Linux Routing eingegangen. Im Folgenden wird unter Punkt 2 erklärt, wie das Verarbeiten und Weiterleiten von Paketen in Linux gehandhabt wurde, sowie die grobe Struktur dieses Prozesses. Punkt 3 geht näher auf den Routing Cache ein und erläutert auch Gründe, warum dieser entfernt wurde. Punkt 4 beschäftigt sich mit der Open Source Software Open vSwitch, welche weiterhin einen Routing Cache implementiert und diskutiert mögliche Gründe dafür. Punkt 5 gibt eine Zusammenfassung über die vorgestellten Ergebnisse.

2. PACKET FORWARDING
Packet Forwarding in Kern Internet Routern ist ein extrem herausfordernder Prozess. Wird ein IP Paket empfangen, haben Router nur ein paar Nanosekunden, das Paket zu puffern, den Longest Prefix Match, welcher das Ziel des Paketes bestimmt, zu ermitteln und das Paket an das korrespondierende aussendende Interface weiterzuleiten [10].

In der Grafik 1 ist eine Abstraktion des Ablaufes angegeben, welcher durchlaufen wird, wenn ein Paket an einem (virtuellen) Interface anliegt. Im Folgenden wird nun (approximativ) der Ablauf des IP Forwardings im Linux Kernel betrachtet.

Soll ein Paket, welches am Interface angekommen ist, verarbeitet werden, so wird im Empfangsmedium ein Interrupt ausgelöst. Das Empfangsmedium allokiert dann Speicherplatz für das anliegende Paket und übergibt dieses dem Bus, um es im allokierten Speicher abzulegen. Das Paket wird als nächstes der Link Schicht übergeben, welche das Paket in die Backlog Queue einfügt. Das Netzwerk Flag für den nächsten ‚bottom-half run[1]‘ wird markiert und das Empfangsmedium kehrt zu seinem vorherigen Prozess zurück. Wenn der Process Scheduler das nächste Mal läuft, bemerkt dieser, dass es Netzwerkaufgaben gibt, die bearbeitet werden sollen. Die Funktion *net_bh* entnimmt der Backlog Queue das Paket, vergleicht das Internet Protokoll und gibt das Paket weiter an die *receive* Funktion. Es wird nun auf Fehler untersucht. Das Paket wird nun entweder der Transport Schicht übergeben, falls es an diesen Host gesendet wurde, oder es bleibt auf der Netzwerk Schicht und wird geroutet [7]. Wie der Routing Prozess funktioniert und welche Besonderheiten es dort gibt, ist ausführlich in der Sektion 3 erläutert. Muss das Paket nun weitergeleitet werden, wird es überprüft und gegebenenfalls eine ICMP Fehlernachricht an den Sender übermittelt, falls ein Fehler auftritt [7]. Das Paket wird nun in einen neuen Puffer kopiert und gegebenenfalls fragmentiert, falls dies notwendig ist [7]. Dies kann z.B. der Fall sein, wenn die MTU (Maximum Transmission Unit) für das Zielnetzwerk, in welches das Paket geschickt werden soll, überschritten wird. Außerdem werden an dieser Stelle die MAC Adressen des Pakets verändert. Die ehemalige Ziel MAC Adresse wird zur neuen Quell MAC Adresse und die neue Ziel Adresse wird dem Routing Prozess entnommen. Schließlich wird es wieder an die Sicherungsschicht übergeben, welche das Paket

[1] Die Unterbrechungsroutine für ein Gerät ist oft in zwei Teile gespalten. Die ‘top half’ Routine ist die Routine, welche tatsächlich auf Unterbrechungen reagiert, und die anstehende Aufgabe zur Bearbeitung einplant. Die ‘bottom-half’ der Unterbrechungsroutine ist die Routine, welche die Aufgabe tatsächlich bearbeitet [9].

doi: 10.2313/NET-2015-09-1_07

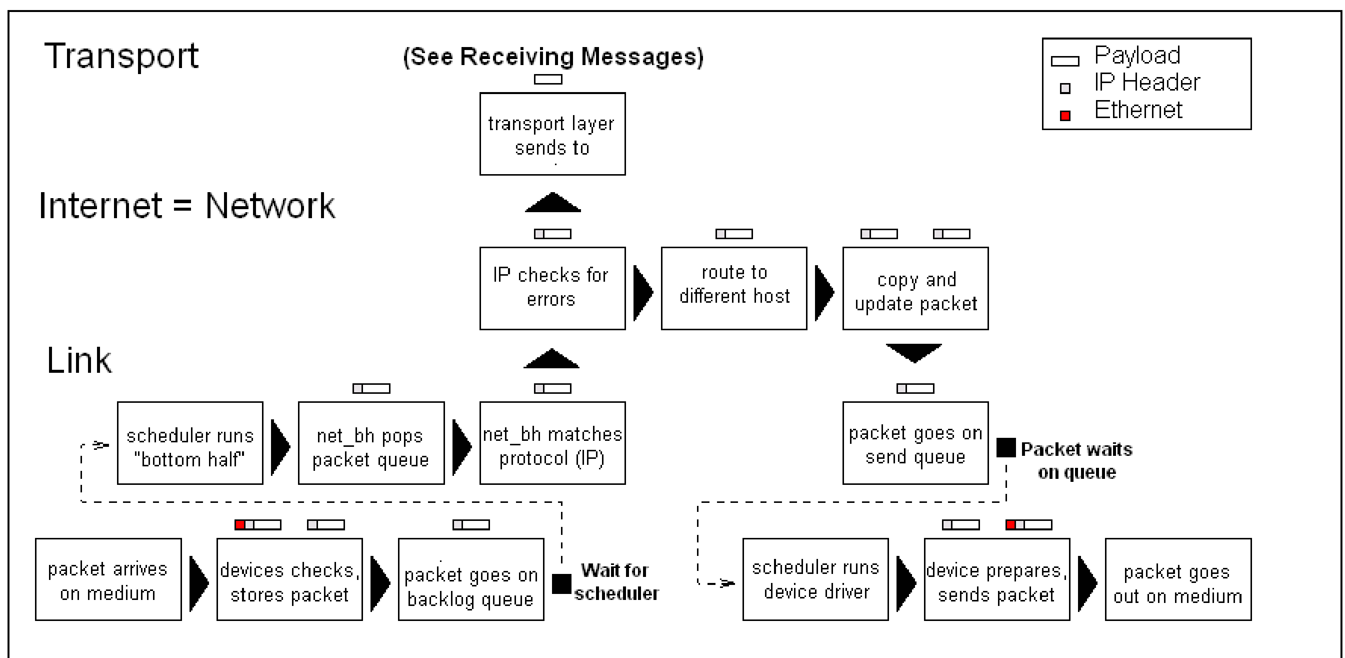

Abbildung 1: Abstraktes Diagramm vom Ablauf des IP Forwarding Prozesses im Linux Kernel. Quelle: [7]

in die Sendewarteschlange des Gerätes einreiht und sicherstellt, dass das Sendergerät weiß, dass es Verkehr zu senden hat [7]. Schlussendlich ordert das Sendegerät (wie beispielsweise eine Netzwerkkarte) seinen Bus an, das bzw. die Paket(e) zu senden [7].

Packet Forwarding kann sich auch auf andere Protokolle beziehen wie z.B. MPLS, ATM und weitere, welche hier nicht genauer behandelt werden.

3. ROUTING CACHE

3.1 Routing in Linux

Linux verwaltet(e) drei Komponenten, um Daten zu routen – eine für Computer, welche direkt mit dem Host verbunden sind (beispielsweise via LAN) und zwei für Computer, zu welchen nur indirekt eine Verbindung besteht [8].

Die Nachbartabelle beinhaltet Informationen über Computer, die physisch mit dem Host verbunden sind. Er beinhaltet Informationen darüber, über welches Gerät welcher Nachbar und welche Protokolle verwendet werden sollen, um Daten auszutauschen. Linux verwendet das Address Resolution Protocol (ARP), um diese Tabelle zu aktualisieren und zu verwalten [8].

Des Weiteren nutzt Linux zwei komplexe Komponenten von Routing Tabellen, um IP Adressen zu verwalten: eine Forwarding Information Base (FIB) mit Einträgen zu allen möglichen Adressen und einen Routing Cache gefüllt mit Daten von häufig genutzten Routen. Wenn ein IP Paket an einen weiter entfernten Host gesendet werden muss, überprüft die Netzwerkschicht zuerst den Routing Cache nach einem Eintrag mit passender Quelladresse, Zieladresse und passendem Diensttypen. Falls es dort einen solchen Eintrag gibt, wird er benutzt. Falls nicht, wird die Routing Information von der komplexeren (aber langsameren) FIB abgefragt, ein neuer Cache Eintrag mit den gefunden Daten wird generiert und der neue Eintrag dann benutzt [8].

Wie die FIB und der Routing Cache initialisiert werden, d.h. mit Einträgen bestückt werden, ist implementationsspezifisch von Anbieter zu Anbieter unterschiedlich und wird hier nicht genauer behandelt. Während die FIB Einträge semi-permanent sind (sie ändern sich nur wenn Router online oder offline gehen), verbleiben Einträge im Routing Cache nur so lange bis sie veraltet sind [8]. Liu et al. [11] stellen in ihrer Arbeit ein effizientes FIB Caching unter der Nutzung von minimalen nicht-überlappenden Präfixen vor.

3.2 Erklärung und Nutzen Routing Cache

Um mit eingehenden und ausgehenden IP Datagrammen umzugehen, muss der Kernel die Routing Tabellen abrufen [3]. Obwohl dies trivial erscheinen könnte, müssen vor der Weiterleitung einige Fragen beantwortet werden:

- Scheinen Quell- und Zieladresse valide zu sein?
- Ist die Quelladresse eine 'Martian Address'?[2]
- Ist die Zieladresse die eigene oder soll das Paket weitergeleitet werden?
- Welche Routing Tabelle soll benutzt werden?
- Passt diese Zieladresse zu dieser Route?

[2] ‚Martian Addresses' sind Adressen die nicht als Quelladressen benutzt werden können, entweder weil sie für besondere Zwecke (so wie Multicast Adressen) reserviert sind oder wegen der Nutzung von ‚reverse path filtering', welches überprüft, ob das auf einem Interface erhaltene Paket auf demselben Interface beantwortet werden müsste, wie in RFC 3704 definiert [3].

- Kann der zu nutzende Gateway momentan kontaktiert werden? [3]

Diese Überprüfungen können Zeit konsumierend sein. Um diese für jedes Paket zu vermeiden, verwaltet Linux einen Routing Cache, welcher abgefragt wird, bevor eine reguläre Suche gestartet wird und nach jeder dieser aktualisiert wird [3]. Wird eine Regel aus dem Routing Cache getroffen, müssen die oben genannten Fragen nicht mehr beantwortet werden, da diese Routen in der Regel erst kürzlich genutzt wurden und die benötigten Informationen im Puffer gespeichert sind.

Der Routing Cache ist die schnellste Methode, die Linux hat, um eine Route zu finden. Er behält jede Route, die momentan benutzt wird oder kürzlich benutzt wurde, in einer Hashtabelle [8]. Adressen werden mittels einer Hashfunktion auf Buckets abgebildet. Jeder Bucket enthält eine Liste, welche auch als Kette bezeichnet wird. Die Routen in den Ketten sind sortiert, am häufigsten genutzte Routen sind am Anfang, und besitzen eine Zeituhr und Zähler, welche sie von der Tabelle entfernen, wenn sie länger nicht mehr in Gebrauch waren [8].

Wird eine Routing Information benötigt, wird der passende Hashbucket durchsucht und die Kette von gepufferten Routen durchsucht bis ein Treffer gefunden wird. Dann wird das Paket auf diese Route geschickt. Falls es keinen Treffer im Routing Cache gibt, wird die FIB durchsucht. Gibt es dort einen Treffer, werden die erforderlichen Informationen ermittelt und die neue Route dem Routing Cache hinzugefügt. Falls es auch dort keine Treffer gibt, wird das Paket an eine default Route weitergeleitet (z.B. an einen Router in einer höheren Instanz). Pakete werden verworfen, falls unzulässige Quell- bzw. Zieladressen verwendet werden. So wird ein Home-Router Pakete verwerfen, die aus dem globalen Internet kommen und als Quelladresse die Broadcastadresse 255.255.255.255 haben.

3.2.1 Cache Hiding Problem

Linux nutzt das Longest Prefix Matching Verfahren, um Zieladressen mit gepufferten Adressen im Routing Cache (aber auch in der FIB) zu vergleichen. Bei dem Vergleich der Adressen wird die aufeinanderfolgende Anzahl an übereinstimmenden Bits (beginnend bei links) zweier IP Adressen gezählt und ab einer bestimmten Anzahl an Übereinstimmungen als valide gewertet.

Um den Longest Prefix Match effizient zu gestalten, nutzt Linux die *LC-Trie* (auch *Patricia Tree*) [14] Datenstruktur (für IPv6 wird ein *Radix Baum* verwendet). Beim Präfix Vergleich im Routing Cache kann möglicherweise auch das *Cache-Hiding* Problem auftreten. FIB Pufferung unterscheidet sich von traditionellen Puffer Mechanismen – selbst wenn für ein Paket ein passendes Präfix im Cache vorhanden ist, kann es trotzdem sein, dass es nicht der korrekte Eintrag ist, um das Paket weiterzuleiten, wenn es ein längeres passenderes Präfix in der FIB gibt [11]. Um das Problem anschaulicher zu gestalten, sind in den Tabellen 2 und 3 eine stark vereinfachte FIB und ein dazugehöriger Routing Cache abgebildet. Zum einfacheren Verständnis werden nur 8-bit lange Adressen betrachtet.

Tabelle 2: Beispieleinträge für eine FIB.

Kürzel	Präfix	Nächster Hop
A	11/2	1
B	101011/6	2
C	10101/5	4

Tabelle 3: Beispieleinträge für einen Routing Cache zur FIB in Tabelle 3.

Kürzel	Präfix	Nächster Hop
A	11/2	1
C	10101/5	4

Ausgehend von der FIB in Tabelle 3 und dem dazugehörigen Routing Cache in Tabelle 4 soll ein Paket mit der Zieladresse ***11010101*** geroutet werden. Zunächst wird der Routing Cache durchsucht und sogleich ein Treffer mit dem Eintrag beim Kürzel ***A*** erzielt, da ***A*** das mit ***11/2*** das längste passende Präfix zu der gesuchten Adresse besitzt. Das Paket wird nun über den nächsten ***Hop 1*** weitergeleitet. Soweit tritt kein Problem auf. Angenommen ein weiteres Paket mit der Zieladresse ***10101110*** soll weitergeleitet werden. Zunächst wird wieder der Routing Cache nach einem passenden Präfix durchsucht. Der Eintrag mit Kürzel ***C*** wird nun ausgewählt, da er mit den ersten 5 übereinstimmenden Bits der beste Eintrag im Routing Cache ist. Durch den Cache Eintrag ***C*** wird allerdings der passendere Eintrag in der FIB übergangen. Eintrag ***B*** in der FIB (Tabelle 3) hätte, statt 5, 6 übereinstimmende Bits und wäre somit die bessere Wahl gewesen. Das Paket wird also nun sozusagen fälschlicher Weise an den nächsten ***Hop 4*** statt ***2*** gesendet. Damit wird der passendere Eintrag in der FIB durch den Eintrag im Routing Cache versteckt.

3.2.2 Garbage Collector

Um den Route Cache angemessen zu verwalten und veraltete Routen, welche zu lange ungenutzt im Cache verweilen, zu löschen, nutzt Linux den Garbage Collector (im Folgenden GC genannt). Der GC spielt eine extrem wichtige Rolle in der Effizienz des Routing Caches. In durch default Einstellungen oder durch den Benutzer festgelegten Intervallen durchläuft der GC alle Einträge des Routing Caches. Zunächst evaluiert der GC die Situation, indem er die momentane Größe des Caches mit seinem konfigurierten Wert vergleicht. Ist die Anzahl der Cache Einträge kleiner oder gleich des Produkts von *rhash_entries* und *gc_elasticity,* wird der GC versuchen, ***höchstens die Hälfte der Differenz*** aus dem Routing Cache zu entfernen, auch wenn dadurch veraltete Einträge im Cache verbleiben sollten. Enthält beispielsweise der Routing Cache 1,5 Mio Einträge und im GC ist der Wert 2 Mio Einträge eingestellt (*rhash_entries* 400.000 und *gc_elasticity* 5), wird er versuchen höchstens (2 Mio – 1,5 Mio) / 2 = 250.000 Einträge zu löschen. Erreicht der GC sein Maximum an zu löschenden Einträgen bevor er den kompletten Cache durchlaufen ist, merkt er sich, wo er aufgehört hat und startet bei seinem nächsten Aufruf an der gemerkten Stelle. Welche Einträge vom GC entfernt werden, hängt von dem eingestellten *gc_timeout* Wert ab. Nur wenn das Alter des ersten Eintrages in einer Kette kleiner als *gc_timeout* wird der Eintrag beibehalten. Wenn es der zweite Eintrag ist, wird er nur behalten, wenn er halb so alt ist wie *gc_timeout*. Der dritte nur dann noch, wenn das Alter unter einem Viertel von *gc_timeout* ist (vlg. [3]). Ob ein Eintrag behalten wird, entscheidet sich also unter der Bedingung ob *„Alter des Eintrags“ < gc_timeout / (2^“Platz in der Kette“)* erfüllt ist. Wobei zu beachten ist, dass der Platz in der Kette bei 0 beginnt und aufsteigend nummeriert wird. Der GC bevorzugt kurze Ketten. Überschreitet die Größe des Routing Caches allerdings das Produkt der im GC eingestellten Werte, wechselt der GC in einen aggressiven Modus. Im aggressiven Modus wird der GC

versuchen, mindestens *rhash_entries* zu entfernen. Der GC wird aufgerufen, wenn ein neuer Cache Eintrag hinzugefügt werden soll und die Anzahl der Cache Einträge *gc_thresh* übersteigt. Mit *gc_interval* lässt sich zudem ein reguläres Intervall einstellen, in welchem der GC aufgerufen werden soll. Der GC kann nur aufgerufen werden, wenn die Anzahl der Einträge im Routing Cache *max_size* (eine Einstellung für die Mindestgröße des Caches) übersteigt und der GC nicht innerhalb der letzten *gc_interval_ms* Millisekunden bereits aufgerufen wurde. Weiterführende und detailliertere Beschreibungen des GC und wie man diesen optimieren kann, können im Blog von Bernat [3] in Erfahrung gebracht werden.

Ist der GC unpassend für das vorliegende Netzwerk eingestellt, können unterschiedliche und möglicherweise unerwünschte Ereignisse eintreten. Beispielsweise könnte der Routing Cache sich zu langsam oder mit zu wenigen Einträgen füllen und produziert damit viele C*ache Misses,* welche die Effizienz des Routings negativ beeinflussen können, da für einen *Cache Miss* anstatt einer, beide Routing Tabellen abgefragt werden müssen und die FIB manchmal in langsamerem Speicher abgelegt ist. Das Hauptproblem beim Konfigurieren des Routing Caches bzw. des GCs ist, dass sie abhängig davon sind, welcher und wie viel Verkehr zu Routen ist. Die Effizienz des Routing Caches ist somit von äußeren Umständen abhängig und somit von außen kontrollierbar (vgl. [13]). Somit gibt es bisher im Linux Kernel keine allgemein optimale Konfiguration des Routing Caches, da der Linux Kernel in verschiedenen Umgebungen zum Einsatz kommt.

Der Routing Cache ist keine neue Erfindung. In den späten 1980er und 1990er Jahren hatten die meisten Router eine Route Caching Fähigkeit eingebaut. Allerdings waren diese Designs nicht in der Lage mit den schnell ansteigenden Packet Forwarding Raten Schritt zu halten. Wegen der hohen Kosten für *Cache Misses,* aufgrund derer es niedrigeren Durchsatz von Paketen gab, waren ein hoher Verlust von Paketen das Resultat [10].

Route Caching kann aus den folgenden Gründen (abgesehen von Effizienz) notwendig sein. Neue Protokolle mit größeren (z.B. IPv6) oder flacheren Adressräumen wurden vorgeschlagen, um das Wachstum und die Konfigurierung des Internets zu erleichtern. Jedoch, würden diese Protokolle eingesetzt werden, stiege die Größe der FIB so signifikant an, dass die Kapazitäten der meisten momentan verwendeten Ausrüstung überschritten wären. Und es ist vorhergesagt, dass mehrere Millionen FIB Einträge innerhalb einiger Jahre benötigt werden, falls der momentane Wachstumstrend weitergehen sollte [10]. Wenn FIBs sich füllen, stürzen herkömmliche Router ab oder beginnen sich inkorrekt zu verhalten [4]. Dies kann Operatoren dazu zwingen neue ‚Line Cards‘[3] oder gar Router mit größerem Speicher einzusetzen [10].

Die Größe der globalen Routing Information Base (RIB) wächst in einem alarmierenden Tempo an. Dies führt direkt zu einem rapiden Wachstum der globalen FIB Größe, welche ernste Probleme für ISP[4] aufkommen lässt, da FIB Speicher in Line Cards viel teurer sind, als reguläre Speichermodule und es sehr kostspielig für ISP ist, diese Speicherkapazität regelmäßig für alle Router zu erhöhen. Eine potentielle Lösung ist es, die beliebtesten FIB Einträge in schnellem Speicher, also einen FIB Cache, zu installieren [11].

3.3 Die Entfernung des Routing Caches aus dem Linux Kernel

In der Linux Kernel Version 3.7 ist der Routing Cache nicht mehr enthalten. David S. Miller, einer der Hauptverantwortlichen für die Entwicklungen im Netzwerkbereich des Linux Kernels [2], begründet die Entfernung des Routing Caches in seinem Commit des Updates [13]. Der Routing Cache ist nicht-deterministisch, bezogen auf Effizienz, und ist Gegenstand von einigermaßen einfach zu startenden *Denial of Service* Attacken. Seiner Meinung nach, arbeite der Routing Cache großartig für ‚braven‘ Verkehr und die Welt sei ein viel freundlicherer Ort gewesen, als die Gründe, die zum Design des Routing Caches führten, diskutiert wurden. Selbst für ‚braven‘ legitimen Verkehr sehen hochfrequentiert besuchte Seiten in den Routing Caches Trefferraten von nur ~10% [13].

Denial of Service (DoS) Attacken oder auch *Distributed Denial of Service* Attacken (DDoS) sind Angriffe, die einen Service im Internet (z.B. eine Internetseite, ein Router oder ein Server) auf Dauer oder zeitweilig unerreichbar machen sollen (vgl. [16]). Dies kann auf unterschiedliche Weise erreicht werden. Beispielsweise können *Overflows* in Tabellen oder Speichern provoziert werden oder die Ressourcen des Zielgerätes werden ausgereizt (z.B. CPU Auslastung maximieren oder RAM überladen). Ein möglicher solcher Angriff, der den Routing Cache als Ziel hat, kann unterschiedliche Absichten verfolgen. Eine davon könnte sein, dass möglichst viele Einträge im Routing Cache auf einen oder einige wenige Buckets verteilt werden. Dies führt dazu, dass das Durchsuchen des Routing Caches nach einem Element dem Durchsuchen einer linearen Liste näher kommt, als der Suche nach einem Element in einer Hashtabelle. Damit würde sich die Suchzeit für Einträge stark erhöhen, und somit der Durchsatz von Paketen insgesamt deutlich verringern. Eine Voraussetzung für diesen Angriff wäre, dass der Angreifer weiß, welche Hashfunktion gerade genutzt wird. Die Ausführung dieses Angriffes wurde bereits im Jahre 2003 erschwert, indem ein Patch die Hashfunktion verschlüsselte, nicht-linear machte und zudem die Hashfunktion ca. alle 10 Minuten variierte (vgl. [21]). Eine weitere Möglichkeit, den Route Cache ineffektiv zu machen, wäre es dafür zu sorgen, dass maximal viele C*ache Misses* beim Durchsuchen des Routing Caches auftreten. Dies würde dazu führen, dass sowohl der Route Cache als auch die FIB immer wieder abgefragt werden müssen. Beschäftigt man das System damit ausreichend, kann dafür gesorgt werden, dass normale (Nicht-Angreifer) Pakete nicht mehr oder nur teilweise verarbeitet bzw. weitergeleitet werden. Obwohl eine bestimmte Menge an Paketen zur Verarbeitung in Zwischenspeichern kurzweilig eingereiht werden können, hat dies kaum eine Auswirkung auf Angriffe dieser Art. Eine Möglichkeit, diesen Angriff zu entkräften wäre es, Pakete die keinen Treffer im Routing Cache haben, an einen anderen Router zu senden, der Routing Entscheidungen nur aufgrund der FIB trifft. Ein weiterer Schutzmechanismus vor solchen Angriffen ist *Ingress-Filtering*, worauf in diesem Paper kein Bezug genommen wird.

Ein weiteres und effektives Ziel des Angreifers könnte es sein, den Routing Cache so stark aufzublähen, dass der GC die Größe des Routing Caches nicht schnell genug reduzieren kann. Dies sollte möglich sein, indem in sehr kurzer Zeit extrem viele Pakete verarbeitet werden sollen. Es könnten nun mehrere Szenarios

[3] ‚Line Card‘ http://www.cscprinters.com/line-card.html

[4] ‚ISP‘ – Internet Service Provider

auftreten. Bei jedem *Cache Miss* wird eine Route aus der FIB in den Routing Cache geladen und dabei der GC aufgerufen. Da aber *gc_interval_ms* die Häufigkeit des Aufrufens des GC beschränkt, sollte es möglich sein, eine Überladung der Routing Tabelle zu erreichen. Alternativ könnten aber auch durch das häufige aufrufen des GCs Pakete verloren gehen, da während der GC läuft, keine Zugriffe auf die Tabelle zugelassen sind (Semaphorenproblem). Eine weitere Alternative, die zu einer Überladung der Tabelle führen könnte, ist, dass durch den GC nicht genügend Tabelleneinträge entfernt werden, da *gc_timeout* bestimmt, welche Tabelleneinträge vom GC entfernt werden.

Der Hauptgrund für die Entfernung des IPv4 Routing Caches war, dass DoS-Attacken gegen ihn einfach waren, da der IPv4 Routing Cache für jeden einzigartigen Flow einen Cache Eintrag generiert. Im Wesentlichen bedeutet das, dass es durch das Senden von Paketen an zufällige Adressen möglich ist, eine unbeschränkte Menge an Einträgen im Routing Cache zu generieren [18].

Obwohl der Routing Cache im Jahre 2012 entfernt wurde, wird erwogen, diesen wieder optional im Linux Kernel zur Verfügung zu stellen [12].

4. OPEN VSWITCH & ROUTING CACHES

Open vSwitch (OvS) ist eine Open Source Software, die einen bzw. mehrere virtuelle Switches implementiert. Sie bieten Netzwerkzugang für Virtual Machines (VMs), indem virtuelle und physische Netzwerk Schnittstellen verbunden werden [5]. Vor Open vSwitch bietet auch der Linux Kernel eine schnelle und verlässliche Bridge (die Linux Bridge) an (vgl. [6]). Die Zielgruppe von Open vSwitch ist allerdings an Multi-Server Deployments ausgerichtet, eine Umgebung, für welche die vorherigen Programme nicht gut ausgelegt waren [6]. Somit findet sich Open vSwitch beispielsweise oft in Cloud Netzwerken wieder.

Wie auch einst der Linux Kernel implementiert OvS einen Routing Cache, wenn es um Packet Forwarding geht. Wie auch bei dem Linux Kernel sind DoS-Attacken auf den Routing Cache möglich. Open vSwitch Entwickler sind sich dessen bewusst und versuchen deshalb Schutzmechanismen einzuführen, welche DoS-Attacken zumindest erschweren sollen. Beispielsweise wurde der GC für die Verwaltung des Caches abgeschafft. In Verbindung damit wurde der *Flow Mask Cache* eingeführt. Um ein beliebiges Paket zu verarbeiten muss OvS eine Flow Mask für den Flow erzeugen und diese dann im Flow Cache abfragen [20].

Wie der Linux Kernel legt auch OvS einen Trie an, um den Longest-Prefix Match möglichst schnell zu bestimmen. Um diesen Prozess weiter zu beschleunigen werden statt Microflows Megaflows bevorzugt. Also könnte man von einem Megaflow Cache sprechen. Anhand des folgenden Beispiels lässt sich diese Vorgehensweise veranschaulichen. Zum Vereinfachten Verständnis werden IPv4 Adressen, welche nur nach oktalen Präfixen unterschieden werden (z.B. /8, /16, /24 und /32), verwendet. Angenommen OvS möchte einen Megaflow für ein Packet generieren, welches an die Adresse 1.2.3.4 gesendet werden soll. Besteht nun nur ein Match für beispielsweise das erste Oktett, also 1/8, wird der Trie nur um das Präfix 1.2/16 erweitert, statt um das Präfix 1.2.3.4/32. Dieses Vorgehen verkürzt den Longest Prefix Match für folgende Pakete [15].

Um bei großem Paketdurchsatz effizient zu bleiben, verfügt der Flow Cache maximal über 200.000 Einträge. Um außerdem zu verhindern, dass mehr Flows hinzugefügt werden, als gelöscht werden, wurden zusätzliche Threads eingeführt, die Routen während der Verarbeitung von Paketen revalidieren. Dies geschieht indem Cache-Statistiken genutzt werden, welche ungefähr einmal pro Sekunde vom Kernel abgefragt werden. Der Kernel gibt dann einen Zähler (eine Variable, die zählt, wie oft ein Flow seit der letzten Abfrage genutzt wurde) für die jeweiligen Flow Einträge zurück und entscheidet basierend auf dem Zähler, ob der Flow beibehalten oder gelöscht wird. Der Cache passt sich im Betrieb so an, dass er für die ein Revalidierungsintervall weniger als eine Sekunde benötigt. Diese Vorgehensweise soll den Revalidierungsprozess auch bei großem Paketdurchsatz effizient halten [15].

Der Hauptgrund, einen Routing Cache zu verwalten, sei laut Miller hauptsächlich auf Effizienzgründe zurückzuführen (vgl. [20]). Eine Effizienz Analyse von Virtual vSwitch von Emmerich et al. [5] veranschaulicht diesen Umstand. In einer anderen Arbeit von Rotsos et. al [19] wird die Implementation von OpenFlow[5] in verschiedenen Routern bzw. Switches untersucht, darunter auch Open vSwitch, und deren Performanz verglichen.

Um den Routing Cache zu umgehen und somit auch Angriffe, empfiehlt Miller die darunterliegende Datenstrukturzugriffe schneller zu machen, um somit Abfragen effizient zu gestalten (vgl. [20]). Im Linux Kernel ist dies vermutlich durch die Implementierung des *LC-Trie* realisiert worden, welcher Longest Prefix Abfragen effizienter gestalten kann.

5. ZUSAMMENFASSUNG

In dieser Arbeit wurde die grundlegende Struktur der Paketverarbeitung im Linux Kernel erläutert. Besonderes Augenmerk liegt auf dem vom Linux Kernel verwalteten Routing Cache, welcher in neueren Versionen nicht mehr enthalten ist. Allgemein lässt sich sagen, dass der Routing Cache in erster Linie als Effizienz Werkzeug angesehen werden kann, aber auch als eine Möglichkeit Kosten bei Internet Routern einzusparen, da diese dann nicht regelmäßig mit teurem Hochleistungsspeichern wieder aufgerüstet werden müssen. Allerdings birgt ein Routing Cache nicht nur Vorteile, denn es kann sich als schwer erweisen, diesen passend für seine Netzwerkumgebung zu konfigurieren. Außerdem ist er, trotz aller Anstrengungen von Entwicklern, ein großes Sicherheitsrisiko. Es können relativ leicht ausführbare Paketmassen bzw. -abfolgen den Routing Cache ineffizient machen oder überlasten. Trotz der bekannten Risiken bezüglich der Anfälligkeit gegenüber DoS-Attacken scheinen Open vSwitch Entwickler in näherer Zukunft den Routing Cache nicht entfernen zu wollen. Andererseits haben auch Linux Entwickler bereits erwogen, den Routing Cache, diesmal zwar optional, erneut einzuführen.

6. ACKNOWLEDGEMENTS

Diese Arbeit wurde im Rahmen eines Seminars an der Technischen Universität München am Lehrstuhl für Rechnernetze und unter Betreuung von Daniel Raumer und Paul Emmerich angefertigt.

[5] http://openflow.org

7. REFERENCES

[1] Linux kernel readme. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/README?id=f3b8436ad9a8ad36b3c9fa1fe030c7f38e5d3d0b. Accessed: 2015-03-29.

[2] List of Maintainers. http://lxr.linux.no/L3703. Accessed: 2015-03-31.

[3] Vincent Bernat. Tuning Linux IPv4 route cache. http://vincent.bernat.im/en/blog/2011-ipv4-route-cache-linux.html, 2012. Accessed: 2015-03-30.

[4] Di-Fa Chang, Ramesh Govindan, and John Heidemann. An empirical study of router response to large bgp routing table load. In *Proceedings of the 2Nd ACM SIGCOMM Workshop on Internet Measurment*, IMW '02, pages 203–208, New York, NY, USA, 2002. ACM.

[5] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Performance characteristics of virtual switching. In *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, pages 120–125, Oct 2014.

[6] Thomas Graf. Why Open vSwitch? https://github.com/openvswitch/ovs/blob/master/WHY-OVS.md, October 2014. Accessed: 2015-04-01.

[7] Glenn Herrin. Linux IP Networking - A Guide to the Implementation and Modification of the Linux Protocol Stack: Chapter 7. http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html#tth_chAp7, May 31 2000. Accessed: 2015-03-29.

[8] Glenn Herrin. Linux IP Networking - A Guide to the Implementation and Modification of the Linux Protocol Stack: Chapter 8. http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html#tth_chAp8, May 31 2000. Accessed: 2015-03-30.

[9] Alessandro Rubini Jonathan Corbet, Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3 edition, February 2005. Ch 10.4 Interrupt Handling.

[10] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting route caching: The world should be flat. In SueB. Moon, Renata Teixeira, and Steve Uhlig, editors, *Passive and Active Network Measurement*, volume 5448 of *Lecture Notes in Computer Science*, pages 3–12. Springer Berlin Heidelberg, 2009.

[11] Yaoqing Liu, Syed Obaid Amin, and Lan Wang. Efficient fib caching using minimal non-overlapping prefixes. *SIGCOMM Comput. Commun. Rev.*, 43(1):14–21, January 2013.

[12] David S. Miller. Things that need to get done in the linux kernel networking. http://vger.kernel.org/ davem/net_todo.html. Accessed: 2015-03-31.

[13] David S. Miller. ipv4: Delete routing cache. http://git.kernel.org/cgit/linux/kernel/git/davem/net-next.git/commit/?id=89aef8921bfbac22f00e04f8450f6e447db13e42, July 2012. Accessed: 2015-03-31.

[14] S. Nilsson and G. Karlsson. Ip-address lookup using lc-tries. *Selected Areas in Communications, IEEE Journal on*, 17(6):1083–1092, Jun 1999.

[15] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[16] Radware. *DDoS Surivial Handbook*. Radware, 2013.

[17] Éric Leblond. David miller: routing cache is dead, now what? https://home.regit.org/2013/03/david-miller-routing-cache-is-dead-now-what/. Accessed: 2015-03-29.

[18] Rami Rosen. *Linux Kernel Networking - Implementation and Theory*. Apress, 2014. p. 134.

[19] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. Oflops: An open framework for openflow switch evaluation. In *Proceedings of the 13th International Conference on Passive and Active Measurement*, PAM'12, pages 85–95, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] Pravin B Shelar. [Git net-next] Open vSwitch. http://comments.gmane.org/gmane.linux.network/325250, August 2014. Accessed: 2015-04-01.

[21] Florian Weimer. Algorithmic Complexity Attacks and the Linux Networking Code. http://www.enyo.de/fw/security/notes/linux-dst-cache-dos.html, July 2003. Accessed: 2015-03-30.