Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich

Proceedings of the Seminars Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)

Winter Semester 2016/2017

Munich, Germany

Editors

Georg Carle, Daniel Raumer, Lukas Schwaighofer

Publisher

Chair of Network Architectures and Services

Lehrstuhl für Netzarchitekturen und Netzdienste
Fakultät für Informatik
Technische Universität München

Proceedings zu den Seminaren Future Internet (FI) und Innovative Internet-Technologien und Mobilkommunikation (IITM)

Wintersemester 2016/2017

München, 17. 07. 2016 – 28. 02. 2017

Editoren: Georg Carle, Daniel Raumer, Lukas Schwaighofer

Network Architectures
and Services
NET 2017-05-1

Proceedings of the Seminars
Future Internet (FI) and Innovative Internet Technologies and Mobile Communication (IITM)
Winter Semester 2016/2017

Editors:

Georg Carle
Lehrstuhl für Netzarchitekturen und Netzdienste (I8)
Technische Universität München
85748 Garching b. München, Germany
E-mail: carle@net.in.tum.de
Internet: `https://net.in.tum.de/~carle/`

Daniel Raumer
Lehrstuhl für Netzarchitekturen und Netzdienste (I8)
E-mail: raumer@net.in.tum.de
Internet: `https://net.in.tum.de/~raumer/`

Lukas Schwaighofer
Lehrstuhl für Netzarchitekturen und Netzdienste (I8)
E-mail: schwaighofer@net.in.tum.de
Internet: `https://net.in.tum.de/~schwaighofer/`

Cataloging-in-Publication Data

Seminars FI & IITM WS 16/17
Proceedings zu den Seminaren „Future Internet“ (FI) und „Innovative Internet-Technologien und Mobilkommunikation“ (IITM)
München, Germany, 17. 07. 2016 – 28. 02. 2017
ISBN: 978-3-937201-55-9

ISSN: 1868-2634 (print)
ISSN: 1868-2642 (electronic)
DOI: 10.2313/NET-2017-05-1
Lehrstuhl für Netzarchitekturen und Netzdienste (I8) NET 2017-05-1
Series Editor: Georg Carle, Technische Universität München, Germany

Vorwort

Vor Ihnen liegen die Proceedings der Seminare „Future Internet“ (FI) und „Innovative Internet-Technologien und Mobilkommunikation“ (IITM). Wir sind stolz, Ihnen Ausarbeitungen zu aktuellen Themen, die im Rahmen unserer Seminare im Wintersemester 2016/2017 an der Fakultät für Informatik der Technischen Universität München verfasst wurden, präsentieren zu dürfen. Den Teilnehmerinnen und Teilnehmern stand es wie in der Vergangenheit frei, das Paper und den Vortrag in englischer oder in deutscher Sprache zu verfassen. Dementsprechend finden sich sowohl englische als auch deutsche Paper in diesen Proceedings.

Unter allen Themen, die sich mit Aspekten der Computernetze von morgen befassen, verliehen wir in jedem der beiden Seminare einen Best Paper Award. Im IITM-Seminar ging dieser an Herrn Christoph Rudolf, der in seiner Ausarbeitung „SQL, noSQL or newSQL – Vergleich und Anwendung in Smart Spaces“ die Eignung verschiedener Datenbanken für die Anwendung in Smart Spaces untersucht. Im FI-Seminar wurde dieser Herrn Dominik Schöffmann verliehen für seine Ausarbeitung „Vergleich verschiedener Datenstrukturen für Routingtabellen“, in welcher er verschiedene Datenstrukturen für Routingtabellen in softwarebasierten Routern vergleicht.

Einige der Vorträge wurden aufgezeichnet und sind auf unserem Medienportal unter `https://media.net.in.tum.de` abrufbar.

Im FI-Seminar wurden Beiträge zu den folgenden Themen verfasst:

- Anonymität & Bitcoins
- Visualisierung sich ändernder Wahrscheinlichkeitsverteilungen
- Vergleich verschiedener Datenstrukturen für Routingtabellen

Auf `https://media.net.in.tum.de/#%23Future%20Internet%23WS16` können die aufgezeichneten Vorträge zu diesem Seminar abgerufen werden.

Im IITM-Seminar wurden die folgenden Themen abgedeckt:

- OpenOnload: Ein Framework zur schnellen Paketverarbeitung
- Source Packet Routing in Networking (SPRING)
- SQL, noSQL or newSQL – Vergleich und Anwendung in Smart Spaces
- P4 Compiler & Interpreter: Ein Überblick
- Verifizierbare Secret Sharing Mechanismen – Ein Überblick

Auf `https://media.net.in.tum.de/#%23IITM%23WS16` können die aufgezeichneten Vorträge zu diesem Seminar abgerufen werden.

Wir hoffen, dass Sie den Beiträgen dieser Seminare wertvolle Anregungen entnehmen können. Falls Sie weiteres Interesse an unseren Arbeiten habe, so finden Sie weitere Informationen auf unserer Homepage `https://net.in.tum.de`.

München, Mai 2017

Georg Carle

Daniel Raumer

Lukas Schwaighofer

Preface

We are pleased to present you the proceedings of our seminars on "Future Internet" (FI) and "Innovative Internet Technologies and Mobile Communication" (IITM) which took place in the winter semester 2016/2017. In both seminar courses, the authors were free to write their paper and give their talk in English or German.

We honored the best paper of each seminar with an award. This semester the award in the IITM seminar was given to Mr Christoph Rudolf who analysed different database technologies in the domain of smart spaces in his paper "SQL, noSQL or newSQL – comparison and applicability for Smart Spaces". In the FI seminar the award was given to Mr Dominik Schöffmann for his paper "Comparison of Efficient Routing Table Data Structures" wherein he compared different data structures for routing tables in software-based routers.

Some of the talks were recorded and published on our media portal `https://media.net.in.tum.de`.

In the seminar FI we dealt with issues and innovations in network research. The following topics were covered:

- Towards a More Anonymous Bitcoin
- Visualizing Changing Probability Distributions
- Comparison of Efficient Routing Table Data Structures

Recordings can be accessed on `https://media.net.in.tum.de/#%23Future%20Internet%23WS16`.

In the seminar IITM we dealt with different topics in the area of network technologies, including mobile communication networks. The following topics were covered:

- Comparing OpenOnload: A High-Speed Packet IO Framework
- Source Packet Routing in Networking (SPRING)
- SQL, noSQL or newSQL – comparison and applicability for Smart Spaces
- P4 Compiler & Interpreter: A Survey
- Verifiable Secret Sharing Mechanisms - A Survey

Recordings can be accessed on `https://media.net.in.tum.de/#%23IITM%23WS16`.

We hope that you appreciate the contributions of these seminars. If you are interested in further information about our work, please visit our homepage `https://net.in.tum.de`.

Munich, May 2017

Seminarveranstalter

Lehrstuhlinhaber

Georg Carle, Technische Universität München, Germany

Seminarleitung

Daniel Raumer, Technische Universität München, Germany
Lukas Schwaighofer, Technische Universität München, Germany

Betreuer

Edwin Cordeiro (cordeiro@net.in.tum.de)
Technische Universität München, Mitarbeiter I8

Paul Emmerich (emmericp@net.in.tum.de)
Technische Universität München, Mitarbeiter I8

Sebastian Gallenmüller (gallenmu@net.in.tum.de)
Technische Universität München, Mitarbeiter I8

Stefan Liebald (liebald@net.in.tum.de)
Technische Universität München, Mitarbeiter I8

Heiko Niedermayer (niedermayer@net.in.tum.de)
Technische Universität München, Mitarbeiter I8

Marc-Oliver Pahl (pahl@net.in.tum.de)
Technische Universität München, Mitarbeiter I8

Daniel Raumer (raumer@net.in.tum.de)
Technische Universität München, Mitarbeiter I8

Dominik Scholz (scholz@net.in.tum.de)
Technische Universität München, Mitarbeiter I8

Matthias Wachs (wachs@net.in.tum.de)
Technische Universität München, Mitarbeiter I8

Seminarhomepage

`https://net.in.tum.de/teaching/ws1617/seminars/`

Inhaltsverzeichnis

Seminar Future Internet

Seminar Innovative Internet-Technologien und Mobilkommunikation

Towards a More Anonymous Bitcoin

Thea Heim
Betreuer: Heiko Niedermayer
Seminar Future Internet SS2016
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: heimt@in.tum.de

KURZFASSUNG

Ein Bitcoin Konto funktioniert zwar ohne Angabe persönlicher Daten wie Name, Adresse oder Telefonnummer kann trotzdem aber nicht als anonym bezeichnet werden. Alleine die Blockchain, die alle Transaktionen vom Tag eins bis heute speichert, und von jeder beliebigen Person eingesehen werden kann, bietet eine Menge Möglichkeiten für einen erfolgreichen Deanonymisierungs-Angriff. Darüber hinaus kann über das Peer-to-peer Netzwerk, welches keiner zentralen Kontrolle unterliegt, mit vergleichsweise geringem Aufwand auf die Identitäten der Nutzer geschlossen werden. Eine Lösung für mehr Anonymität im Bitcoin System ist der CoinJoin Mechanismus. Hier schließen sich für eine Transaktion mehrere Teilnehmer des Netzwerkes zusammen und erschweren somit die Möglichkeit durch die Analyse der in der Blockchain festgehaltenen Transaktionen (Clustering) auf Identitäten zu schließen erheblich. JoinMarket ist eine Implementierung für diesen Mechanismus, die zudem noch eine Lösung für das Grundproblem der CoinJoin Idee - nämlich, dass sich die Gruppe der Teilnehmer, die für eine gemeinsame Transaktion in Frage kommt, auch erst einmal finden muss - liefert. Dennoch kann auch dieses System in der derzeitigen Implementierung noch nicht vollkommen die Anonymität seiner Nutzer schützen, weil es immer noch Angriffsmöglichkeiten wie beispielsweise die Sybil-Attacke bietet.

Schlüsselworte

Bitcoin, Blockchain, CoinJoin, JoinMarket

1. EINLEITUNG

In der heutigen Zeit der Digitalisierung verliert Bargeld als Zahlungsmethode zunehmend an Bedeutung. Elektronische Bezahlmethoden wie z.B. PayPal, Geldkarte, Kreditkarte oder Online-Banking werden immer häufiger eingesetzt und nehmen bereits einen großen Anteil im Handel ein [12].

Neben den genannten Online-Bezahlsystemen erfreuen sich auch digitale bzw. virtuelle Währungen immer größerer Beliebtheit. Der Unterschied zu den herkömmlichen elektronischen Währungen besteht darin, dass nicht die Ursprungswährung (z.B. Euro) erhalten bleibt, sondern in eine künstliche Währung (z.B. Bitcoin) gewechselt wird [17]. Die bekannteste, weltweit verfügbare digitale Währung ist Bitcoin. Sie hat sich seit seiner Einführung (2009) durch den bis heute unbekannten Erfinder Satoshi Nakamoto zunehmend verbreitet [19]. Im Jahr 2013 ist die Währung Bitcoin auch durch die Bundesregierung Deutschland als „privates Geld“ anerkannt worden [15]. Sorge und Krohn-Grimberge [20] beziffern die Anzahl an Bitcoin-Nutzern im Jahr 2013 auf 3,4 Millionen und die Anzahl an Bitcoin-Konten auf ca. 12 Millionen (ein Nutzer kann beliebig viele Konten eröffnen).

Während die Währung in Teilen der USA bereits im Arbeitsalltag angekommen ist (Angestellte im Bundestaat Kentucky können sich bereits ihr Gehalt in Bitcoin ausbezahlen lassen [4]), ist die Anzahl von Internet-Plattformen, auf denen mit Bitcoin bezahlt werden kann, noch stark eingeschränkt. Die Währung wird vor allem auf kleineren Internet-Marktplätzen genutzt und teilweise durch größere Organisationen wie z.B. Wikileaks oder Wordpress akzeptiert. Die Verwendung bei gewöhnlichen Zahlvorgängen wie beispielsweise in Supermärkten ist momentan noch nicht absehbar [12].

Der Anreiz mit digitalen Währungen wie Bitcoin zu bezahlen besteht nach Peyrl ([17]) für den Nutzer v.a. darin, dass für die Verwendung nichts weiter als ein Rechner mit Internetzugang benötigt wird, auf dem die entsprechende Bitcoin-Software installiert ist. Die Zahlungsvorgänge werden abhängig von der Größe der Transaktion innerhalb kürzester Zeit abgeschlossen und es fallen keine bzw. sehr niedrige Transaktionsgebühren an. Zudem genießt der Nutzer eine gewisse Unabhängigkeit, da virtuelle Geldsysteme keiner staatlichen Kontrolle unterliegen. Die Tatsache, dass Bitcoin ohne Angabe personenbezogener Daten verwendet werden kann, macht Bitcoin besonders interessant für Personen, die bei ihren Transaktionen eine gewisse Anonymität wahren möchten.

Diese Eigenschaft der Unabhängigkeit bzw. Anonymität wird nach einer kurzen allgemeinen Einführung in das Bitcoin Konzept (Teil 2) in dieser Arbeit näher betrachtet. Dabei wird die in der Literatur häufig diskutierte Frage aufgeworfen, wie anonym Bitcoin wirklich ist (Teil drei). Hier wird v.a. die Funktion der Blockchain, die alle getätigten Transaktionen speichert, kritisch betrachtet. Zudem werden die Schwächen des dezentralen Peer-to-Peer Netzwerkes, auf dem Bitcoin basiert, dargestellt und ein Überblick über erfolgreiche Hackerangriffe in der Vergangenheit gegeben. Im vierten Teil dieses Artikels wird die Bitcoin-Platform JoinMarket vorgestellt, die ein Ansatz für Transaktionen mit mehr Anonymität ist. Abschließend wird ein kurzes Fazit über das Bitcoin-System in Bezug auf dessen Anonymität gezogen.

2. EINFÜHRUNG IN BITCOIN

In diesem Kapitel wird eine Einführung in die Funktionsweise von Bitcoin gegeben. Dazu wird im ersten Teil zunächst erläu-

doi: 10.2313/NET-2017-05-1_01

tert, wie das System der kryptographischen Hashfunktionen in der Bitcoin Software eingesetzt wird. Im zweiten Teil wird darauf eingegangen, welche Möglichkeiten ein Nutzer hat Bitcoins zu erwerben und welches Angebot gegenwärtig für den Handel mit Bitcoins besteht.

2.1 Funktionsweise

Wie in der Einleitung bereits beschrieben, ist die einzige Voraussetzung für die Verwendung von Bitcoin ein Rechner mit Verbindung zum Internet, auf dem die Bitcoin-Software installiert ist. Jeder, der mit seinem Rechner die Bitcoin-Software nutzt, ist automatisch Teil des Bitcoin-Netzwerkes. Das Netzwerk realisiert Überweisungen zwischen den Konten der Nutzer, wobei jeder Nutzer ein oder mehrere Konten besitzen kann. Jedes Konto wird über eine Bitcoin-Adresse identifiziert und jeder Bitcoin-Adresse ist ein bestimmtes Bitcoin-Guthaben zugeordnet [20].

Das Bitcoin-Netzwerk ist ein Peer-to-Peer-Netzwerk (siehe Abbildung 1), was bedeutet, dass es - anders als im herkömmlichen Zahlungsverkehr über Banken - keine zentrale Instanz oder Server gibt, worüber die Kommunikation unter den Teilnehmern abgewickelt wird. Demnach sind alle Teilnehmer gleichberechtigt miteinander verbunden und bezüglich neuer Informationen zu jedem Zeitpunkt auf dem gleichen Wissensstand [18].

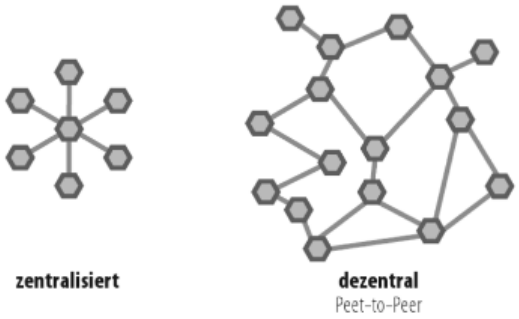


Abbildung 1: Vergleich Peer-to-Peer Netzwerk (dezentral) mit zentralem Netzwerk [18]

Gerade aufgrund des Fehlens einer vertrauenswürdigen Instanz muss man sich die Frage stellen, wie die Integrität der Bitcoins erreicht werden kann. Dazu verwendet das System sogenannte (kryptographische) Hashfunktionen, d.h. jede Bitcoin-Adresse setzt sich aus einem öffentlichen Schlüssel (jedem Nutzer bekannt) und einem mathematisch korrespondierenden privaten Schlüssel (nur dem Besitzer bekannt) zusammen. Kann ein Nutzer nachweisen, dass er den zu einer öffentlichen Adresse zugehörigen privaten Schlüssel kennt, darf er über den entsprechenden Bitcoin verfügen [18].

Um Betrug zu verhindern (z.B. mehrfaches Überweisen eines Beitrages von einem Konto) werden in Bitcoin alle durchgeführten Transaktionen in der sogenannten „Blockchain“ veröffentlicht. Jede neue Transaktion gilt zunächst als unbestätigt und wird erst nach Überprüfung durch einen Bitcion Teilnehmer, dem sogenannten „Miner“, in die Blockchain eingetragen. Als Miner kann jeder Teilnehmer des Netzwerkes fungieren. Sein Ziel ist es einen sogenannten „One-Way-Hash“zu lösen. Dies schafft ein Nutzer nur mit extrem großer Rechenleistung, wobei sein Ergebnis sehr leicht rückwärts gerechnet und somit überprüft werden kann [20]. Nach Becker et al. [2] sichert also die ständige Erbringung hoher Rechenleistung die Vertrauenswürdigkeit der Bitcoins.

2.2 Erwerb & Handel

Eine Möglichkeit zum Erwerb von Bitcoins ist das im vorherigen Abschnitt beschriebene freiwillige Bitcoin-Mining. Für einen erfolgreich erbrachten Arbeitsbeweis werden einem Miner die mit den Transaktionen verbundenen Transaktionsgebühren und eine Belohnung für die Erbringung der Rechenleistung gutgeschrieben [19]. Nach Vogel [21] nehmen die Miner (also Teilnehmer, die durch Mining Bitcoins erwerben) nur einen geringen Teil des Bitcoin-Netzwerkes ein, da hierfür ein hohes technisches Verständnis sowie ein stetig steigender Rechenaufwand benötigt wird. Eine weitere Variante des Bitcoin-Mining ist das Cloud-Mining. Hierbei erwerben die Miner ihre Bitcoins „durch das Anmieten von Rechenleistung [21]“.

Neben dem Mining existieren noch sogenannte Faucets und Give-Away-Applikationen (z.B. `http://moonbit.co.in/`). Diese verteilen Gratis-Bitcoins für Leistungen wie z.B. das Anklicken von Werbeanzeigen. Zudem können geringe Mengen an Bitcoins über Bitcoin-Onlinespiele (z.B. `https://freebitco.in`) erworben werden [21].

Am häufigsten werden Bitcoins aber über Tauschplattformen (z.B. `coinbase.com`) erworben. Hier erhalten die Nutzer Bitcoins im Tausch gegen eine konventionelle Währung wie Euro oder Dollar [21].

Die Bitcoins der Teilnehmer werden mit Hilfe der Bitcoin-Software in sogenannten „Wallets“ gespeichert. Wallets sind digitale Brieftaschen, die die Schlüsselpaare eines oder mehrerer Nutzer aufbewahren und wesentliche Funktionen wie die Durchführung von Überweisungen, das Erzeugen neuer Schlüsselpaare sowie die Verwaltung von Adressen gewährleisten [19].

Gegenwärtig steigt die Anzahl von Anbietern, die Bitcoin als Währung anerkennen. Der Großteil des Angebots ist sehr technisch orientiert, wobei „Internet- und mobile Dienstleistungen wie VPN, (Web-)Hosting und Programmierung [10]“ überwiegen. Laut Bitcoin Wiki [6] reicht das Angebot außerdem von materiellen Gütern verschiedenster Art (Musik, Bücher, Kleidung, Lebensmittel, etc.) über den Reise- bzw. Tourismusbereich (Reiseportale, Hotels, Flüge, etc.) bis hin zu ersten Restaurants, die eine Bezahlung in Bitcoin akzeptieren.

3. WIE ANONYM IST BITCOIN?

Für eine Transaktion mit den gängigen Bezahlmethoden (Kreditkarte, Online-Banking, etc.) ist es erforderlich seine Kontodaten entweder direkt an die involvierte Person oder an einen unabhängigen Zwischenakteur (z.B. Paypal) weiterzugeben. Im Gegensatz dazu ermöglicht Bitcoin die Durchführung von Transaktionen ohne die Angabe von persönlichen Daten [21]. Aus diesem Grund wird dem Bitcoin-System gelegentlich Anonymität unterstellt.

Im Zuge einer Internet Infrastruktur, in der es möglich ist IP

doi: 10.2313/NET-2017-05-1_01

Adressen nachzuverfolgen, und der bereits angesprochenen Blockchain, die alle jemals durchgeführten Transaktionen speichert, muss man sich die Frage stellen, inwieweit man beim Bitcoin-System überhaupt von Anonymität sprechen kann. Dies soll nachfolgend diskutiert werden.

3.1 Blockchain

Vogel [21] bezeichnet die Blockchain als „Buchhaltungsdatenbank [...], die alle Bitcoin-Transaktionen seit 2009 speichert." Es handelt sich dabei um eine Kette aus Blöcken, in der jeder Block verschlüsselte Daten zu den durchgeführten Transaktionen enthält (siehe Abbildung 2). Ein neuer Block wird erst an den vorherigen Block angehängt wenn er den Verifizierungsvorgang durch die Miner bestanden hat (siehe 2.1). Tatsächlich wird ca. alle 10 Minuten ein neuer Block gebildet, wobei die Blockgröße derzeit auf 1 MB limitiert ist [5].

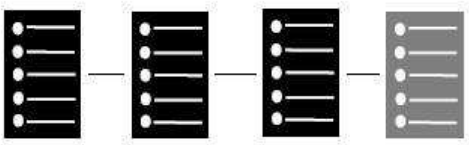


Abbildung 2: Blockchain [20]

Die Blockchain kann also von jedem Nutzer eingesehen werden und enthält Informationen über jede Transaktion, die eine Adresse jemals an eine andere Adresse getätigt hat. Die Anonymität ist dadurch aber auf den ersten Blick nicht gefährdet, weil eine Adresse nur aus einem zufälligen Zahlenwert besteht und jeder Teilnehmer beliebig viele Adressen besitzen kann [9].

Trotzdem birgt dieses System in Bezug auf die Anonymität auch Gefahren für die Nutzer. Ein Risiko besteht nach Hobson [9] darin, dass ein Nutzer seine Bitcoin-Adresse zusammen mit seiner Identität entweder versehentlich oder sogar absichtlich veröffentlicht (z.B. um eine Spende zu erhalten). Jede Transaktion, die von oder an diesen Nutzer getätigt wird, wäre dann für alle Teilnehmer einsehbar. Spendet dieser Nutzer beispielsweise an eine umstrittene Gruppe oder Person, deren Bitcoin Adresse ebenfalls irgendwann veröffentlicht wurde, kann jeder Teilnehmer eine Verbindung zwischen den beiden Bitcoin-Adressen bzw. Identitäten herstellen. Zudem ist es relativ wahrscheinlich, dass ein Nutzer Bitcoins an einen anderen, ihm persönlich bekannten Nutzer, transferiert. Kennt man also die Identität der einen Bitcoin-Adresse besteht für die andere Bitcoin-Adresse automatisch erhöhte Gefahr ebenfalls erkannt zu werden.

Herrera-Joancomarti [8] beschreibt in seinem Artikel „Research and Challenges on Bitcoin Anonymity" die Methode des Clustering, mit deren Hilfe man bei Analyse der Blockchain von anonymen Bitcoin-Adressen auf Identitäten schließen kann. Der Grundgedanke dahinter ist, dass eine Transaktion aus mehreren sogenannten „Input"-Adressen besteht. Input-Adressen sind Outputs vorheriger Transaktionen und müssen immer als Ganzes verschickt werden (d.h. es können keine Teilbeträge von einem Konto genommen werden und der gesendete Gesamtbetrag muss größer oder gleich dem Output-Betrag sein). Werden nun beispielsweise 1 BTC von der einen und 5 BTC von der anderen Adresse an eine weitere Adresse transferiert, kann man davon ausgehen, dass beide Adressen denselben Besitzer haben. Verwendet dieser Nutzer nun für eine andere Transaktion nochmals eine der beiden Adresse plus eine weitere Adresse, so kann man diesem Nutzer schon drei Adressen zuordnen. Auf diese Weise können Cluster über Adressen im Bitcoin-Netzwerk gebildet und einem Nutzer zugeordnet werden. Bezieht man dann noch Informationen mit ein, die man - wie im vorheringen Abschnitt beschrieben - extern gesammelt hat (z.B. veröffentlichte Bitcoin-Adressen auf Twitter) so kann sogar auf die Identität, die sich hinter diesen Clustern verbirgt, geschlossen werden.

Für Vogel [21] ist ein weiteres Risiko „die Stelle, an der ein Bitcoin-Verwender seine BTC in konventionelle Währungen umtauscht". Der Teilnehmer muss an diesem Punkt seine „persönlichen Daten wie Name und Kontonummer - sofern er nicht gegen Bargeld tauscht - an eine Handelsplattform weitergeben." Sobald er das tut ist ein Zusammenhang zwischen Bitcoin-Adresse und Identität hergestellt.

3.2 P2P-Netzwerk

Wie in 2.1 beschrieben ist die zugrundeliegende Technik für Transaktionen im Bitcoin-System ein Peer-to-Peer-Netzwerk. Zum einen ermöglicht dieses dezentrale Netzwerk einen Zahlungsverkehr fernab von staatlicher Kontrolle, zum anderen birgt es Risiken für die Nutzer bzgl. Anonymität. Im Jahr 2014 veröffentlichten drei Forscher der Universität Luxemburg [3] in ihrem Paper „Deanonymisation of clients in Bitcoin P2P network" eine Vorgehensweise wie man die IP-Adressen der Nutzer im Bitcoin-Netzwerk aufdecken kann. Die Forscher benötigten dazu die folgenden vier Schritte.

1. Mit Hilfe der Methode „GETADDR" wird nach allen bekannten peers im Netzwerk gesucht. Diese Liste muss ständig aktualisiert werden.
2. Im zweiten Schritt wird eine Liste von denjenigen Bitcoin Clients, deren Identität man aufdecken möchte, erstellt. Dazu verwendet der Angreifer entweder IP-Adressen-Bereiche von bekannten Internet Providern oder sammelt bereits im Bitcoin Netwzerk publizierte Adressen.
3. Anschließend wird damit begonnen die Clients, die sich mit dem Netzwerk verbinden, zu ihren Entry Nodes zuzuordnen. Dazu verwendet der Angreifer die in 4 beschriebene Methode. Laut den Forschern reichen schon drei Entry Nodes, um einen Nutzer eindeutig zu identifizieren. Bereits bei zwei Entry Nodes wird schon ein großer Prozentsatz an Nutzern identifiziert.
4. Der vierte Schritt läuft parallel zu den Schritten 1 bis 3. Hier versucht der Angreifer die Transaktionen, die im Netzwerk erscheinen, den Entry Nodes und anschließend den Nutzern zuzuordnen. Um das zu schaffen, wartet der Angreifer bei allen eingerichteten Verbindungen auf „INVENTORY" Nachrichten und sammelt für alle Transaktionen T die ersten q Adressen von Bitcoin Servern, die die INVENTORY Nachricht weitergeleitet haben. Nachfolgend vergleicht er die gesammelten Adressen mit der in 1 erstellten Liste und findet

doi: 10.2313/NET-2017-05-1_01

mögliche Paare aus Entry Nodes und Clients. Bei ihrem Versuch erreichten die Forscher eine Quote von 11%, mit der sie IP-Adressen erfolgreich den Nutzern zuordnen konnten.

Zusammenfassend bleibt festzuhalten, dass die Forscher mit vergleichsweise geringem Aufwand (ca. 1500 Dollar pro Monat) eine Lösung für die Deanonymisierung von IP-Adressen gefunden haben. Auch wenn die Erfolgsquote mit 11% noch relativ gering ausfällt, kann bei Bitcoin nicht von vollständiger Anonymität gesprochen werden. Gerade für Kriminelle könnte diese Studie ein Anreiz sein, um die Identität von Nutzern herauszufinden.

3.3 Erfolgreiche Hackerangriffe

Wenngleich der Verzicht von Kontrolle beim dezentralen Bitcoin System eine Neuerung bedeutet und für viele Nutzer reizvoll ist, ermöglicht diese Tatsache auch ein unkontrollierbares Ausmaß an kriminellen Aktivitäten. In der Vergangenheit gab es trotz großer Sicherheitsvorkehrungen immer wieder erfolgreiche Hackerangriffe auf Bitcoin Wallets. Erst im August diesen Jahres sollen Hacker Bitcoins im Wert von 65 Millionen Dollar von der Plattform Bitfinex gestohlen haben [16]. Der bislang größte Hackerangriff betraf die japanische Bitcoin-Börse Mt.Gox, einst die größte Bitcoin-Börse der Welt. Bei diesem Angriff wurde fast eine halbe Milliarde Dollar gestohlen, was schließlich dazu führte, dass die Börse Konkurs anmelden musste [13]. Solche Hacker verschaffen sich Zugriff auf Bitcoin Wallets privater Nutzer und überschreiben die Guthaben an eine andere Adresse. Ist dies passiert, besteht für den Nutzer keine Möglichkeit mehr sich die Bitcoins wiederzubeschaffen. Besonders betroffen sind von diesen Angriffen tatsächlich die privaten Nutzer, weil sie meist keine besonderen Maßnahmen ergreifen, um ihre Anonymität zu bewahren. Personen, die Bitcoins für ihre kriminellen Machenschaften verwenden sind in dieser Hinsicht meist wesentlich versierter und treffen entsprechende Vorkehrungen, um ihre Identität zu schützen [7].

3.4 CoinJoin für mehr Anonymität

Eine Lösung für mehr Anonymität in Bitcoin Transaktionen bietet der sogenannte „CoinJoin Mechanismus“. Die Grundidee dahinter ist, dass sich mehrere voneinander unabhängige Sender und Empfänger von Bitcoins zusammenschließen und - anstelle von mehreren einzelnen Transaktionen - eine einzige Transaktion gemeinsam tätigen (siehe Abbildung 3)[14]. Dabei gibt es keine Begrenzung in der Anzahl der Nutzer, die sich für eine Transaktion zusammenschließen [1].

Der Vorteil darin ist, dass die Inputs und zugehörigen privaten Schlüssel einer Transaktion nicht nur einer einzelnen Person oder mehreren zusammengehörigen Nutzern bekannt sind und somit die in 3.1 beschriebene Deanonymisierungs-Methode des Clustering erheblich erschwert wird [11]. Dabei gilt, je höher die Anzahl der Teilnehmer an einer CoinJoin-Transaktion, desto schwieriger wird es für den Angreifer irgendetwas aus der Blockchain herauszulesen. Limitiert wird die CoinJoin Methode vor allem dadurch, dass für die Teilnahme an einer Transaktion andere Nutzer gefunden werden müssen, die zur selben Zeit eine Zahlung tätigen wollen. Dies ist nicht immer einfach und führt dazu, dass CoinJoin nur selten genutzt wird [1].

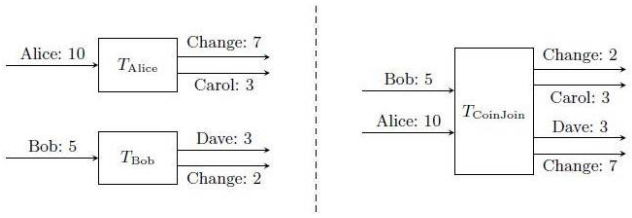


Abbildung 3: CoinJoin-Mechanismus: Zwei oder mehrere einzelne Transaktionen (links) werden in einer Transaktion kombiniert (rechts)[14]

Im nachfolgenden Kapitel wird die Plattform JoinMarket beschrieben, die eine Lösung für das grundsätzliche Problem von CoinJoin Transaktionen bietet.

4. JOINMARKET

JoinMarket ist eine Plattform für den Handel mit Bitcoins, welche auf dem Konzept des CoinJoin Mechanismus aufbaut und mit dem Ziel entwickelt wurde den Nutzern bei ihren Transaktionen mehr Anonymität zu bieten. Laut einer Analyse von Möser und Böhme [14], betrug der Umsatz dieser Plattform in einem betrachteten Zeitraum von acht Monaten bis zu acht Millionen USD.

In diesem Kapitel werden sowohl die Funktionsweise der Plattform näher erläutert, als auch die Ergebnisse einer Studie, in der die Plattform über einen 8-monatigen Zeitraum analysiert wurde, zusammengefasst.

4.1 Funktionsweise JoinMarket

Die Plattform JoinMarket ermöglicht ihren Nutzern die Umsetzung von CoinJoin Transaktionen. Im Gegensatz zur „normalen“ CoinJoin Transaktion wurde hier aber ein Konzept entwickelt, das dem Nutzer jederzeit entsprechende Transaktionspartner für eine gemeinsame CoinJoin Transaktion zur Verfügung stellt. Dazu werden die Nutzer in sogenannte „Makers“ und „Takers“ eingeteilt [14].

Makers stellen ihre Bitcoins beliebigen Takers zur Durchführung von CoinJoin Transaktionen zur Verfügung. Zum einen erhöhen sie dadurch die Anonymität ihrer eigenen Transaktion (haben aber keine Eile ihre Transaktion durchzuführen) und zum anderen erhalten sie eine kleine Gebühr für die Bereitstellung ihrer Bitcoins.

Takers haben keine Zeit auf einen Transaktionspartner zu warten und verwenden deshalb die Bitcoins der Makers für ihre CoinJoin Zahlung. Im Gegenzug bezahlen sie die sogenannte „maker fee“ an die Makers [22].

Ein Nutzer, der möglichst schnell eine Transaktion durchführen möchte, nimmt also die Rolle eines Takers ein und wählt einen oder mehrere Maker aus, um mit ihm oder ihnen eine gemeinsame Transaktion durchzuführen. Erlaubt sind auf JoinMarket 2 bis maximal 20 Transaktionspartner in einer Transaktion. Die maker fee, die er als Gegenleistung bezahlen muss, ist dabei ein Prozentwert, der sich an der Höhe der Zahlung orientiert. Um eine Transaktion endgültig zu tätigen,

doi: 10.2313/NET-2017-05-1_01

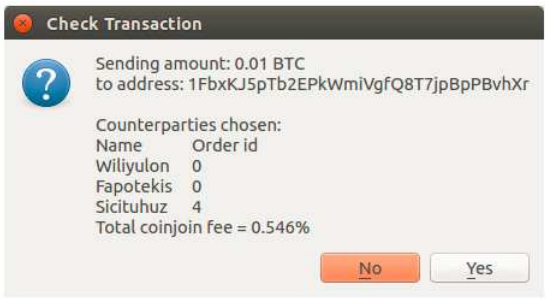


Abbildung 4: Beispiel: Eine Transaktion mit 3 Transaktionspartnern kostet 0,534% maker fee [22]

muss jeder Nutzer der entsprechenden maker fee zustimmen (siehe Abbildung 4). Andernfalls kann die Transaktion nicht durchgeführt werden [22].

In der derzeitigen Implementierung von JoinMarket kommunizieren Takers und Makers über einen zentralisierten Internet Relay Chat (IRC) Kanal. Ein Maker tritt dem Kanal bei und eröffnet sein Angebot für eine Bitcoin Transaktion. Die Angebote werden bei JoinMarket nicht auf einem zentralen Server gespeichert, sondern in einem öffentlichen Auftragsbuch, das jeder Teilnehmer lokal verwalten kann. Wenn ein Maker ein neues Angebot erstellt, ein bereits vorhandenes Angebot verändert, oder die Plattform verlässt wird die lokale Datenbank aktualisiert. Ein Maker muss seine Verbindung zum IRC Server halten, um an CoinJoin Transaktionen teilnehmen zu können [14].

JoinMarket ordnet Makers und Takers nicht automatisch zueinander. Ein Taker, der eine Transaktion durchführen möchte, hat bei der Auswahl seiner Transaktionspartner folgende drei Möglichkeiten [14]:

1. Zufällige Auswahl von einer Liste, die Angebot und zugehörige Gebühr entsprechend gewichtet
2. Auswahl nach Gebühr (niedrigste Gebühr zuerst)
3. Manuelle Auswahl

4.2 Analyse JoinMarket

Möser und Böhme [14] analysierten die Plattform JoinMarket über einen acht-monatigen Zeitraum von Anfang Juni 2015 bis Ende Januar 2016. Die Ergebnisse sind in ihrer Publikation „Join Me on a Market for More Anonymity“ beschrieben und werden nachfolgend kurz zusammengefasst.

Abbildung 5 zeigt die Marktveränderungen von JoinMarket im betrachteten Zeitraum. Die Gesamtzahl der verfügbaren Bitcoins entwickelte sich von anfangs wenigen hundert bis zu einem Maximum von über 2000 BTC Ende November 2015. Obwohl die Anzahl zum Ende des Betrachtungszeitraumes wieder bis auf 1500 BTC sank, wurden die Transaktionsangebote innerhalb der acht Monate nahezu konstant mehr. Da aber die Anzahl der Makers über den gesamten Zeitraum nahezu gleich blieb, schlossen die Forscher darauf, dass die Makers mehr und mehr dazu übergingen nicht mehr nur eine sondern gleich mehrere Installationen des Programms zu verwenden. Ein Fakt, der die nachfolgende Graphik verfälschen könnte, ist, dass die Plattform nicht gewährleisten kann, dass jedes Transaktionsangebot wahrheitsgemäß ist. Makers könnten beispielsweise mehr Bitcoins anbieten als sie eigentlich haben, oder besonders niedrige Gebühren verlangen und die Transaktion dann im letzten Schritt noch abbrechen.

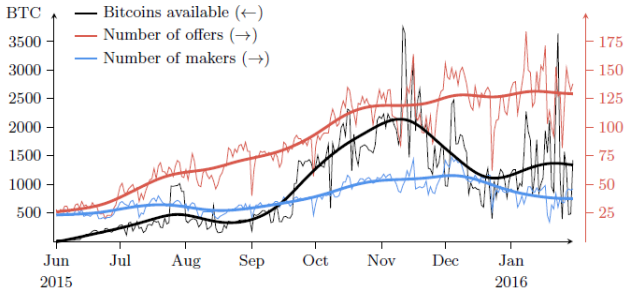


Abbildung 5: Marktveränderungen der Plattform JoinMarket im Zeitraum von 8 Monaten [14]

Die Anzahl der abgeschlossenen JoinMarket Transaktionen während der betrachteten acht Monate belief sich laut der Studie auf ca. 10.000, d.h. ungefähr 41 Transaktionen pro Tag. Die höchste Transaktionsrate wurde dabei im Oktober und November gemessen mit ca. 150 Transaktionen pro Tag.

Die wohl größte Gefährdung für die Nutzer besteht darin, dass die Plattform besonders anfällig für Sybil-Angriffe ist. Um einen Taker zu deanonymisieren müsste der Angreifer lediglich eine große Anzahl an Makers verkörpern und sich mit diesen in einer CoinJoin-Transaktion als unabhängige, einander unbekannte Individuen ausgeben. Ziel wäre es dann der einzige Partner des Takers in der entsprechenden Transaktion zu sein und - weil der Angreifer genau weiß welche Inputs und Outputs zu ihm selbst gehören - sofort aufzudecken wohin der Taker sein Geld überweisen möchte. Damit wäre der CoinJoin Mechanismus ausgehebelt und eine Transaktion genauso transparent wie im „normalen“ Bitcoin System.

Wie hoch die Wahrscheinlichkeit für eine erfolgreiche Sybil-Attacke aus Sicht des Angreifers ist, hängt stark von der Anzahl der Teilnehmer an einer CoinJoin-Transaktion ab. Je größer die Personenzahl bei einer Transaktion, desto geringerer ist die Erfolgsquote. Vor allem weil die voreingestellte Anzahl an Makers, die ein Taker für eine Transaktion benötigt auf JoinMarket bei 2 steht, schließen sich die meisten Takers nur mit zwei Makers für eine Transaktion zusammen. Auch nachfolgende Statistik (siehe Abbildung 6) belegt, dass sich nur sehr wenige Takers für mehr als zwei Makers entscheiden, wodurch ein möglicher Versuch einer Deanyonymisierung für einen Angreifer noch erleichtert wird.

Zusammenfassend bleibt zu sagen, dass JoinMarket zwar eine Lösung für das grundsätzliche Problem von CoinJoin Transaktionen - nämlich das Finden eines Transaktionspartners - liefert, die Implementierung aber im jetzigen Zeitpunkt noch nicht optimal ist. Zum einen schützt die Plattform die Nutzer nicht ausreichend vor Angriffen (v.a. Sybil) und zum anderen baut die Architektur auf einem zentralisierten IRC Server auf, was eigentlich diametral entgegengesetzt dem dezentralen Gedanken von Bitcoin ist.

doi: 10.2313/NET-2017-05-1_01

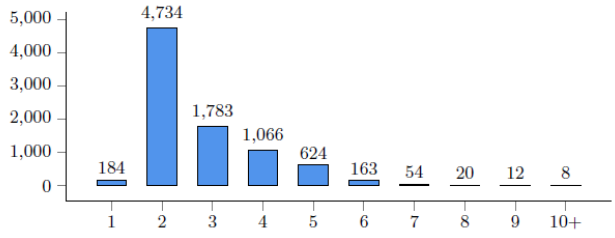


Abbildung 6: Anzahl der Makers, die von den Takers ausgewählt werden (Default = 2) [14]

Der Vollständigkeit halber muss an dieser Stelle noch erwähnt werden, dass JoinMarket nicht die einzige Implementierung des CoinJoin Mechanisums ist. Nach Young [22] gibt es folgende fünf weitere Implementierungen: „Mycelium CoinJoin wallet“, „CoinShuffle“, „Dark Wallet“, „Darksend of Dash“, „Shared Coins“.

5. ZUSAMMENFASSUNG

Diese Arbeit beschäftigte sich mit der virtuellen Währung Bitcoin und insbesondere der Frage, inwieweit der Nutzer bei der Bezahlung mit Bitcoins anonym bleibt. Um diese Frage zu beantworten war es zunächst wichtig das Konzept des Bitcoin Systems zu erläutern. Die Basis bildet ein dezentrales Peer-to-Peer System, das keiner staatlichen Kontrolle unterliegt. Ein Bitcoin Konto kann ohne Angabe persönlicher Daten eröffnet werden und wird über eine kryptographische Hashfunktion adressiert. Der öffentliche Schlüssel eines solchen Kontos ist allen Teilnehmer bekannt, der private Schlüssel nur dem Administrator. Alle Transaktionen von der Einführung des Bitcoin Systems bis heute werden in der Blockchain gespeichert und können von jeder beliebigen Person eingesehen werden. Werden die öffentlichen Schlüssel mit Informationen von externen Quellen (Facebook-Einträge, Blog-Einträge, IP-Adressen) verknüpft können Angreifer relativ schnell auf die Identität hinter den Konten schließen. In der Vergangenheit gab es bereits mehrere solcher erfolgreichen Hackerangriffe, bei denen Bitcoins im Wert von mehreren Millionen Dollar gestohlen wurden. Die Leidtragenden dieser Angriffe waren meist private Bitcoin Nutzer, die sich nicht mit zusätzlichen Maßnahmen vor solchen Attacken geschützt haben.

Doch welche Möglichkeiten gibt es die Anonymität des Nutzers besser zu bewahren? Dazu wurde in dieser Arbeit die JoinMarket Plattform näher erläutert. Diese basiert auf dem CoinJoin Mechanismus, bei dem - anders als im normalen Bitcoin System - eine Transaktion nicht mehr nur von einer Person zu einer Person abgewickelt wird, sondern sich mehrere Personen für eine Transaktion zusammenschließen. Dieser Mechanismus macht es deutlich schwieriger für einen Angreifer, die Identität einer Person nachzuvollziehen. Dennoch hat auch die CoinJoin Methode eine Schwäche und zwar das „Finden“ der Teilnehmer untereinander, um dann gemeinsam eine Überweisung zu tätigen. Dafür hat die JoinMarket Plattform das Prinzips der Takers und Makers eingeführt, bei dem die Makers Angebote für Bitcoin Transaktionen öffentlich machen und von den Takers gegen Bezahlung ausgewählt werden können, um eine dringende CoinJoin Transaktion zu tätigen. Eine in dieser Arbeit beschriebene Analyse zeigt deutlich, dass die Plattform JoinMarket im letzten Jahr an Attraktivität gewonnen hat, aber auch hier nicht von vollständiger Anonymität gesprochen werden kann. Besonders anfällig scheint die Plattform für Sybil-Angriffe zu sein und in der derzeitigen Implementierung gibt es noch keine Schutzmechanismen gegen diese Attacke. Das Fazit dieser Ausarbeitung ist, dass Bitcoin zwar anonymer als andere elektronische Zahlungsmittel, aber noch weit entfernt von vollständiger Anonymität ist.

6. LITERATUR

[1] J. Barcelo. User Privacy in the Public Bitcoin Blockchain. *Journal of Latex Class Files*, 6(1), 2007.

[2] J. Becker, D. Breuker, T. Heide, J. Holler, H. P. Rauer, and R. Böhme. Geld stinkt, Bitcoin auch-Eine Ökobilanz der Bitcoin Block Chain. In *GI-Jahrestagung*, pages 39–50, 2012.

[3] A. Biryukov, D. Khovratovich, and I. Pustogarov. Deanonymisation of clients in Bitcoin P2P network. *ACM Conference on Computer and Communications Security*, 2014.

[4] J. Breithut. Digitales Zahlungsmittel: Bank of America orakelt über große Bitcoin-Zukunft. *Spiegel Online*, Dec. 2013. `http://www.spiegel.de/netzwelt/web/bitcoin-zahlungsmittel-der-zukunft-a-937597.htmll`, zuletzt besucht am 22. August 2016.

[5] B. community. Block size limit controversy. *Bitcoin Wiki*, 2016. `https://en.bitcoin.it/wiki/Block_size_limit_controversy`, zuletzt besucht am 15. November 2016.

[6] B. community. Trade. *Bitcoin Wiki*, 2016. `https://en.bitcoin.it/wiki/Trade`, zuletzt besucht am 28. August 2016.

[7] O. Harman. Bitcoin - Hype oder Währung? Bachelorarbeit, Universität Bern, 2014.

[8] J. Herrera-Joancomartí. *Research and Challenges on Bitcoin Anonymity*, pages 3–16. Springer International Publishing, Cham, 2015.

[9] D. Hobson. What is Bitcoin? *XRDS*, 20(1):40–44, Sept. 2013.

[10] A. Krohn-Grimberghe and C. Sorge. Bitcoin - Anonym Einkaufen im Internet? In *Der gläserne Verbraucher - Wird Datenschutz zum Verbraucherschutz?*, pages 105–114, 2014.

[11] S. Meiklejohn and C. Orlandi. *Privacy-Enhancing Overlays in Bitcoin*, pages 127–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[12] S. Merklin, B. Schneider, and M. Schoop. Bitcoin - eine strukturierte Analyse. In *Multikonferenz Wirtschaftsinformatik 2014*, pages 2138–2149, 2014.

[13] T. Mochizuki. So lief die spektakuläre Pleite der Bitcoin Börse ab. *Wall Street Journal*, 2015. `https://www.welt.de/wall-street-journal/article129565422/So-lief-die-spektakulaere-Pleite-der-Bitcoin-Boerse-ab.html`, zuletzt besucht am 27. September 2016.

[14] M. Möser and R. Böhme. Join Me on a Market for Anonymity. *The 15th Annual Workshop on the Economics of Information Security*, 2016. `http://weis2016.econinfosec.org/wp-content/uploads/sites/2/2016/05/WEIS_2016_paper_58.pdf`.

[15] F. Nestler. Deutschland erkennt Bitcoins als privates Geld an. *Frankfurter Allgemeine Zeitung*, Aug. 2013. http://www.faz.net/aktuell/finanzen/devisen-rohstoffe/digitale-waehrung-deutschland-erkennt-bitcoins-als-privates-geld-an-12535059.html, zuletzt besucht am 22. August 2016.

[16] o.A. 65 Millionen Dollar Bitcoins gestohlen. *Handelsblatt*, 2016. http://www.handelsblatt.com/finanzen/anlagestrategie/trends/hackerangriff-65-millionen-dollar-bitcoins-gestohlen/13963284.html, zuletzt besucht am 27. September 2016.

[17] R. Peyrl. Digitale Währungen - Zahlungsmittel der Zukunft? *Zukunftsthema Oberösterreich*, 2015. http://www.ooe-zukunftsakademie.at/Zukunftsthema_digitaleWaehrung_2015.pdfl, zuletzt besucht am 22. August 2016.

[18] J. Platzer. *Bitcoin - kurz & gut*. O'Reilly Verlag, 2014.

[19] C. Sorge and A. Krohn-Grimberghe. Bitcoin: Eine erste Einordnung. *Datenschutz und Datensicherheit - DuD*, 36(7):479–484, 2012.

[20] C. Sorge and A. Krohn-Grimberghe. Bitcoin - das Zahlungsmittel der Zukunft? *Wirtschaftsdienst*, 93(10):720–722, 2013.

[21] M. Vogel. *Relevanz und Risiken von virtuellen Währungen am Beispiel von Bitcoin*. Hochschule Hof, Fachbereich Wirtschaft, 2016.

[22] J. Young. Advances in Anonymity: JoinMarket Releases Version 3 and GUI. *BTCManager.com*, 2016. https://btcmanager.com/news/tech/advances-in-anonymity-joinmarket-releases-version-3-and-gui/, zuletzt besucht am 23. September 2016.

doi: 10.2313/NET-2017-05-1_01

Visualizing Changing Probability Distributions

Björn-Aljoscha Kullmann
kullmann@in.tum.de

Supervisors: Paul Emmerich and Sebastian Gallenmüller
Seminar Future Internet WS 2016/17
Chair of Network Architectures and Services
Department of Informatics, Technical University of Munich

ABSTRACT

The visual representation of data is a powerful tool to aid scientific research and publications. Graphics can be used to illustrate the findings in an accessible way and help explore the meaning of experiments by displaying the data intuitively. Data often takes the form of probability distributions, such as the latency distribution of network traffic. These results can be hard to interpret for the human brain. Thus it is exceedingly important to aid the viewer with good graphical displays.

This paper gives an overview of how informative visualization can be achieved and how the data can unfold its story. Possible ways to deceive the viewer are explored and guidelines how to stay true to the data are given. Based on this knowledge, means to visualize individual data sets as well as series of experiment data are looked at. Two data series from the realm of network traffic measurement serve as example data for the graphics.

Keywords

Statistics, Graphical Display, Visualization

1. INTRODUCTION

The graphical representation is of great importance to every publication. Researchers rely on visual data representation to convey relevant information in publications. Especially statistical data is hard to understand without the right means to visualize it. This paper handles the central challenges when graphing probability distributions and highlights good ways to present statistical data accordingly. Two data sets are used to create the graphics for this paper. The realtime versus background traffic data set measures latency times in a stress test for high-speed network devices. The switch under test is tasked to prioritize forwarding the packets of the realtime traffic flow . The FLOWer concept utilizes the MoonGen packet generator and OpenFlow programmable switches to achieve high network traffic with customizable packets [5]. MoonGen is presented in the paper that is also the source of the other test data set [6]. Complex traffic patterns are generated by the novel packet generator to stress a software switch. The patterns looked at in this paper are a constant bitrate (CBR) flow and packets generated by a Poisson process [6].

The graphing tool used for the displays is the Python library matplotlib [10] and Seaborn [23], composed in Jupyter Notebook [15]. The code for the graphics is available in the LRZ GitLab repository [12].

The first chapter deals with the basic principles of visual presentation. The foundations of graphical perception are discussed and guidelines how to avoid pitfalls in the presentation of data. This is followed by various examples of how to present univariate data. The cumulative plot, dot plot and different kinds of histograms are explained. The sections dealing with the box plot and the violin plot show how summary indicators and distribution estimates can act in combination to produce a sound representation of the data. Finally, we take the step to visualize changing probability distributions over the course of a series of measurements. Utilizing advanced data representation techniques like graphic matrices, summary statistics and color makes the graphic overstep the two dimensions of paper to showcase multivariate data sets.

2. EXPRESSIVE DATA VISUALIZATIONS

2.1 Motivation behind Graphics: Presentation and Exploration

The most general term to describe data recorded in a human-readable fashion is the chart. Charts can take the form of a table to show numbers directly. If the data sets become too large to grasp from text alone or when relations in the data should be highlighted, a graphical chart should be chosen. Common charting tool like Microsoft's spreadsheet software Excel offer a preset number of chart topologies. Pie charts, bar charts or line charts can be created by a simple click. This might push users to present their data in a carelessly constructed display. A good graphing tool should instead provide users with means to implement their vision of a good display and help reveal the information hidden in the data [25]. Examples for powerful graphing tools are the programing language R [16] and the Python library matplotlib [10], the latter of which is used for the illustrations in this paper.

The first step when working with data should be to explore its meaning. Simple plots of the values from each batch or test series can give a first impression of the underlying distributions and relations. On that basis, more complex exploratory plots can be created to explore the facets and characteristics of the data. Once a thorough analysis of the data has been conducted, the results should be condensed into a well crafted presentation graphic. All conclusions made by the analyst should be easily reconstructible by looking at the graphic. Complete definitions and explanations are important to support full comprehension of the data [22].

doi: 10.2313/NET-2017-05-1_02

2.2 Scale and Comparability: Adapting data to a frame

To bring numbers into a graphic, we have to define the space they should exist in. The graphic frame is defined by its axis. On a two dimensional surface of a book page, these are formed by the horizontal x-axis and the vertical y-axis. The scales measure the contents of a frame along their axis. Scales are driven by the data. With nominal scales, data measures falling into different categories can be differentiated. These categories have no inherent ordering. An arbitrary order can be chosen as appropriate and categories are usually spaced out evenly along the axis. An ordinal scale enforces total ordering over all possible values. The value marks should be drawn on the axis in their ascending order. An interval scale can be applied when a range of values is looked at. Adding one value positioned at a point on the axis to another value, the resulting value will be placed at the position equal to the sum of the added values' distance to the axis origin. A ratio scale demands such behavior for magnitude ratio comparisons between values [25].

Multiple variables can be plotted on the same axis. They have to share the same unit and magnitude level. The International System of Units (SI) describes base classes and transformation rules for units like length, weight, time or temperature. From these base classes, composite measurements can be derived to represent an interplay of SI units, for example volume, pressure or power. Basic categorical dimensions and scalar values are not part of the SI system and are therefore called dimensionless measurements [25].

Choosing the scale greatly influences the perception of the data. It can be a good idea to choose the axes whose boundaries extend slightly beyond the minimum and maximum of the data. Including zero is beneficial, because it serves as a good baseline and point of orientation for the viewer. If the data demands an origin different from zero, the reasoning behind the new baseline should be made clear. If multiple graphics show similar data it can be wise to choose the same scale to facilitate comparison between them [22]. The frame lines can be adjusted to reflect properties of the data. Cropping the frame lines to extend only to the maximum and minimum values produces a so called range frame. Ticks along the frame line can not only be used to present a regular spacing of the data, but also to mark single values or special properties of the data, like quartiles [20]. Reporting too much information through the frame however might clutter the display and confuse the viewer [22].

All data should fit nicely into the frame. Choosing an oversized scale to incorporate large values can obscure small features of the distribution. Splitting up the graphic into multiple figures to show each cluster individually should be considered. An alternative approach would be to re-express the scale [21]. Moving from a linear to a logarithmic scale can be a good choice for rapidly growing data. Thereby, multiplicative distances between data points turn to additive and ratios to differences. The most common log base is 10, but the base most appropriate for the data should be chosen. Scale breaks may be chosen to avoid wasting blank space between values. Two tilted lines breaking a scale line should be used to indicate a partial scale break, the line resumes with at a higher value. Values must not be connected across scale breaks. A full scale break can even change the resolution of the scale. A full vertical line at each of the breaking ends should indicate a shift in scale [2].

When all the scaling work is done, the shape of the graphic itself has to be chosen. The proportions of the display should orientate themselves in relation to the data. Regression displays might benefit from a square frame, because the graphic is split in two equally large sections along the 45° line. Vertical displays imply stark growth, while horizontal frames facilitate the impression of change over time. In general, it is more pleasant to watch a graphic wider than tall. The proportions can be chosen accordingly. Following the Classical ideal of ancient Greece, aesthetic can be determined by the golden section with height $a = 1$ and the width $b = \frac{\sqrt{5}+1}{2} \approx 1.618$ [20]. The common aspect ratios in media can be a reference, too. The ratio $4 : 3 \approx 1.333$ leans more towards the square. The modern standard TV resolution $16 : 9 \approx 1.778$ is closer to the golden section. The cinema ration $19 : 10 = 1.9$ is almost twice as wide as it is tall. The graphics in this paper will use the $16 : 10 = 1.6$ ratio, because it is the standard that comes closest to the golden section.

2.3 Graphical Perception

The effect of a visual display is determined by the viewers perception of its content. Great data sets have no merit when the observer cannot make out its meaning. Values should be presented so that they can be judged effectively. Weber's law suggests that the contrast in magnitude of two physical values is not perceived by their absolute difference. Instead, the human brain distinguishes their ratio. Two line segments A and B that are not aligned are read as "line A is 30 percent longer than line B" instead of "the difference in length between between A and B is 2 centimeters". Therefore displaying values that differ in ratio is important when they do not align to a common base [2].

Steven's law establishes a metric for how judgment of magnitudes differ from reality. The perceived scale $p(x) = cx^{\beta}$ is skewed by a constant c and a power β to the magnitude x. When testing for the judgment of lengths, mild error factors β of 1 ± 0.1 have been estimated. For area, this error becomes more severe to levels of 0.6 to 0.9. It only gets worse for volume with average β of 0.5 to 0.8. Ratio judgment therefore become increasingly skewed for unintuitive attributes [2].

Cleveland assigns graphical elements to a hierarchy of graphical perception tasks [2]. Graphical elements that can be judged more easily should take precedence in graphical design and elements further down the hierarchy should only be chosen once the limits of better attributes have been reached. The hierarchy is divided into 7 levels. Attributes in the same level do not differ in their quality of magnitude perception.

1. Position along a common scale and axis
2. Position along a identical scales with nonaligned axes
3. Length
4. Angle and Slope
5. Area
6. Volume
7. Color hue, Color saturation, Density of occurrence

doi: 10.2313/NET-2017-05-1_02

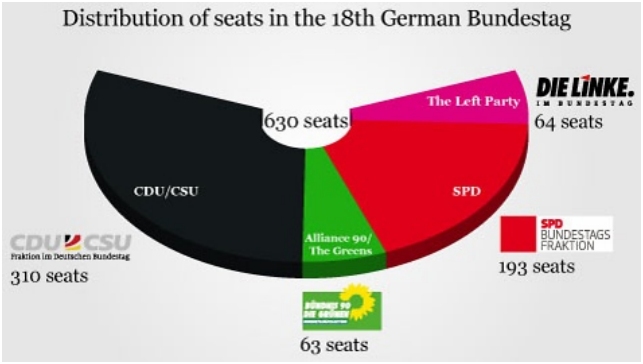


Figure 1: Party seat distribution of the 18th German parliament [3].

Tufte introduced the Lie Factor [20] to express how the physical measure on the graphic surface relates to the numerical quantities represented.

$$\text{Lie Factor} = \frac{\text{size of effect shown in graphic}}{\text{size of effect in data}} \tag{1}$$

A lie is induced by a misrepresentation of quantities. A linear value that is represented by an area has exaggerated graphical impact in relation to the more fitting line or dot along its axis. Even bigger perceived lies can be told when expressing a one dimensional value by volume portrayed through an additional foreshortened axis. In consequence, values become hard to decipher and heavily skewed, even before perceptive error comes into play. The principle to avoid this kind of lie is to have the number of informative dimensions depicted not exceed the number of dimensions in the data [20]. Every graphic property not bound to a data value introduces the potential of misjudgment.
A value is well represented by an area if it is naturally computed by the product of two values or the the integral over a line [20]. Compare the histogram to a bar chart. In the bar chart, the bars extend to a single value and are merely a figurative representation of a dot value. The histogram, described in later chapters, on the other hand groups observations over the bin range on the x-axis and maps their count to the y-axis [25].
A similar case can be made for pie charts. The viewer is challenged to compare angle and radian ratios combined with narrow areas [20]. Data that actually relies on polar coordinates can be visualized with a rose diagram. This type of graphic is inspired by the wind rose that has been used for centuries on navigational charts. It is especially useful for data distributed around compass points, for example wind direction data [25]. An exemption from this principle could be made to represent data in a familiar real world context [25]. The distribution of seats resulting from the parliamentary elections are often presented in a manner resembling the parliament hall. Figure 1 shows the distribution of the party seats along the traditional political spectrum from left to right (in this case from "behind", center-right to left), as they might actually be located in a sitting of parliament [3].

For the construction of graphics, less is often more. Erasing as much non-information carrying elements as possible makes a graphic more concise. The data-ink ratio measures the portion of ink that is actually devoted to showing numbers. All ink in the graphic should add information to the graphic that was not there before. This holds especially true for fancy decoration, extensive grid lines, frames and grid ticks [20]. Graphics today are mostly generated by computers and the underlying data is often publicly available as precise digital records. Grid lines lose importance for plotting and retracing exact data values. It becomes more essential to show the inherent information than the exact values [22]. Tick marks and labels can instead be used to mark important events in the data.
In conclusion, graphics should always meet their purpose to convey new interesting information. Small and highly labeled data sets are often better suited for tables and text. The graphic designer should bear the basic principles of graphical perception in mind. Only when information is accurately conveyed, the design of the data graphic is a success.

2.4 Kernel Density Estimate

The key challenge when looking at statistical data is to know the underlying probability distribution. One way to estimate the distribution function of the data is to compute the kernel density. The result is a continuous approximation of the probability function, prominently featured by the violin plot we expand on later. The kernel is a tool to assess the target probability function by weighing the samples of the data. It is formed by a probability measure that should be similar to the target probability function. In practice, standard kernel like the ones described below yield good results even for complex distributions. By taking a sample of n values x_i from the data set, the density function estimate can be deduced by aggregating the kernel $K(x)$ for each of the values x_i [14]:

$$\hat{f}(x_0) = \frac{1}{n}\sum_{i=1}^{n} K_h(x_0 - x_i) \tag{2}$$

A commonly used kernel is the the Epanechnikov kernel, a truncated parabola

$$K_E(t) = \frac{3}{4}(1-t^2) \quad -1 \leq t \leq 1. \tag{3}$$

The standard normal kernel, also called Gaussian kernel [25], given by

$$K_N(t) = \Phi(t) = \frac{1}{\sqrt{2\pi}} e^{-0.5t^2}, \tag{4}$$

can be more convenient for its smoothness [14].
Additionally, the kernel function is weighted by a bandwidth factor h such that $K_h(t) = \frac{K(t/h)}{h}$. The bandwidth determines how strong the kernel for each kernel sample influences a point in the density function. This limits the spread of the kernel [25]. Choosing a large value for h leads to oversmoothing, a small h results in an unstable multimodal estimate [14]. Assuming a normal distribution for the data, an optimized bandwidth is achieved by the normal reference rule [18]:

$$h = 1.06\hat{\sigma}n^{-1/5}. \tag{5}$$

Thereby $\hat{\sigma}$ denotes the estimated standard deviation and can be deduced from the sample standard deviation (ssd) of the data. A more stable choice for data deviating from the assumed Gaussian distribution is the interquartile range

doi: 10.2313/NET-2017-05-1_02

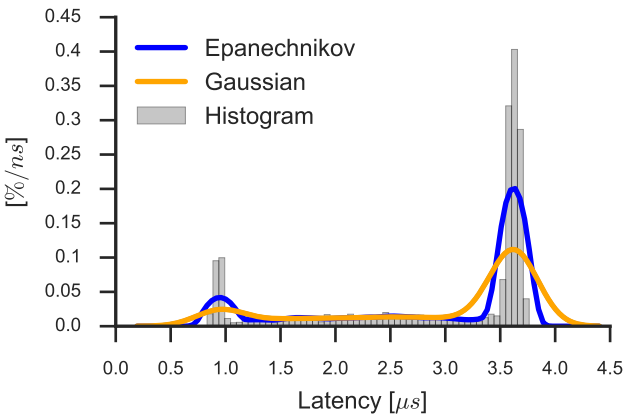


Figure 2: Kernel density estimates of priority traffic latency times with an Epanechnikov kernel and a Gaussian kernel. The bandwidth has been chosen by Scott's normal reference rule. A thinly binned histogram shows the distribution.

IQR of the data set: $\hat{\sigma} = min(ssd, IQR/1.34)$ [13]. The final kernel density function for a Gaussian kernel using the normal reference rule can then be computed:

$$\hat{f}(x_0) = \frac{1}{n}\sum_{i=1}^{n}\frac{e^{-((x_0-x_i)/h)}}{h}. \quad (6)$$

Figure 2 compares an Epanechnikov kernel and a Gaussian kernel estimate of a highly bimodal data set. Both estimates use Scott's normal reference rule for the bandwidth. The Epanechnikov kernel fits the peaks of the data more closely, while the Gaussian kernel applies stronger smoothing to the extreme values.

3. VISUALIZING UNIVARIATE DISTRIBUTIONS

3.1 Cumulative Plot

Measuring a statistical variable raises the question of its distribution. Discrete variables can easily be visualized by plotting their values against their respective absolute or relative frequency in the measurement. With continuous data this becomes impossible, since each observation might not be identical to any other observation. The frequency of individual continuous values is one and therefore insignificant in the overall data set. To visualize a continuous distribution we can use a cumulative plot. The graph lists the range of values on the x-axis. It starts from zero on the y-axis and each sample adds one to the vertical axis at the x-position of its value until all observations are processed. Figure 3 shows such a plot for the relay of prioritized traffic. Two slopes with an increased amount of arriving packages are visible at approximately $1\mu s$ and $3.7\mu s$ and a steady stream in between. Yet the display only shows the cumulated values and their intrinsic distribution is not obvious. A different approach is needed to show the distribution.

3.2 Dot Plot

The most simple solution to show a distribution is the dot plot. It shows the sample values on a one dimensional number line. It can be a good starting point to get to know a distribution [14]. The plot is only sensible for small data sets. Once the points start to overlap, it is possible to offset the point in the vertical axis [2], effectively forming a tally for a region of values on the number line corresponding to the size of the dot. Figure 4 shows such a stacked dot plot of a randomized sample of 100 data points in the realtime traffic data set. Note how the peaks of the bimodal distributions and the sparse values in between are clearly visible. Still, the display reaches its limit even for relatively few data points. Would we visualize more samples, the graphic would become convoluted and confusing.

3.3 Histograms

The concept of the histogram solves this problem. It is most commonly associated with the visualization of probability distributions widely used in publications from a wide range of scientific fields. This concept is also known by the term frequency diagram [1].
By dividing the range of the data values into intervals called bins, the number of observations falling into each bin become countable and bins can be plotted as bars [25]. For meaningful comparability of the bins, especially when bin sizes are different, it is important to note that the number of observations in a bin should be proportional to the area of the bin bar [1]. This ensures that larger bins are not excessively weighted in comparison to their smaller counterparts [14]. The vertical axis then shows the bin count divided by the bin width and the total number of observations. This denotes the average relative frequency for the bin.
The result is a density histogram, plotting the bin count v_j and the bin width h_j in the bin $B_j = [t_j, t_{j+1})$ that covers the interval between j and $j+1$, to the density estimate of each bin, relative to the sample size n [14]. The area of a bin then represents its bin count. The unit for the bin height is therefore in percent per width unit, in this case percent per nanosecond [8].

$$\hat{f}(x) = \frac{v_j}{nh_j} \quad x \in B_j. \quad (7)$$

Choosing an appropriate number of bins is the key step to make a histogram that reflects the data well. A rule of thump is to use $\sqrt{n}$ bins for n samples [24]. The bin width should furthermore respect a possible granularity of the data

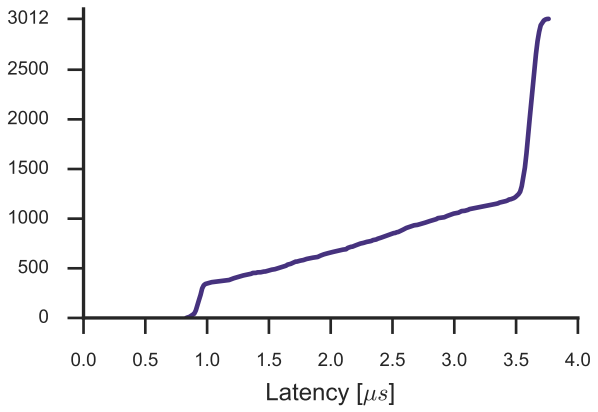


Figure 3: Cumulative plot of priority traffic latency times with 3012 observations.

doi: 10.2313/NET-2017-05-1_02

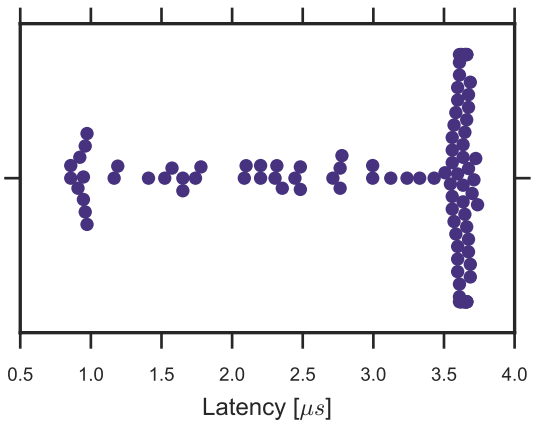


Figure 4: Dot plot of 8 Gbit/s realtime traffic with vertically offset dots to avoid overlapping.

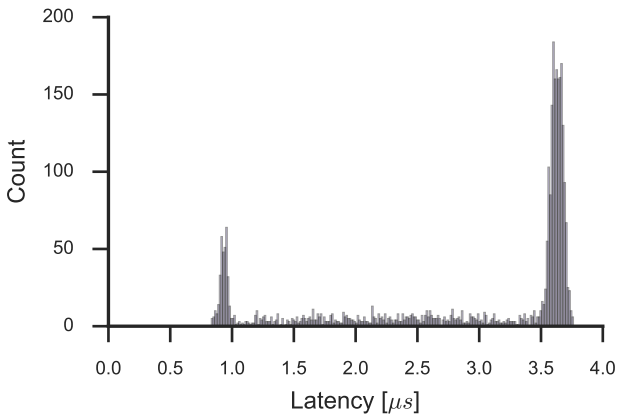


Figure 5: Histogram of priority traffic with a minimum bin size of $12.8ns$ determined by hardware granularity.

and should not be smaller than the smallest difference between two adjacent values [25]. Choosing the bin width as an integer multiple of the granularity can also add clarity to the display [26].
More sophisticated bin count estimates can be derived from the statistical attributes of the sample data. Sturges [19] deduces a good number of bins to be $k = \log_2 n$. This was further refined by Doane [4], who recommended $k = 3 + \log_{10} n \log_2 n$ to account for skewness [25]. To estimate a good bin width h of a bin, Scott [17] proposes the sample standard deviation ssd as the deciding factor to calculate $h_S = 3.5 ssd n^{-1/3}$. Similarly, Freedman and Diaconis [7] suggest using the more robust interquartile range IQR, resulting in about 30% lower bin width than Scott's for a normal density [14]:

$$h_{FD} = 2IQRn^{-1/3}. \tag{8}$$

The rules should only serve as a general guideline. For presentational graphics, the bin width should be chosen in a way that presents the data with a tolerable loss in accuracy [2].
The data for the example histograms is taken from the realtime and background traffic experiment. The data set has the following properties:

- Number of samples: 3012
- Minimum value: 832.0
- Maximum value: 3763.2
- Sample standard deviation: 997.62
- Interquartile range: $3635.2 - 2265.6 = 1369.6$
- Sample granularity: $12.8ns$

Figure 5 uses a number of bins that is determined by the granularity of the measuring hardware, resulting in 229 bins. The two peaks at the beginning and the end of the data range are clearly visible. The part in between is made up of a constant noise of packages coming through at other

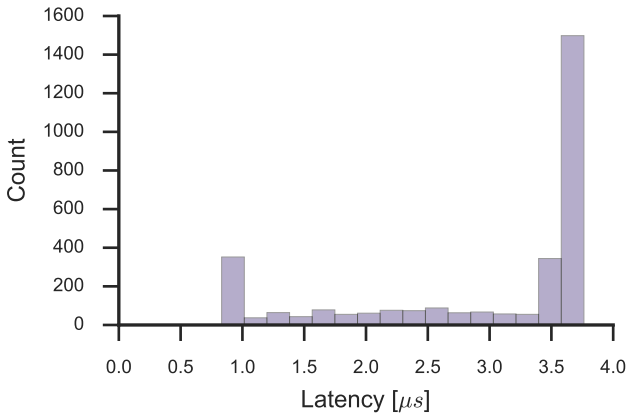


Figure 6: Absolute frequency histogram of priority traffic with the bin size determined by the Freedman-Diaconis rule.

latencies. The histogram of figure 6 applies the Freedman-Diaconis rule $h_{FD} = 2 \cdot 1369.6 \cdot 0,07 \sim 15.45$ rounded up to 16 bins. The bimodal distribution is still explicitly visible, but some of the local peaks in the data have been oversmoothed. The $\sqrt{n}$-rule results in a number of 55 bins in figure 7. Sturges' $\log_n$ only leads to 12 bins. Using the sample standard deviation according to Scott computes 255 bins, a result heavily skewed by the data's deviation from the normal distribution. A minimal histogram in figure 8 of 3 bins shows only the peaks and the noise in between with differing bin width. Grouping empty or sparsely occupied bins together can remove clutter from the display. When grouping bins together, it is important to keep the ratio between the bars intact and not mask data points that could be interesting to the reader.

3.4 Box Plots and Violin Plots

A schema [26] is a concise display that shows characteristic features of a distribution [2]. The box-and-whiskers-plot [21], or simply box plot, is the most common schematic plot, featuring the median, the first and third quartile, as well as a measure to identify values considered normal or

doi: 10.2313/NET-2017-05-1_02

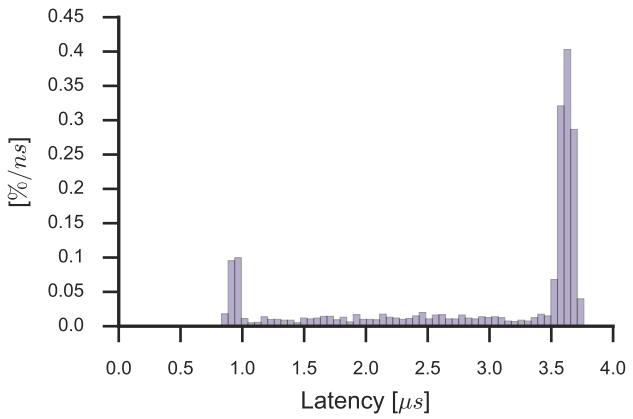


Figure 7: Density histogram of priority traffic with the bin size determined by the $\sqrt{n}$ rule.

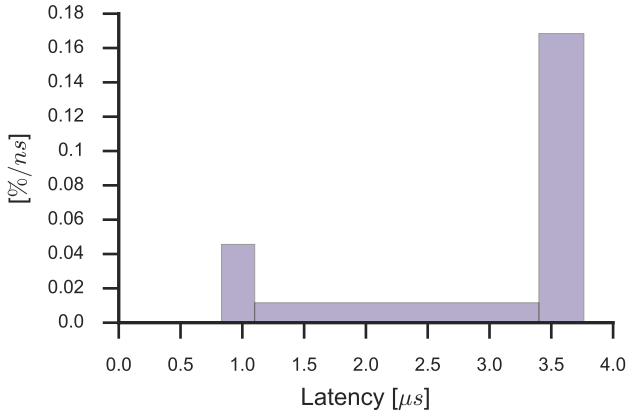


Figure 8: Density histogram of priority traffic with variable bin size to only show the peaks.

outliers.
The box plot was initially described by Tukey [21] and refined by Wilkinson [25]. The first step is to outline a thin box with its lower edge at the 25th quantile (1st quartile) and the upper edge at the 75th quantile (3rd quartile), forming the hinges of the box. The box is then crossed with a horizontal line at the position of the median.The whiskers stretch out from the hinges to the lower and upper fences of the plot. The position of the fences are determined by the interquartile range times one and half from the corresponding hinges: $whisker\_range = hinge \pm 1.5 \cdot IQR$. Values beyond these fences are treated as outliers and are each marked with a dot.
Figure 9 shows a box plot of constant bitrate traffic, which was generated by the MoonGen traffic generator. The median is closer to the top of the box, indicating a distribution skewed towards higher values. A single observation exceeds the normal range of the data with a latency of more than $140\mu s$.
The box plot is best suited for data with a steady unimodal distribution. Peculiarities such as bimodal distributions are masked by the terse display of summary statistics. The vio-

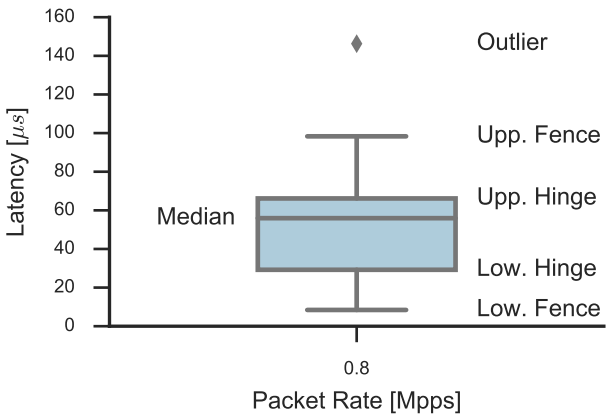


Figure 9: Box plot of latency times at $0.8 Mpps$ CBR traffic. The schematic features are labeled accordingly.

lin plot adds additional information to the box plot [9]. The box becomes a thick line and thinner lines extend to the positions of the upper and lower fences. A circle marks the position of the median. Then a kernel density estimate is plotted symmetrically to both sides of the box, resembling the form of a violin. Outliers are not tagged by any symbols. When comparing multiple violins, the scaling of the density estimate should be chosen according to the intended effect of the comparison. By scaling the violins to the same area or maximum width, the distributions can be compared effectively. Scaling proportional to the sample size places emphasis on the differences in data population.
The categorical nature of box and violin plot makes it possible to arrange the plots for multiple data series on one axis. A special variant of the violin plot is to make it asymmetrical [11]. Each side's density estimate shows a different subgroup of the data. This enables direct comparison of the two distributions.
Figure 10 shows the latency measurements for multiple packet rates in a single graphic. The kernel density graph for the CBR traffic, that is shown on the left of the box for the $0.8 Mpps$, reveals the deep valley that was hidden in the box plot visualization of figure 9. The density estimates are also easily comparable to the measurements for other packet rates. The bitrate CBR kernel density estimates form the left side of each violin and are opposed by the Poisson traffic type density estimates on the right side. A trend towards a more gradual slope for the Poisson latencies can be surmised.

4. MULTIVARIATE PLOTS

If we want to look at data varying in more than two dimensions, we have to find find strategies to make all the values accessible in the two-dimensional space of paper. When evaluating series of measurements, a so called small multiple display can help structure the data. The visualizations of each measurement are arranged as a matrix. Additional information can be encoded by using the vertical and horizontal axes to vary two different variables, or the series can be alloted arbitrarily for categorical or ordered from left to right and top to bottom for a single ordinal scale. It is especially important for this display that the design remains constant

doi: 10.2313/NET-2017-05-1_02

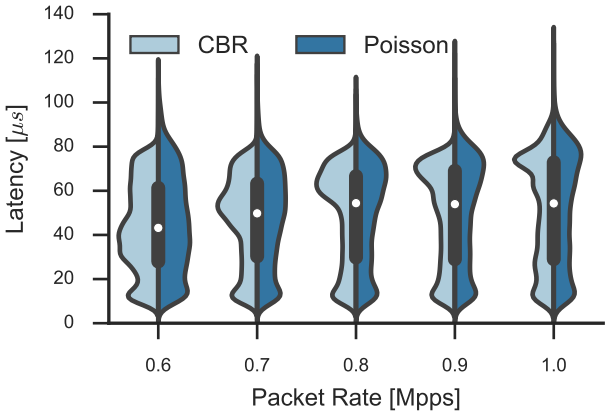


Figure 10: Split violin plot series of latency times from 0.6*Mpps* to 1.0*Mpps* CBR and Poisson traffic, scaled to the same area.

for all sub-graphics. All scales and color schemes have to be linked, enforcing an identical display for all frames [20]. Figure 11 shows a small multiple for the MoonGen data, comparing the constant bitrate against the Poisson process latencies of generated packets on the x-axis and measurements for different packet rates on the y-axis.

A simple way to reduce the dimensions of the data is to conflate each measurement to a numerical indicator. Summary statistics provide a simple solution for expressing a distribution in a single value. Taking the arithmetic mean or the interquartile range of a sample subset is inevitably associated with information loss, but trends in the data are still easily traceable. Figure 12 plots the medians of the CBR data and accompanies them with vertical lines corresponding to their interquartile range. This effectively produces a simplified box plot series.

Instead of introducing new dimensions in space, color can be used to represent a numerical magnitude. A heatmap features tiles arranged on the grid of a 2D graphic. Each tile is then colored by a third variable to reveal patterns in the distribution [25]. In color theory, color space can be modeled as a cube with the primary colors black, red, green and blue and the secondary colors white, magena, cyan and yellow serving as edges. These colors are the extreme coordinates and colors in between can be achieved by interpolating between the edges. To express a triple of data values, the numbers need to be normalized to fit inside the color cube and the result can be used as a color value to be painted on a graphic. Because color is likely perceived as ambiguous, graphics should resort to a single color dimension. Using brightness makes it possible to shade elements continuously, even without the need to print color. Hue is the spectral component of color, for example red, green or purple. Easily distinguishable color schemes are well suited to represent categorical data. Saturation describes the degree of pure color, or rather amount of hue, in a patch. This color dimension transits from fully saturated color to gray without changing the brightness. An easy distinction between different levels of saturation is not easily possible. An attribute like uncertainty might therefore be well suited to be expressed by saturation [25].

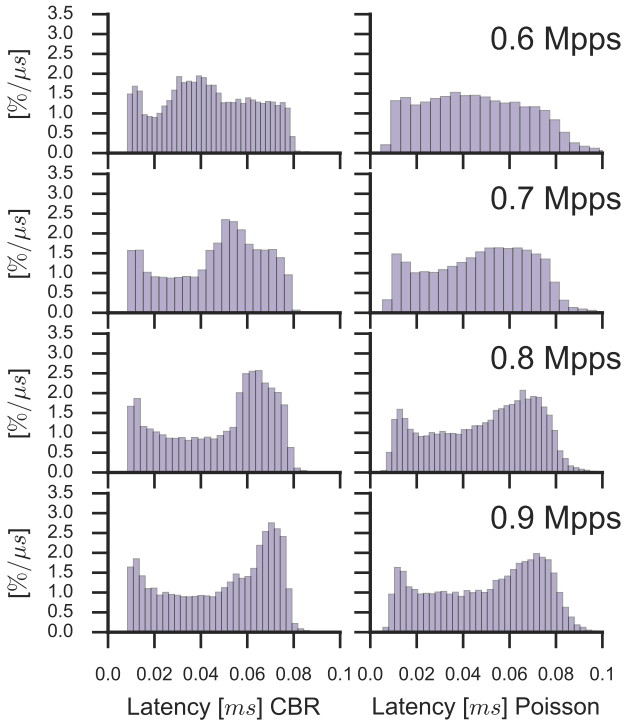


Figure 11: A small multiple comparing CBR and Poisson traffic over the course of four measurements with different packet rates, indicated in the top right of each row.

Figure 13 uses 55 bins to present the Poisson traffic data for different packet rates. The color reaches from a light blue for low frequencies to a dark blue for highly populated bins. Each packet rate row is normalized in order to give the rows with lower packet rates and fewer events the same visual impact as the rows with higher packet rate and therefore higher event density. The valleys and peaks are visible, but more smoothing might be necessary for the color histogram to become a sound data display.

5. CONCLUSION

Creating a good statistical graphic means to iterate many steps of graphical design, evaluation and redesign. When confronted with a probability distribution, basic plots like the dot plot and the histogram can give a good idea of the data. Bivariate data can be graphed on a scatterplot.

The second step is to take it further and highlight relations and peculiarities inherent in the data. Kernel density estimates and summary statistics help create concise comparisons of data subsets. A third variable dimension can be used by introducing color, in form of a heatmap or a color histogram. The analyst is encouraged to experiment creatively. The best way to present the data at hand might not be found in traditional charting conventions. A good graphing tool aids the analyst on the quest to find the perfect visualization.

References

[1] G. E. Box, J. S. Hunter, and W. G. Hunter. Statistics for experimenters. 2005.

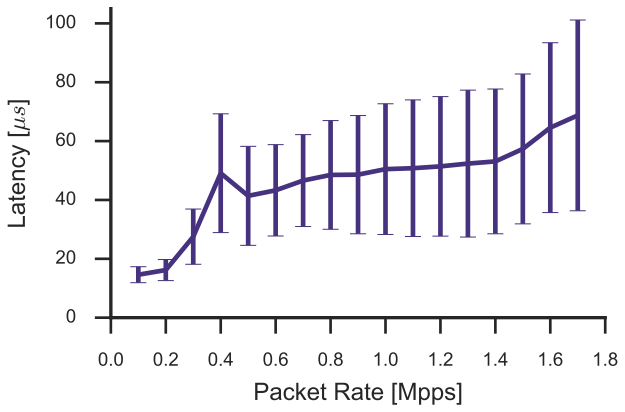


Figure 12: CBR traffic latency means with IQR ranges modelled as error bars.

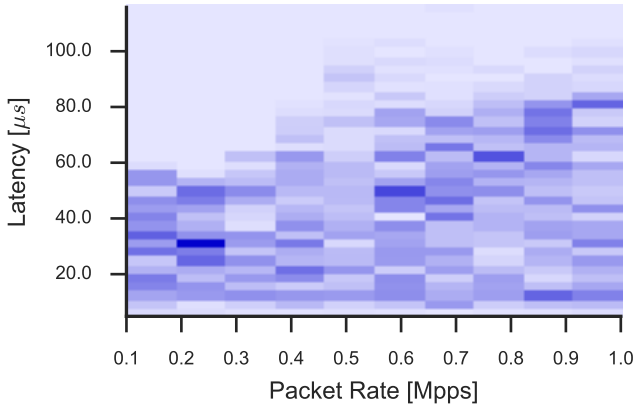


Figure 13: A color histogram of Poisson traffic for a series of tests with different packet rates. The more intense blue shows regions with higher frequency of occurrence.

[2] W. S. Cleveland et al. *The elements of graphing data*. Wadsworth Advanced Books and Software Monterey, 1985.

[3] Deutscher Bundestag. Sitzverteilung des 18. Deutschen Bundestages. `https://www.bundestag.de/bundestag/plenum/sitzverteilung_18wp`, 2015. Accessed on September 26th 2016.

[4] D. P. Doane. Aesthetic frequency classifications. *The American Statistician*, 30(4):181–183, 1976.

[5] P. Emmerich, S. Gallenmüller, and G. Carle. Flower – device benchmarking beyond 100 gbit/s. In *IFIP Networking 2016*, Vienna, Austria, 2016.

[6] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Conference on Internet Measurement*, pages 275–287. ACM, 2015.

[7] D. Freedman and P. Diaconis. On the histogram as a density estimator: L_2 theory. *Probability theory and related fields*, 57(4):453–476, 1981.

[8] D. Freedman, R. Pisani, and R. Purves. Statistics. *New York*, 1980.

[9] J. L. Hintze and R. D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, 1998.

[10] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[11] P. Kampstra et al. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of statistical software*, 28(1):1–9, 2008.

[12] B. Kullmann. Code for the Future Internet Seminar. `https://gitlab.lrz.de/kullmann/Future_Internet_Seminar`, 2016. Accessed on September 28th 2016.

[13] B. Liu, Y. Yang, G. I. Webb, and J. Boughton. A comparative study of bandwidth choice in kernel density estimation for naive bayesian classification. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 302–313. Springer, 2009.

[14] M. C. Minnotte, S. R. Sain, and D. Scott. Multivariate visualization by density estimation. In *Handbook of Data Visualization*, pages 389–413. Springer, 2008.

[15] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, 2007.

[16] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2008.

[17] D. W. Scott. On optimal and data-based histograms. *Biometrika*, 66(3):605–610, 1979.

[18] D. W. Scott. *Multivariate density estimation: theory, practice, and visualization*. John Wiley & Sons, 1992.

[19] H. A. Sturges. The choice of a class interval. *Journal of the American Statistical Association*, 21(153):65–66, 1926.

[20] E. R. Tufte and P. Graves-Morris. *The visual display of quantitative information*, volume 2. Graphics press Cheshire, 1983.

[21] J. W. Tukey. Exploratory data analysis. 1977.

[22] A. Unwin. Good graphics? In *Handbook of data visualization*, pages 57–78. Springer, 2008.

[23] M. Waskom et al. seaborn: v0.7.1. `https://doi.org/10.5281/zenodo.54844`, 2016.

[24] M. B. Wilk and R. Gnanadesikan. Probability plotting methods for the analysis of data. *Biometrika*, 55(1):1–17, 1968.

[25] L. Wilkinson. *The grammar of graphics*. Springer Science & Business Media, 2006.

[26] G. Wills. *Visualizing time: Designing graphical representations for statistical data*. Springer Science & Business Media, 2011.

Comparison of Efficient Routing Table Data Structures

Dominik Schöffmann
Betreuer: Sebastian Gallenmüller
Betreuer: Paul Emmerich
Seminar: Future Internet WS2016/17
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: schoeffm@in.tum.de

ABSTRACT
The topic of this paper is to compare multiple IP address longest prefix matching lookup schemes, especially algorithms implemented in software. Goals which are key to constructing an efficient lookup algorithm are explained. Different data structures and their respective approaches to size compression get illustrated. These approaches include a hardware based algorithm called "DIR-24-8" for better reference, and multiple software based algorithms. Different trie structures are reviewed in detail. Tests about lookup acceleration were conducted by the author and are presented and discussed in this paper.

Keywords
routing, lookup, IP, trie, dxr, poptrie

1. INTRODUCTION
"The datagrams are routed from one internet module to another through individual networks based on the interpretation of an internet address." [1] This quote from the IPv4 protocol specification dating back to 1981 still is the core of our modern Internet as we know, and use, it today.

Nowadays, routers within Internet Exchange Points (IXPs) are announcing routes to specific subnets, to each other, thus compiling comprehensive databases of next hops to those subnets. Once the packets enter an autonomous system, other routing protocols take over and guide the datagram to its destination. Oftentimes, there is talk about these algorithms used to exchange routes, like BGP and OSPF, whereas the methods of actually using this information is regularly hidden in a black box fashion. Either specialized hardware from well-known vendors such as Cisco, Juniper or Huawei is found inside the datacenters, or just plain general purpose x86 hardware using an open source Operating System (OS). The almost definitely most used OSs are Linux and FreeBSD. Both are highly reliable, well tested and deployed in a multitude of places around the world.

This paper aims to give an overview of popular and fast data structures which are either implemented in the aforementioned OSs, or are available in specialized libraries. These libraries are particularly interesting in combination with modern high performance packet frameworks such as netmap [2] or Intel DPDK [3]. As a matter of fact, Intel DPDK does include an implementation of a modified version of the DIR-24-8 algorithm which is explained later [4]. The pfSense router distribution is currently planning to deploy netmap in combination with a path compressed trie, which also is explained later [5].

2. RELATED WORK
Various authors published research regarding routing lookup data structures. Some dating back into the last millennium others were just recently presented. Since this paper gives details about the various algorithms, previous research is not presented in depth at this point.

Gupta et al. [6] developed a hardware optimized routing algorithm called DIR-24-8. This algorithm is not in use today in its original form, but a basic understanding is useful nevertheless.

Nilsson and Tikkanen [7] published a research paper about dynamic tries and compression techniques which might be utilized. Their work deeply influenced the lookup scheme of the Linux kernel [8].

The trie which is used for lookups within the FreeBSD kernel is described in a book written by McKusick and Neville-Neil [9].

More recently, specialized library implementations were developed and presented. The DXR algorithm is described by Zec el al. [10], which borrows some aspects from the DIR-24-8 algorithm, but performs better in software.

Poptrie is the fastest software based routing lookup algorithm presented in this paper. It is the only data structure utilizing specific instructions available in modern CPUs. It was published by Asai and Ohara [11] in 2015.

3. OPTIMIZATION GOALS
In order to build a fast routing lookup data structure, multiple factors are important. All of them are direct consequences of the hardware architecture, most notably the CPU, in use.

If the data structure is too big to fit inside the CPU cache, main memory accesses are needed. Such accesses introduce a high latency, during which no work inside the processing thread can be performed. Therefore the core might idle and thus waste resources.

Another important element is the number of memory accesses which need to be performed within one lookup. Even

doi: 10.2313/NET-2017-05-1_03

if the data structure is small enough to fit into the L3 cache, getting the data from there might still be expensive. If data needs to be fetched from the L3 cache, it is better to have as few of such accesses as possible. Modern cache lines store 64 bytes each, therefore all accesses transfer 64 bytes in one step. It is more efficient to store more data in the same cache line and thus utilize the size of a cache line, than to distributing it across multiple cache lines.

Furthermore, specifics about the targeted CPU architecture can also be of use, in order to build efficient data structures. This includes single instructions as can be observed with the poptrie algorithm. [11]

4. HARDWARE LOOKUP

The vast amount of traffic flowing through datacenters and IXPs are handled by hardware based routers. Those routers contain application-specific integrated circuits (ASICs) which are built to allow fast routing lookups. When reviewing routing algorithms, it is important to keep the differences of the underlying hardware in mind, in order to choose which method fits the given platform best.

4.1 DIR-24-8

The DIR-24-8 is a hardware based routing lookup algorithm, which was first introduced by Gupta et al. in 1998 [6].

The basic design uses three distinct tables, which are called "TBL24", "TBLlong" and "Next Hop". Routes with a prefix length of equal or less than 24 bits are completely stored in the TBL24, prefixes longer than that need to use the TBLlong. In the next hop table the IP addresses of all the next hops, which might be used by any route are stored. An IP address which is to be routed using this scheme is split into two parts, the first 24 bit and the last 8 bit. The upper part is used to index the TBL24 which contains 2^{24} entries, whereby each entry can have two different kinds of content. Depending on the routes, there might, or might not, be a route having a prefix length of more than 24 bit, sharing the first 24 bit with the IP address for which the lookup is performed. In this case the TBLlong is used. [6]

If no such route exists, the entry's most significant bit is set to 0. The 15 least significant bits are then used to index the next hop table. In this case, the lookup is finished. [6]

Whenever there is a route with a prefix length, longer than 24 bit, the table TBLlong is needed. This is indicated by a 1 in the most significant position of the corresponding TBL24 entry, which then also contains an offset into the TBLlong. Each of the 256 possible options of the last 8 bit of the IP address are represented by their corresponding next hop inside of the TBLlong. As soon as the next hop index is extracted from the TBLlong and used to index the next hop table, the lookup is finished. [6]

The DIR-24-8 algorithm is suitable for hardware implementations for multiple reasons. Oftentimes the next hop is found using a /24 entry of the routing table. All of these entries are stored in the TBL24, which furthermore has a fixed size. Therefore this table can be realized in hardware with a high-throughput and low-latency link to the CPU. The other two tables usually are not very large and thus fit into a reasonably sized RAM.

There exist multiple extensions to this data structure in order to save memory or memory accesses, and thus speed up the lookup. [6]

5. SOFTWARE LOOKUP

Although a significant part of the Internet's backbone infrastructure is powered by hardware routers, software lookup algorithms play an important role. As software routers and general purpose CPUs gain maturity and speed, they may be viable options in the future.

5.1 Naive

The naive lookup algorithm is quite simple. All of the routes are contained inside of a list, which is sorted by the prefix length. Longer prefixes are at the beginning of the list, shorter ones at the end.

Every time an IP address is to be routed, this list is traversed until the first match. In the worst case, this is the default gateway. A match is computed as follows:

$$(IP\&(\sim(2^{32-length})-1)) \oplus prefix \quad (1)$$

whereby *length* denotes the prefix length, & a binary AND operation, $\sim$ the one's complement, and $\oplus$ a binary XOR operation. If the result is zero, the route matches and the next hop is found. In case it is not zero, the next entry needs to be checked until a match is found.

5.2 Trie

Most modern lookup algorithms use some kind of tree as their basic data structure. Commonly, this tree is in fact a radix tree, which is also called a trie. The invariant of a trie is, that all the child nodes of a node share a common prefix, which is the key of the node.

Tries can be used is a number of different fashions, meaning that there is not one single valid basic lookup scheme using tries. The algorithm presented here is based upon the author's intention of simplifying the basic algorithm and the work of Nilsson and Tikkanen [7].

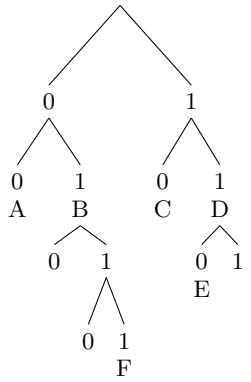


Figure 1: Basic trie

Figure 1 shows a basic trie, demonstrating the prefix sharing. The corresponding routing table if given in Table 1. The

doi: 10.2313/NET-2017-05-1_03

Node	Route	Next Hop
A	0.0.0.0/2	1.2.3.4
B	64.0.0.0/2	2.3.4.5
C	128.0.0.0/2	3.4.5.6
D	192.0.0.0/2	4.5.6.7
E	192.0.0.0/3	5.6.7.8
F	112.0.0.0/4	6.7.8.9

Table 1: Basic trie, Routing table

terms "left" and "right" are used to indicate a 0-bit and a 1-bit respectively. Inside of the trie, a route matches at exactly the position corresponding with the prefix.

When looking up an IP address, the n-th most significant bit of the address is used to index the n-th level of the trie. The trie is traversed using this fashion, until a leaf is found. Once this is achieved, the route at the leaf might match the IP address in question, or might not match it. If it does match, the search is over. If it does not match, the trie needs to be backtracked upwards again.

For example, if the IP address which is to be looked up is 128.0.1.24, the algorithm would take the first branch to the right, and the next branch to the left, reaching the leaf labeled C. Indeed this route matched the IP address, and the lookup was successful at this step.

Another example is the address 96.4.5.6, which would branch left, then two times right, and left again. The leaf which is found does not contain any route. Since the algorithm did not reach a leaf which corresponds with a route matching the IP address, upwards traversal of the trie is needed. As soon as the node B is rediscovered, the algorithm found the longest matching prefix and finishes.

It should again be noted, that this is not the only possible method of making use of a basic trie, but was chosen for simplicity. For example one simple extension would be to save the last found route, in order to avoid the upwards traversal.

5.3 Path compression

When developing a routing lookup algorithm, the size of the data structure, and the number of memory accesses, are of big importance. One method to decrease the number of needed memory accesses is called "path compression". Again, different implementations exist, which utilize path compression in their own way. In the first part, the notation of the previous section is preserved, and the compression is influenced by Nilsson and Tikkanen [7]. The second part describes the approach of FreeBSD.

Path compression allows the data structure to skip intermediate nodes which only have one child each.

Figure 2 shows two tries, whereby Figure 2a is a none compressed trie and Figure 2b is the path compressed version, representing the same data. The routing table corresponding to Figure 2 is given in Table 2. Empty leafs are omitted for brevity.

So far the change is straight forward, by just adding an intermediate node, which represents a path of nodes which would only lead to this subtrie.

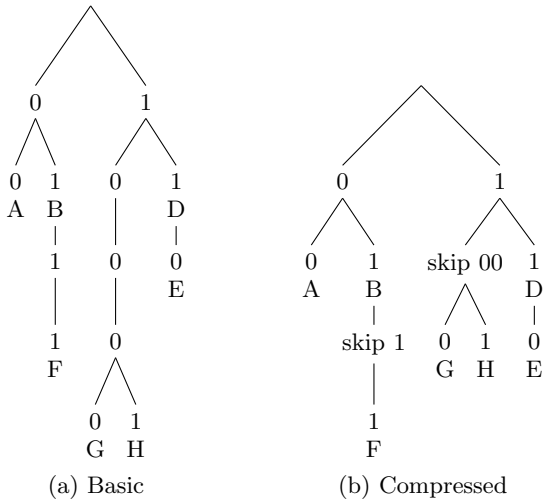


Figure 2: Path compressed trie

Node	Route	Next Hop
A	0.0.0.0/2	1.2.3.4
B	64.0.0.0/2	2.3.4.5
D	192.0.0.0/2	4.5.6.7
E	192.0.0.0/3	5.6.7.8
F	112.0.0.0/4	6.7.8.9
G	128.0.0.0/5	7.8.9.0
H	136.0.0.0/5	8.9.0.1

Table 2: Path compressed trie, Routing table

The FreeBSD Operating System actually uses such a path compressed trie as its IP lookup scheme, although in a different flavor. In the examples used here, routes could be internal nodes, whereas in the FreeBSD implementation, routes are always leafs. [9][12] The same routing table, as above, is also represented in Figure 3, using the FreeBSD implementation. It should be noted, that the numbers of the internal nodes depict bit positions, on which the trie branches. Furthermore, in this representation, two different routes sharing the same prefix, but having a different prefix length cannot be handled solely within the trie. As a solution, the nodes have a list of routes, which are checked in the order of the prefix length. Longer prefixes get checked first. Internal nodes can reference leafs, which is useful for the backtracking, and depicted in Figure 3 as the dashed lines.

Again, an example lookup may help to understand this data structure. The IP address to be looked up is 96.45.56.67. For simplicity, let's split the first octet up into its binary representation: $96_{10} = 01100000_2$ This means, that we need to take a left branch followed by two right branches, and left branches from there on.

Following these directions reveals the leaf F which does not match the IP address in question. The algorithm backtracks one level up, where is encounters a reference to the leaf B. B in return holds a route which matches the IP address. This terminates the algorithm.

doi: 10.2313/NET-2017-05-1_03

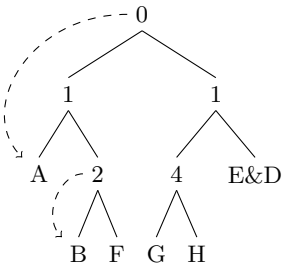


Figure 3: FreeBSD compressed trie, Adapted from the FreeBSD book [9]

Another interesting lookup yields the IP address 168.1.2.3. Again the binary representation of the first octet is helpful: $168_{10} = 10101000_2$, which means that the bits 0, 2, and 4 are set. In this case, the zeroth bit gets tested and evaluates to 1, so the search continues in the right subtrie. The second most significant bit (position 1) is 0, which means, that the left branch is taken. A 1 at position 4 means, that the route H is tested for a match, which is negative. Due to this, the backtracking process begins. The nodes labeled "4" and "1", which get traversed in this order, do not reference any leafs. The process continues to the root node, which references a route which does not match. Since the root is already reached, there is no further node to visit. The lookup was unsuccessful, and the packet cannot be routed, because the routing table does not contain a route to this host.

It should be noted, that this rarely reflects the reality, since almost always a default gateway exists, which can be used as a last resort.

5.4 Level compression

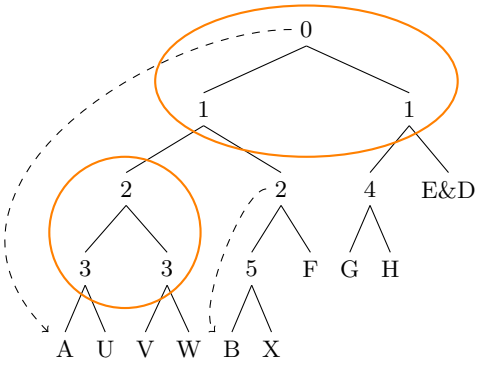
Although path compression is a pretty good start, for reducing the memory consumption and accesses, level compression can further help to achieve these goals. As the name already suggests, multiple nodes on the same level get merged, in order to have more information at the same memory location.

Saving nodes means, that less pointers to nodes need to be held, which is good for the memory consumption, and it also means, that less nodes need to be accessed to find the desired information, which is helpful to counter expensive memory accesses. The increased size of a single node is only a minor concern, because it usually still easily fits inside one cache line.

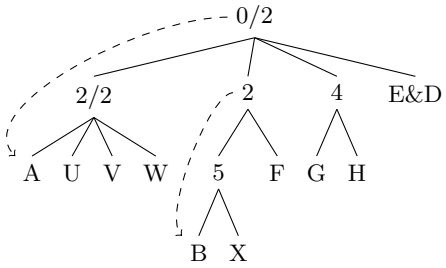
A node can be doubled when all of its children themselves have the same amount of children, and all children split at the same bit position for path compression.

Figure 4a shows an extended version of the FreeBSD trie used above. A level compressed version of this trie is given in Figure 4b. As can be seen, the level compression merges multiple nodes into a single one. The notation "x/y" of the internal nodes means, that this node starts comparing bits at position x for a length of y bits.

The internal structure of the nodes and therefore parts of



(a) PC-Trie



(b) LPC-Trie

Figure 4: Level compressed FreeBSD trie, Adapted from the FreeBSD book [9], Nilsson and Tikkanen [7]

the algorithm are dependent on the actual implementation to be used. The algorithm presented for path compressed tries still holds for this kind of level compressed trie, with only minor changes.

Although the FreeBSD trie does not use level compression, the Linux kernel does. [8] The Linux kernel furthermore adapts an optimization presented by Nilsson and Tikkanen [7]. Using the strict criteria presented in this paper imposes great cost on the maintenance of the trie. Abstaining from the demand that no child node is empty, but introducing thresholds as to when to double or halve a node allows for good compression at reduced cost.

5.5 Poptrie

Poptrie is a high performance IP lookup data structure based upon LPC tries, developed by Asai and Ohara [11]. It incorporates both path and level compression, as well as support for a new CPU instruction commonly found in commodity hardware, namely "popcount". This instruction counts the bits set to 1 in an integer. The data structure is highly compressed in order to fit into lower level caches, compared to other schemes, which may have to make use of main memory or the L3 cache.

Poptrie maintains two arrays, one for the internal nodes, and one for the leafs. An internal node is a struct, which contains an offset into both of these arrays, and a field of bits. As usual the IP address is used as the key into the data structure which is divided into multiple parts, one for each level. The part of the IP address to be processed at the moment is used to index the bit field. If the corresponding

doi: 10.2313/NET-2017-05-1_03

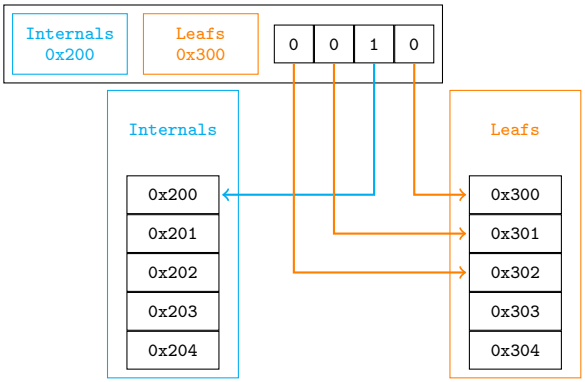


Figure 5: Poptrie illustration(adapted from Asai and Ohara [11])

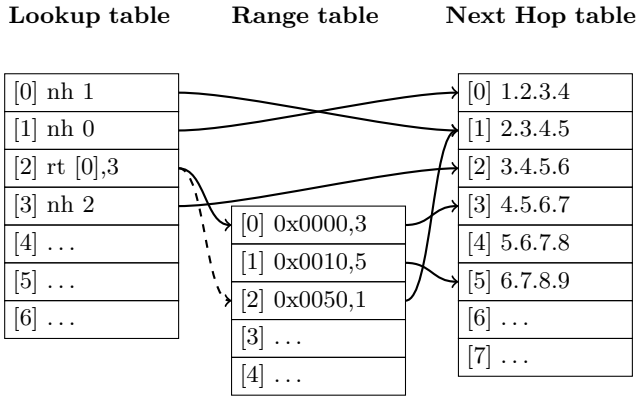


Figure 6: DXR data structures, redrawn from Zec et al. [10]

bit is a 1, the next step is to use another node, if it is a 0, a leaf is reached. This mapping is illustration in Figure 5.

In case a leaf is not yet reached, but the trie is further traversed, the bit field is masked, to force the current index and everything of higher significance to 0. The remaining 1s in the field are counted. At this point the popcount instruction can be used to speed this process up. The resulting number is added to the offset into the array which is saved in the current node, and thus the next node is found.

As soon as a leaf will be accessed in the next step, the n-th least significant 0s are counted, whereby n is the current part of the IP address. Analogous to the internal nodes, this number is added to the offset and then yields the position of the searched leaf. The leaf contains the next hop to be used. This also means, that routes are projected similarly to DIR-24-8.

Multiple extensions are developed, which further reduce the size of the data structure and save traversal steps. The leaf vector extension reduces the number of leafs with the same content, by introducing a new vector inside the internal node. Another extension is direct pointing, which works by using the first x bits of the IP address to index an array of internal nodes and next hops. These internal nodes are anyways traversed for nearly all address lookups. This reduces the number of steps and memory lookups and thus increases the performance.

5.6 DXR

The DXR algorithm was developed by Zec et al. [10] in 2012. It has similarities to DIR-24-8 and the direct pointing extension of poptrie, but does not build upon a trie itself. Route projection, as with DIR-24-8 and poptrie is also used, meaning the discrete routes are expanded into ranges within the continues IP address space.

Three arrays are used inside of the DXR algorithm, which are called the “lookup table”, the “range table” and the “next hop table”. The lookup table is indexed using the first n bits of the IP address, thus having 2^n entries. Commonly used values for n are 16 and 18. An entry can have two different kinds of content. Either a route with a prefix longer than n exists, which thus cannot be handled inside the lookup table, or such a route is not present. If there is no such route for a given n-bit prefix, the corresponding entry in the lookup table directly references an entry in the next hop table. This can be seen in Figure 6 for the entries 0x00, 0x01, 0x03.

In the likely case, that a route with a prefix length of more than n bits needs to be handled, the range table is used. Therefore the lookup table entry contains an offset into the range table, and the number of entries corresponding to this lookup table entry. An entry in the range contains the first IP address of the range, and the next hop responsible for the IP addresses between this first IP address (including) and IP address of the next entry (excluding). In order to find the correct range entry, a binary search algorithm can be used. An example for a range table is also included in Figure 6.

According to Zec et al. [10], DXR is, depending on the exact setting, up to 3.5 times as fast as a DIR-24-8 software implementation.

6. PRACTICAL EXPERIMENT

In the scope of this paper, the naive lookup scheme and a basic trie were developed and tested. The used dataset was obtained from the IXP in Amsterdam. Both algorithms were developed in an address-by-address and a batched fashion. Furthermore cache prefetching was evaluated. All tests were preformed on an Intel i5-5200U CPU, and GCC 5.4.0 was used. This CPU has a cache of 3 MB.

For the naive approach, neither optimization, batching nor prefetching, did yield any performance speedup. Using plain C arrays instead of C++ STL containers, and enabling compiler optimization, were very effective. These methods reduced the time needed to successfully route 100000 addresses from 10 seconds to 0.2 seconds.

For a basic trie the performance gain is still small, but noticeable. The time to route 10000000 addresses in dependence of the batch size is presented in Figure 7.

A batch size of one indicates the not-batched version of the algorithm. It can be observed, that the batch sizes 2, 16, and 32 have the best speed. In the end, the best enhancement

doi: 10.2313/NET-2017-05-1_03

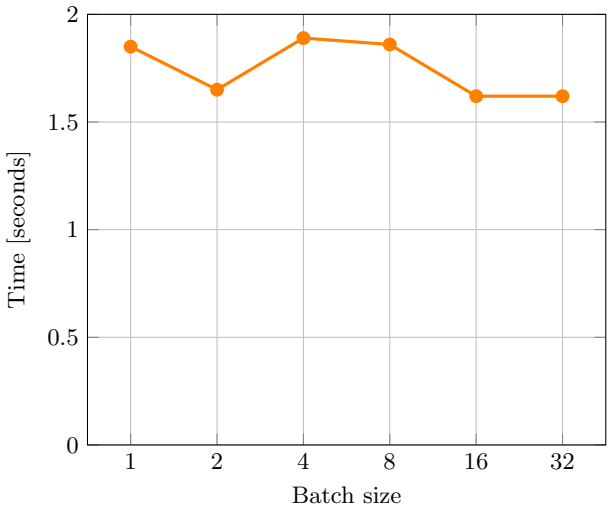


Figure 7: Performance of the basic trie

still is to enable the compiler optimization, which produced a speedup of factor 3.1.

7. CONCLUSION

Various algorithms for routing lookups and their respective data structures are described in this paper.

Prominently tries are used for real-world software routing, while the fastest known algorithm is also based on a trie. Multiple compression techniques exist such as path compression and level compression, which can help to reduce the size of the trie. Caches can be used more efficiently this way, since the memory accesses are cheaper, if the data is already in a low-level cache.

Apart from tries, the DIR-24-8 hardware based algorithm, and the DXR algorithm are presented. Both utilize multiple tables in order to split the IP address into two parts and use the content of the tables to access a next hop table.

An experiment based upon a naive lookup scheme and a basic trie implementation was performed. The result showed, that batching was of no significance for the naive algorithm, and of marginal use for the trie.

8. REFERENCES

[1] J. Postel. (1981, September) Internet protocol. [Online]. Available: https://tools.ietf.org/html/rfc791

[2] L. Rizzo, "netmap: a novel framework for fast packet I/O," in *USENIX Annual Technical Conference*, 2012.

[3] DPDK Developers. (2016, September). [Online]. Available: http://dpdk.org/

[4] Intel, *DPDK Programmer's Guide*, version: 16.04.

[5] J. Thompson, "pfSense around the world, better IPsec, tryforward and netmap-fwd," October 2015. [Online]. Available: https://blog.pfsense.org/?p=1866

[6] P. Gupta, S. Lin, and N. McKeown, "Routing lookups in hardware at memory access speeds," in *INFOCOM'98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 1998, pp. 1240–1247.

[7] S. Nilsson and M. Tikkanen, "An experimental study of compression methods for dynamic tries," *Algorithmica*, vol. 33, no. 1, pp. 19–33, 2002.

[8] Linux Kernel Developers. (2016, October) Version: 4.7.2. [Online]. Available: https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.7.2.tar.xz

[9] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[10] M. Zec, L. Rizzo, and M. Mikuc, "Dxr: towards a billion routing lookups per second in software," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 5, pp. 29–36, 2012.

[11] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 57–70.

[12] FreeBSD Developers. FreeBSD src tree. Version: 9a3c 17b7 80af 5d03 15d0 d362 a209 8484 a37b a0ef. [Online]. Available: https://github.com/freebsd/freebsd

Comparing OpenOnload: A High-Speed Packet IO Framework

Ulrich Huber
Betreuer: Daniel Raumer, Paul Emmerich
Seminar Innovative Internet-Technologien und Mobilkommunikation (WS1617)
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: huberu@in.tum.de, {raumer | emmericp}@net.in.tum.de

KURZFASSUNG

In dieser Arbeit wird das Framework *OpenOnload* [1] von Solarflare, als Alternative zu Frameworks wie *netmap* [2] und *DPDK* [3], untersucht und mit diesen verglichen. Als Vergleichskriterien werden dabei Performance, Stabilität und Nutzung herangezogen. Dabei zeigt sich, dass *OpenOnload* eine ausgereifte Alternative zu *netmap* und *DPDK* darstellt. Besonders die Transparenz, mit welcher sich *OpenOnload* in das Betriebssystem einfügt, übertrifft die anderen Frameworks. Allerdings bestehen an mehreren Stellen noch Verbesserungsmöglichkeiten.

1. EINLEITUNG

Die Datenmenge, welche jährlich die Infrastruktur des Internets belastet, übersteigt Ende 2016 bereits 13 Exabyte [4], mit stark steigender Tendenz. Aufgrund der enormen Belastung der Infrastruktur vom Versand bis zum Empfang von Paketen, ist es immer bedeutender, effiziente und performante Lösungen für die Paketübertragung im TCP-/ UDP-Protokoll zu finden.
Unter dem Gesichtspunkt, effiziente Softwarelösungen für Routing, Überwachung und zur Bereitstellung von Inhalten [5] zu erreichen, hielten auch General Purpose Betriebssysteme den Einzug in die Welt der Paketverarbeitung [2]. Diese sind allerdings nicht für High-Speed Paketnetzwerke ausgelegt, da der Netzwerkstack nicht ausreichend optimiert ist [6]. Um diesen Engpass des Betriebssystems zu umgehen, entwickelten sich sogenannte High-Speed Packet IO Frameworks [2].

Im Weiteren werden die thematischen Grundlagen zum Verständnis von High-Speed Paket IO Frameworks erläutert. Anschließend wird das Framework *OpenOnload* theoretisch beleuchtet. Besonderes Augenmerk liegt dabei auf der Funktionsweise sowie den verwendeten Datenstrukturen. Weiterhin wird auf Einschränkungen des Frameworks eingegangen. Darauf folgend beschäftigt sich diese Arbeit mit einem Vergleich zwischen *OpenOnload* sowie weiteren Frameworks.
Darunter befinden sich *netmap*, als ein einfach verwendbares Framework, und *DPDK* als umfangreichere Lösung. Auf die jeweiligen Besonderheiten dieser Frameworks wird an geeigneter Stelle eingegangen. Der Vergleich in Kapitel 4 bezieht sich auf die drei Rubriken Performance, Stabilität und Nutzung. Im Verlauf dieses Vergleichs wird für jedes Framework ein Codebeispiel gezeigt und erläutert. Diese Ausschnitte sollen dem Leser als eine Möglichkeit von vielen zur Nutzung der Frameworks dienen.

2. GRUNDLAGEN

Um die Vorgehensweise von *OpenOnload* besser verstehen zu können, ist es Anfangs sinnvoll die am weitesten verbreiteten Schwächen, der Implementierung zur Verwaltung von Netzwerkaufgaben, in verschiedenen Systemkernel[1] zu analysieren und mögliche Lösungen zu finden.

Einer dieser Schwachpunkte liegt in der Nutzung von Interrupts, um die Ankunft eines neuen Paketes im Puffer des Netzwerkadapters an den Kernel zu melden. Der Interrupt unterbricht dabei sämtliche aktuell ausgeführte Arbeit, um die Routine zur Aktualisierung des Netzwerkstacks auszuführen. Hierzu führt er einen aufwendigen Kontextwechsel aus, welcher vielen tausenden Operationen entspricht und damit sehr zeitintensiv ist [7].
Zu dem Zeitpunkt der Entwicklung dieser Vorgehensweise war dies ein sinnvoller Ansatz. Da auf einer CPU mit einem Rechenkern alle Ressourcen durch die Verarbeitung des Pakets belegt werden und der Empfang eines Pakets ein zeitkritisches Event ist, soll dieses möglichst ohne Aufschub bearbeitet werden.
In modernen Systemen mit Mehrkern-CPUs ist dieser Ansatz allerdings nicht länger sinnvoll, sofern eine große Datenrate auf einem Netzwerkadapter zu erwarten ist. Denn durch die ständigen Interrupts und zeitintensiven Wechsel in den Kernelspace wird die eigentliche Verarbeitung der Pakete ständig unterbrochen, was schlussendlich zu einem Livelock führen kann [2].
Um diesen Zustand bei hohen Datenraten zu verhindern, gilt es die zeit- und rechenintensiven Aufgaben wie Interrupts und Kontextwechsel möglichst zu minimieren. So wäre eine Lösung, dass der Netzwerkadapter die Ankunft eines neuen Pakets nicht dem Kernel durch einen Interrupt meldet, sondern dieses in einen Empfangsring speichert. Die CPU kann nun je nach Belastung die Pakete aus diesem Ring auslesen und verarbeiten. Sofern der Ringpuffer vollständig gefüllt ist, werden neu ankommende Pakete vom Netzweradapter verworfen. Dies geschieht wennder Puffer vom Netzwerkadapter schneller gefüllt wird, als er von der CPU geleert werden kann. Dieser NAPI genannte Ansatz, wurde auch im Linux Kernel 2.4.20 eingeführt, um Interrupts in Situationen mit hoher Empfangsrate zu verringern [8]. Dadurch ist ein kontinuierliches und synchrones Empfangen und Verarbeiten der Pakete möglich, ohne häufige Interrupts, und damit einhergehende Kontextwechsel zu benötigen.

[1] Die einzelnen Verbesserungsmöglichkeiten sind [6] entnommen und weiter ausgeführt worden.

doi: 10.2313/NET-2017-05-1_04

Weiteres Einsparpotential liegt in der Speicherung empfangener Pakete. Wenn man den Weg eines Pakets vom Empfangszeitpunkt bis zur vollständigen Verarbeitung verfolgt, wird ein Paket mehrmalig im Speicher verschoben. So wird es Anfangs vom Netzwerkadapter in seinem Puffer zwischengespeichert, bis der Kernel es ausliest und im Arbeitsspeicher im Netzwerkstack des Kernels speichert. Eine Anwendung welche das Paket verarbeiten soll, wird unter normalen Bedingungen im Userspace des Betriebssystems ausgeführt und hat daher keinen Zugriff auf den Netzwerkstack des Kernels. Ein weiterer Verschiebungsvorgang in den Userspace findet daher statt. Diese vielen Vorgänge kosten zusätzliche Zeit und Ressourcen, was die Verarbeitung der Pakete verlangsamt und das System zusätzlich belastet. Indem man diese Vorgänge minimiert, kann man eine Beschleunigung und damit einen größeren Durchsatz von Paketen erreichen. So bietet sich an, dass der Netzwerkadapter per DMA[2] die empfangenen Pakete sofort in einem Puffer im Arbeitsspeicher ablegt, welcher zwischen Kernel- und Userspace per Memory-Mapping geteilt wird [2].

Betrachtet man die Speicherverwaltung genauer, kann man eine weitere Einsparung vornehmen. Neben Kopiervorgängen und Interrupts sind Allokationen von Arbeitsspeicher ebenfalls sehr zeitintensiv, da auch sie einen Kontextwechsel benötigen. Indem man den benötigten Speicherplatz für die Puffer zu Beginn eines Programms alloziert und auf weitere Allokationen oder Deallokationen im Programmverlauf verzichtet, kann der Ablauf weiter beschleunigt werden.

Zusätzlich zu diesen Möglichkeiten bietet sich noch die simultane Verarbeitung mehrerer Pakete an. Der sogenannte Batch-Betrieb ist bedeutend effizienter, da er Paketverarbeitung mit der Aktualisierung der Puffer durch den Netzwerkadapter parallelisiert. Dadurch entstehen keine Wartezeiten für eben jene Auffrischung der Daten im Speicher. Gleichzeitig werden Systemaufrufe und Sperrungen von Puffern minimiert[9]. Dadurch werden Ressourcen gespart, und insgesamt gesehen die Verarbeitung beschleunigt.

3. OPENONLOAD

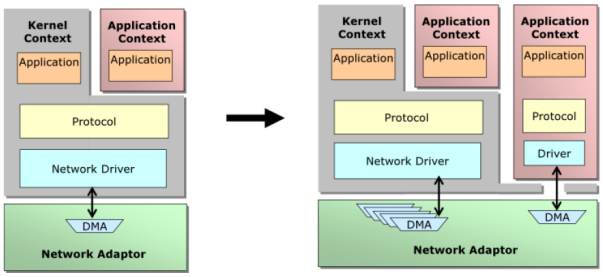


Abbildung 1: Zugriffsmöglichkeiten auf den Netzwerkadapter bei Verwendung von OpenOnload [10]

Das Framework *OpenOnload* wird von Solarflare seit über acht Jahren entwickelt[11] und kostenlos unter der GPLv2-Lizenz zur Verfügung gestellt. Es bietet eine beschleunigte Verarbeitung des TCP-, UDP- und IP-Protokolls unter Linux, ohne größere Änderungen an der nutzenden Software zu fordern, indem unter Anderem auch die POSIX-API implementiert wird. Das Hauptziel der Middleware liegt somit in der Beschleunigung des Netzwerkstacks, ohne die Programmierung zu erschweren und der Wahrung vollständiger Transparenz im Betriebssystem. Um diese Ziele zu erreichen, wird für beschleunigte Applikationen die gesamte Datenebene im Usermode verarbeitet und der Kernel umgangen. Dagegen wird für normale Applikationen weiterhin der Netzwerkstack des Kernels zur Verfügung gestellt. Dies geschieht durch einen sogenannten Hybrid-Stack [10]. Trotzdem bietet *OpenOnload* weiterhin die gleichen Vorzüge wie der Kernel, und senkt weder das Sicherheitslevel, noch schmälert es die Multiplexing-Möglichkeiten [1].

Wie Abbildung 1 zeigt, fügt sich *OpenOnload* dadurch möglichst transparent in das Betriebssystem ein und bietet die Möglichkeit, beschleunigte Applikationen synchron zur normalen Nutzung des Kernel-Netzwerkstacks zu unterstützen.

Die folgende, genauere Beschreibung von *OpenOnload* stützt sich auf das Einführungs-Dokument [10] des Frameworks, sowie auf die Benutzeranleitung für *OpenOnload* [7] von Solarflare.

3.1 Funktionsweise

Wie obig bereits erwähnt bietet *OpenOnload* die POSIX-API an und benötigt daher keine größere Neuprogrammierung von Software, welche dieses Framework nutzen will. Um diese Transparenz zu ermöglichen, nutzt *OpenOnload* einen Hybrid-Stack. Die Middleware kann dabei dynamisch zwischen der Verarbeitung im Kernelspace und Userspace wechseln, und jederzeit die beste Funktionalität bieten.

OpenOnload ist eine passive Bibliothek, was eine Applikation weder an einen Gestaltungsrahmen bindet, noch eine bestimmte Programmiersprache voraussetzt. Zusätzlich bietet das Framework verschiedene Operations-Modi an. Der sogenannte *lazy-receive* Modus findet direkt im Kontext der jeweiligen ausführenden Applikation statt. Dadurch entstehen geringe Overheads und die Bibliothek beginnt erst mit der Verarbeitung von Paketen, wenn der nutzende Thread der Applikation aktiv wird. Durch diesen Aufbau entsteht eine zeitliche und räumliche Lokalität, welche durch Caching unterstützt, einen Performancevorteil bietet.

Ein weiterer Modus ist der *asynchrone Betrieb*, für die Verarbeitung von Paketen, in Threads welche für längere Zeit unterbrochen werden oder einer Applikation welche frühzeitig beendet wird. So gilt es bei asynchroner Verarbeitung der Pakete, die robuste und zeitliche Einhaltung des verwendeten Verbindungsprotokolls zu wahren, was im Kernelspace problemlos möglich ist. Eine reine Userspace-Implementierung könnte diese Funktionalität nicht zur Verfügung stellen. Hier würde bei einer Beendung oder einem Absturz der Applikation der Netzwerkstack mit den Applikationsdaten aus dem Speicher gelöscht werden, wobei jeglicher gespeicherte Zustand verloren gehen würde. Eine sinnvolle Weiterführung der Netzwerkaktivitäten wäre dadurch unmöglich.

Einen vergleichbaren Vorteil bietet der Hybrid-Stack von *OpenOnload* bei Anwendungen mit deutlich mehr Threads, als CPU-Kerne vorhanden sind. Durch den dabei entstehenden großen Scheduling-Aufwand, ist es sinnvoll gewisse Ar-

[2]Auf DMA wird in Kapitel 3.2 weiter eingegangen.

doi: 10.2313/NET-2017-05-1_04

beiten im Hintergrund, beziehungsweise im Kernel auszuführen. Besonders die Bereitstellung der Paketdaten durch einen Dateideskriptor ist eine Aufgabe, welche unabhängig von Threads und zeitnah passieren sollte. Daher ist eine Verarbeitung im Kernelmodul des Frameworks deutlich sinnvoller als in der Bibliothek des Frameworks im Userspace, da eine deutlich geringere Latenz erreicht werden kann.

Durch Memory-Mapping zwischen dem Kernelmodul und der Userspace-Bibliothek[3] ist *OpenOnload* im Stande, den Status des Protokolls, eines genutzten Sockets, direkt aus dem Userspace zu beeinflussen. Diese Aufgabe durch einen Systemaufruf dem Kernel zu überlassen, ist deutlich weniger performant. Des Weiteren bietet sich dadurch die Möglichkeit, die Stacks der durch *OpenOnload* unterstützten Netzwerkadapter und der nicht unterstützten Adapter zu vereinen. Dadurch fügt sich eine weitere Schicht der Transparenz hinzu. *OpenOnload* führt dies vollständig autonom im Hintergrund aus und abstrahiert dabei beschleunigte Sockets um weiterhin eine Verwaltung durch den Kernel zu ermöglichen. Dadurch kann jede Benachrichtigung des Kernels über Events, welche den Netzwerkadapter betreffen, vom Framework empfangen werden. Zusätzlich werden diese Events über ein schreibgeschütztes Memory-Mapping im Userspace zur Verfügung gestellt.

Durch all diese Optimierungen werden Systemaufrufe so weit wie möglich, ohne die Transparenz zu beeinflussen, minimiert.

3.2 Datenstrukturen

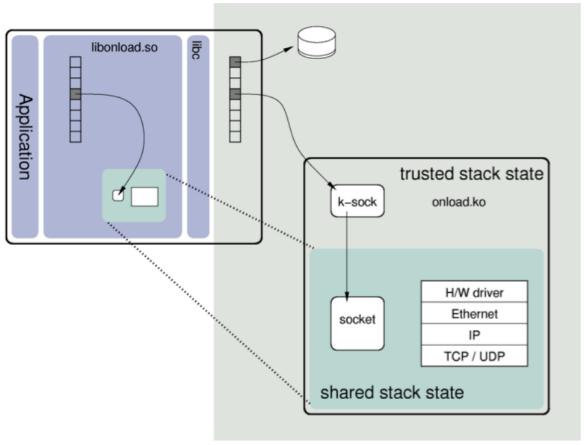


Abbildung 2: Vorgeschaltete Dateideskriptortabelle von OpenOnload mit zugrundeliegendem Netzwerkstack [10]

Durch den Hybrid-Stack von *OpenOnload* ist die zugrundeliegende Datenstruktur, wie sie in Abbildung 2 gezeigt wird, aufwendiger als die anderer High-Speed Packet IO Frameworks. Weiter verkompliziert wird diese Struktur durch die hohe Transparenz die *OpenOnload* bietet. Besonders die Möglichkeit synchron, durch *OpenOnload* beschleunigte und durch den Kernel verwaltete Applikationen zu unterstützen, verkompliziert das Framework.

Die POSIX-API, welche *OpenOnload* implementiert, basiert auf Dateideskriptoren, welche in der Dateideskriptortabelle verwaltet werden. Wobei die Dateideskriptoren, welche für den Netzwerkverkehr von Bedeutung sind, auf den Netzwerkstack verweisen. Dieser hält Informationen wie geöffnete Verbindungen, zugehörige Ports und Sockets, sowie Puffer für den Empfang und den Versand von Paketen bereit. Jedes Betriebssystem besitzt im Kernel einen solchen Netzwerkstack[4], welcher den gesamten Netzwerkverkehr für die Maschine verwaltet. *OpenOnload* bietet neben diesem vom System verwalteten Stack eigene Netzwerkstacks an, um die Verarbeitung zu beschleunigen. Dabei wird jedem beschleunigten Prozess ein eigener Stack zur Verfügung gestellt, wobei je nach Nutzerwunsch ein Stack auch von mehreren Applikationen geteilt werden kann, oder ein Prozess mehrere Stacks besitzen kann.
Um den gewünschten Performanceschub zu erreichen, während die Transparenz weiterhin gewahrt bleibt, schaltet sich *OpenOnload* vor die Dateideskriptortabelle. Durch diese Tabelle ist das Framework in der Lage zu entscheiden, ob ein Dateideskriptor durch die Bibliothek behandelt werden kann oder ob dieser an den Kernel weitergeleitet werden soll. Dateideskriptoren können dabei einer von drei Varianten angehören:

- Sie können lediglich auf dem Kernelstack beruhen, was einem Paketempfang/-versand über einen, von *OpenOnload* nicht unterstützten, Netzwerkadapter gleicht.
- Sie können nur auf dem Stack von *OpenOnload* verweisen, was impliziert, dass die Pakete über einen unterstützen Adapter geroutet werden.
- Sie können einen gemischten Verweis auf den Kernelstack sowie den Stack von *OpenOnload* darstellen, was zum Beispiel bei einem Bonding von unterstützen und nicht unterstützten Netzwerkadaptern der Fall ist.

Die erste Variante wird an den Kernel weitergereicht, da nur dieser sie Verwalten kann. Die Zweite wird vollständig durch das Framework behandelt und die Dritte wird in einem hybriden Modus durch Kernel und Framework verwaltet. Denn beim Auftreten gemischter Dateideskriptoren ist keine alleinige Verarbeitung im Userspace mehr möglich. Dies liegt an dem Umstand, dass Teile der Informationen der Sockets und deren Pakete nur durch den Kernel zur Verfügung gestellt werden können. Hier versucht *OpenOnload* möglichst performant zu Handeln und verbindet eine Active-Waiting Methodik auf Seite der im Userspace verwaltbaren Sockets mit periodischem Abfragen der durch den Kernelstack verwalteten Sockets.

Im Kernel ist des Weiteren ein realer Socket dem beschleunigten Socket zugeordnet, was es *OpenOnload* ermöglicht, weitere Ressourcen wie Ports anzufordern. Durch den Ansatz, den Kernelstack nicht vollkommen zu ersetzen, ist es *OpenOnload* möglich die gesamte POSIX-API vollständig und korrekt zu implementieren.

[3] Auf den genaueren Aufbau der Datenstrukturen wird im nächsten Unterkapitel genauer eingegangen.

[4] Im Weiteren Kernelstack genannt.

doi: 10.2313/NET-2017-05-1_04

Durch die Verwaltung des Protokoll-Status im Kernel wird so auch zum Beispiel `fork()` und `exec()` korrekt implementiert. Die Funktion `fork()` dupliziert eine Applikation und `exec()` ersetzt den Kontext eines Prozesses durch einen Neuen und verliert dabei jegliche im Userspace gespeicherten Zustände [12].

Für ein rein im Userspace agierendes Framework, würde dies den unwiderruflichen Verlust, aller auf den Netzwerkadapter zeigenden Dateideskriptoren, bedeuten. *OpenOnload* kann durch den hybriden Aufbau diese Daten aus dem Kernelstack wiederherstellen und somit das Verhalten der POSIX-Funktionen komplett umsetzen. Erreicht wird dies durch ein selektives Memory-Mapping vom Kernel in den Userspace um die Dateideskriptoren des Frameworks wieder zur Verfügung zu stellen.

Nach einem Aufruf von `exec()` und einer subsequenten neuen dynamischen Verlinkung mit der Userspace Bibliothek von *OpenOnload* besteht die Möglichkeit, durch einen Aufruf von `stat()` fehlende Memory-Mappings bezüglich eines geteilten Dateideskriptors wiederherzustellen. Dadurch ist es in *OpenOnload* auch möglich, dass mehrere Applikationen sich einen Socket im Userspace teilen. Da dies allerdings laut den Entwicklern zu Problemen führen kann, ist das Standardverhalten von *OpenOnload* in diesem Fall, die Verarbeitung von Paketen für diesen Socket vom Framework in den Kernel zu verlagern.

Von *OpenOnload* unterstützte Netzwerkadapter nutzen in Kombination mit dem Framework das Feature *Direct Memory Access* (DMA) [2]. Wie in der Einleitung bereits erwähnt, werden dadurch die Kopiervorgänge deutlich minimiert. Dies geschieht durch das sofortige Ablegen der eingehenden Pakete in den bereitgestellten Puffer im Arbeitsspeicher. Dies geschieht durch den Netzwerkadapter, wobei dieser seinen internen Puffer ignoriert. Gleichzeitig birgt dieses Feature aber auch Risiken, da der Netzwerkadapter unkontrolliert in willkürlichen Adressen des Arbeitsspeichers schreiben könnte. Für diese Sicherheitslücke gibt es allerdings auf Hardwareebene Beschränkungsmechanismen. Sogenannte I/O Memory Management Units (IOMMUs) bieten die Möglichkeit eines Zugriffsschutzes bei DMA [2]. Diese Bausteine sind mittlerweile in allen neueren CPU-Generationen von Intel [13] und AMD [14] verbaut und DMA stellt kein Risiko mehr dar. Da OpenOnload für jede Anwendung einen eigenen virtuellen Netzwerkstack nutzt, besteht auch nicht die Möglichkeit, dass Anwendungen die Pakete anderer Applikationen auslesen oder verändern. Sie erhalten lediglich Zugriff auf ihre eignen Pakete, was den Möglichkeiten bei Nutzung des Kernelstacks entspricht.

3.3 Einschränkungen

Eine der größten Einschränkungen von *OpenOnload* liegt in der Limitierung der Netzwerkadapter. Hier beschränkt sich das Framework auf Interfaces von Solarflare, also eben jenem Entwickler von *OpenOnload*. Durch diese Reglementierung ist *OpenOnload* gegenüber den anderen Frameworks e.g. *netmap*, *DPDK* unterlegen. Denn für diese Frameworks gibt es eine breite Basis an Treibern für die verschiedensten Hersteller wie Intel, RealTek oder nvidia [2]. Zusätzlich arbeitet auch Solarflare für ihre Netzwerkadapter an einem Treiber für DPDK [15].

Da es in *OpenOnload* prinzipiell möglich ist, einen Stack per Semaphor zwischen mehreren Prozessen zu teilen, besteht grundlegend die Möglichkeit einer Verklemmung. Speziell bei der unsauberen Terminierung eines Prozesses besteht diese Gefahr. Beispielsweise beendet der Befehl `exit()` alle Threads eines Prozesses und diesen schließlich auch, ohne darauf zu achten ob sich die Threads gerade in einem kritischen Bereich befinden und eine Ressource sperren. Um aus diesen Zustand aufzulösen, setzt *OpenOnload* sämtliche TCP-Verbindungen zurück. Dies kann andere Anwendungen, die diese Ressource ebenfalls nutzen, beeinflussen.
Mit dieser Einschränkung verbunden, ist auch der Hinweis in der Benutzeranleitung von *OpenOnload*, dass man den Befehl `pthread_cancel()` nicht nutzen soll, da dieser zu unvorhersehbarem Verhalten führt.

Durch den Aufbau von *OpenOnload* ist der Zugriff auf Pakete durch Packet-Capturing Software wie `tcpdump` eingeschränkt. Diese Art von Software liest den Kernelstack aus und ignoriert den Stack von *OpenOnload* im Userspace. Ein vergleichbares Problem besteht auch mit Firewalls wie `iptables`, da diese ebenfalls auf dem Kernelstack basieren. Des Weiteren besteht für Systemtools wie `stackdump` die Möglichkeit, dass von *OpenOnload* beschleunigte Sockets nicht erkannt werden, da diese im Verzeichnis `/proc` Symlinks auf `/dev/onload` bilden. Solarflare bietet für diese Zwecke jedoch eigene Software. Zusammengefasst gilt theoretisch für jede Software die auf dem Auslesen des Verzeichnisses `/proc` basiert, dass von *OpenOnload* beschleunigte Sockets nicht erkannt werden.

OpenOnload bindet seine Funktionen durch `LD_PRELOAD` in einer Applikation ein. Um dies zu unterstützen, muss die Bibliothek dynamisch gelinkt werden und darf nicht statisch eingebunden werden. Wird die Bibliothek statisch eingebunden, wird die Applikation nicht beschleunigt und jeglicher Netzwerkverkehr wird durch den Kernel verarbeitet.

Weiter besteht eine Einschränkung *OpenOnloads* darin, dass Richtlinien-basiertes Routing nicht unterstützt wird. Das Framework sendet grundsätzlich auf einer beliebigen validen Route ein Paket.

Es sei noch erwähnt, dass es zu einem Timeout einer TCP-Verbindung kommen kann, sofern ein Thread einer beschleunigten Applikation ein `SIGSTOP` Signal erhält und in diesem Moment den Stack reserviert hat. `SIGSTOP` dient dem Stoppen eines Threads, ohne Rücksicht auf Timeout-Grenzen des TCP-Protokolls, beziehungsweise der Serversoftware des Verbindungspartners. Somit liegt die Ursache nicht bei *OpenOnload*, sondern am grundsätzlichen Sinn von `SIGSTOP`. Daher kann sie nicht als wirkliche Einschränkung von *OpenOnload* gesehen werden. Da die Nutzung von `SIGSTOP` allerdings zu den Standardwerkzeugen im Debugging zählt, ist es für diese Auflistung relevant.

OpenOnload ist auch in den Möglichkeiten der Beschleunigung beschränkt. So unterstützt es weder Beschleunigung von fragmentierten Paketen auf der Empfangsseite noch das Senden von Broadcasts. Zusätzlich wird IPv6 nicht beschleunigt, kann allerdings weiterhin transparent durch den Kernel verarbeitet werden.
Die Beschleunigung beschränkt sich außerdem auf TCP-,

doi: 10.2313/NET-2017-05-1_04

UDP- und IP-Sockets. Weiter werden über `socketpair()` erstellte Sockets oder über `sendfile()` versendete Daten nicht beschleunigt.

Über Link-Aggregation verbundene Netzwerkadapter müssen ausschließlich unterstützte Adapter von Solarflare sein, um eine Beschleunigung der Verbindung zu ermöglichen. Wird ein nicht unterstützter Adapter verwendet, kann dies zu unvorhersehbarem Verhalten führen, sofern bereits davor eine beschleunigte Verbindung auf einem der anderen Adapter aufgebaut war. War eine solche Verbindung nicht bereits aufgebaut, werden alle neuen Verbindungen transparent durch den Kernel verwaltet.

Weitere Einschränkungen sind in der Benutzeranleitung von OpenOnload [7] im Kapitel 11 zu lesen.

4. VERGLEICH MIT ANDEREN FRAMEWORKS

4.1 Performance

Die Performance ist eines der wichtigsten Entscheidungskriterien für oder gegen ein High-Speed Packet IO Framework. Um eine Aussage über OpenOnload zu treffen, ist daher ein Vergleich in dieser Kategorie nötig. Hierzu werden bereits bestehende Daten über die Performance der Frameworks *OpenOnload*, *netmap* und *DPDK* herangezogen und als Basis für eine Bewertung von *OpenOnload* verwendet. Außerdem wird *OpenOnload* mit dem Kernel von *Red Hat Enterprise Linux 6.2* verglichen, ein speziell für Unternehmen zugeschnittenes General Purpose Betriebssystem.

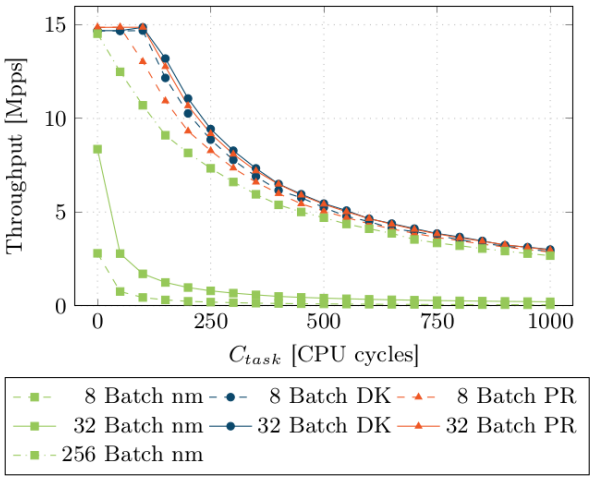


Abbildung 3: Durchsatzrate von *netmap* (nm) und *DPDK* (DK) sowie PF_RING ZC (PR) in Abhängigkeit vom Zeitaufwand pro Paket der nutzenden Applikation [6]

Für *DPDK* und *netmap* existieren bereits gute Vergleiche, wie zum Beispiel die Arbeit von Gallenmüller et al. [6]. Das in der Abbidlung 3 ebenfalls dargestellte *PF_RING* ist ein Framework, welches über einen Ringpuffer die Pakete weiterreicht. Mit verschiedenen Modulen bietet es zum Basisumfang zusätzliche Funktionalitäten, wie *Zero Copy (ZC)* an, wodurch es ebenfalls Paketbeschleunigung ermöglicht [16]. Da es in diesem Vergleich keinen Mehrwert zu den anderen Frameworks bietet, wird es nicht weiter betrachtet.
DPDK ist gegenüber *netmap* deutlich performanter (siehe Abb. 3), da *netmap* auf vielen, zeitintensiven Systemaufrufen beruht, welche *DPDK* umgeht [6]. Besonders deutlich wird dieser Umstand durch den deutlich höheren Durchsatz, wenn die Batchgröße für *netmap* stark erhöht wird. Die Batchgröße bezeichnet hierbei die Anzahl gleichzeitig verarbeiteter Pakete im Batch-Betrieb. Dadurch treten deutlich weniger Systemaufrufe auf und *netmap* erreicht eine mit *DPDK* vergleichbar gute Datenrate.
DPDK arbeitet im Gegensatz zu *netmap* ausschließlich im Userspace und erreicht durch diesen Unterschied auch keine ansatzweise so extreme Beschleunigung durch Erhöhung der Batchgröße wie *netmap*. Eine kleine Optimierung ist trotzdem zu erkennen und beruht auf der besseren Ressourcennutzung [9]. *OpenOnload* agiert gleichzeitig im Kernelspace und im Userpsace. Daher benötigt es wie *netmap* zeitintensive Systemaufrufe, welche allerdings durch den hybriden Aufbau möglichst vermieden werden. Daher würde auch *OpenOnload* vom Batch-Betrieb profitieren, allerdings nicht so extrem wie *netmap*. Da es größtenteils die Verarbeitung im Userspace vornimmt, ist anzunehmen, dass es eine bessere Performanz als *netmap* bei kleinen Batchgrößen zeigt, allerdings nicht besser als *DPDK*. Diese Aussage setzt allerdings voraus, dass *OpenOnload* und *DPDK* Pakete im Userspace gleich performant verarbeiten.

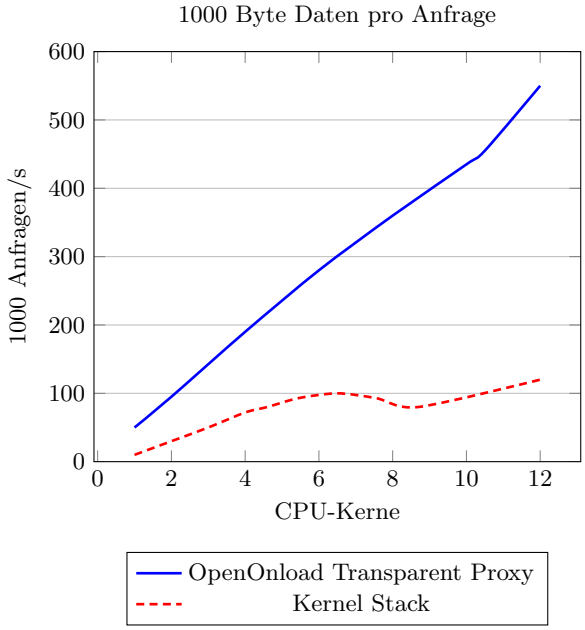


Abbildung 4: Proxybetrieb mit *OpenOnload* und Kernel [17]

Durch die Nutzung von *OpenOnload* vor allem im Serverbereich [18], ist der Großteil der bisherigen Tests für dieses Framework mit CPUs mit einer hohen Anzahl an physischen Kernen durchgeführt worden. Wie in Abbildung 4 gezeigt, gewinnt *OpenOnload* mit der Erhöhung der CPU-Kerne nahezu linear an Performanz. Gegenüber dem Kernelstack ist

doi: 10.2313/NET-2017-05-1_04

dabei ein deutlich schnellerer Anstieg der Geschwindigkeit zu erkennen. Dies spricht für eine, für parallelisierte Zugriffe optimierte Implementierung. Besonders von Vorteil ist hier die gemeinsame Nutzung von Sockets im Userspace (siehe Kapitel 3.2), welche von *OpenOnload* umgesetzt wird.

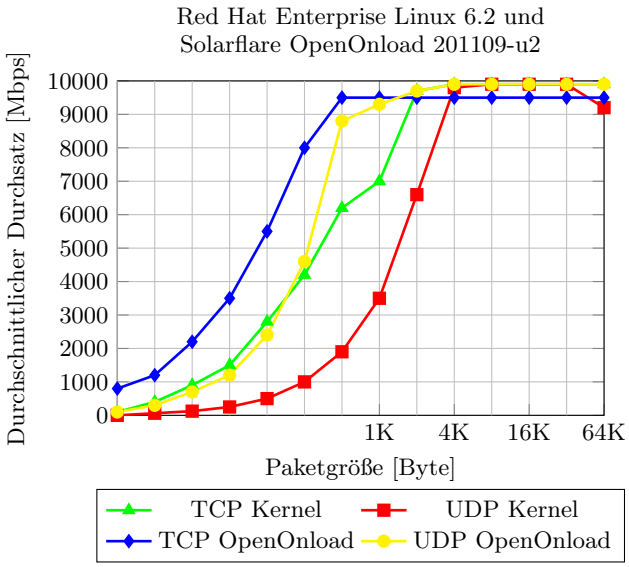


Abbildung 5: *OpenOnload* und Red Hat Kernel [19]

Im Vergleich zum Kernel von *Red Hat Enterprise Linux 6.2*[5] bietet *OpenOnload* auch bereits deutlich früher einen höheren Durchsatz. Wie Gallenmüller et al. [6] schreiben, ist die Belastung des Frameworks, bei gleichbleibendem Durchsatz der Schnittstelle, umso größer, je kleiner der Payload des Pakets ist. Wie in Abbildung 5 erkennbar, bietet *OpenOnload* bereits bei sehr kleinen Paketen eine angemessen hohe Datenrate.
OpenOnload ist besonders auf geringe Latenz und wenig Jitter[6] ausgelegt [19]. Eine Untersuchung der Firma Concurrent hat ergeben, dass die Latenz mit *OpenOnload*, im Vergleich zu Red Hat RHEL 6 um bis zu 20 Mikrosekunden gesenkt werden kann. Gleichzeitig konnte dabei der Jitter von 4 Mikrosekunden auf unter eine Mikrosekunde verringert werden [20].
Zusammengefasst bietet *OpenOnload* vor allem in Bezug auf Latenz einen deutlichen Vorteil gegenüber dem Kernel und ermöglicht die Nutzung von High-Speed Netzwerken mit General Purpose Betriebssystemen bereits bei sehr kleinen Paketgrößen.

4.2 Stabilität

Ein gutes Framework zeichnet sich nicht nur durch Performance aus. Auch die Stabilität ist entscheidend, um ein zuverlässiges System zu gestalten. Daher müssen Frameworks auch mit fehlerhaften Clients umgehen können, um Abstürze des Kernels weitestgehend zu vermeiden. Ziel eines soliden Frameworks sollte sein, die Sicherheit des Kernelspaces nicht zu kompromittieren, und dadurch das System gegenüber Exploits angreifbar zu machen. Außerdem sollten keine Daten, welche nicht im Userspace der Anwendung gehalten werden, verloren gehen, wenn diese abstürzt. Dies ist besonders von Bedeutung, wenn sich mehrere Anwendungen einen Socket teilen.

OpenOnload bietet hier mit dem zweiteiligen Aufbau durch den Hybrid-Stack bereits eine solide Grundstruktur [7]. Da alle Statusinformationen eines Sockets auch im Kernel gespeichert werden, sind diese auch noch nach dem Absturz einer Applikation verfügbar [10]. Weiter ist durch die separaten Stacks pro Anwendung auch die Sicherheit von Paketen anderer Anwendungen tortz Memory-Mapping gesichert. Zusätzlich sind Teile des Memory-Mappings schreibgeschützt, um die Integrität des Kernels zu wahren.
Netmap bietet durch seine Verwendung von Systemaufrufen mit Überprüfungen von Nutzerdaten eine solide Abschirmung vor fehlerhaften Applikationen. Diese Abschirmung geht so weit, dass Luigi Rizzo es in seinem Paper [2] als unmöglich erachtet, einen Absturz des Kernels durch einen fehlerhaften *netmap*-Client hervorzurufen. Da *netmap* vollständig im Kernelspace agiert, kann es wie *OpenOnload* Daten über Anwendungsabstürze hinweg halten.
DPDK ist nach Dominik Scholz ebenso unanfällig durch fehlerhafte Clients Kernelabstürze zu erzeugen [21]. Wegen der rein im Userspace gehaltenen Implementierung verliert das Framework allerdings sämtliche Daten bei einem Absturz der Anwendung.

Wie die Frameworks *DPDK* [3] und *netmap* [2] überprüft *OpenOnload* laufend, ob ein Überlauf seiner Puffer droht, da dies zu einem Deadlock führen könnte [7], [9]. Droht ein solcher Überlauf, leitet es mehrere Gegenmaßnahmen ein, wie zum Beispiel das Verwerfen von ankommenden Paketen durch den Netzwerkadapter. Dadurch kann eine momentane Überlastung verzögert, wenn nicht sogar abgewendet werden.

4.3 Nutzung

Besonders wichtig ist für Programmierer, dass bestehender Code bei der Nutzung einer Bibliothek möglichst wenig verändert werden muss, da dies unnötigen Aufwand darstellt und potentiell Fehler birgt. Daher ist die Verwendung von einer eigenen API in einem High-Speed Packet IO Framework, wie *DPDK* dies praktiziert [3], ein Hindernis. Deutlich besser ist es, wenn eine bestehende, weit verbreitete API genutzt wird. Dies geschieht bei *OpenOnload* und *netmap* durch die Nutzung der POSIX-API. Dadurch wird der Aufwand für Programmierer auf die Verlinkung des Frameworks in seiner Applikation beschränkt und eine einfache Verwendung ermöglicht. Zusätzlich entfällt eine langwierige Einlesephase, die bei einer neuen ungewohnten API nötig wäre.
OpenOnload geht an diesem Punkt noch einen Schritt weiter als *netmap* und bietet die Möglichkeit eine Anwendung lediglich mit dem Aufruf `onload <app_name> [app_options]` zu starten. Hierdurch wird die Anwendung, ohne neu kompiliert werden zu müssen, sofort beschleunigt ausgeführt [7]. Außerdem werden in einer Umgebung alle Anwendungen durch setzen der Umgebungsvariable `LD_PRELOAD=libonload.so` beschleunigt.

[5] Der Red Hat-Kernel implementiert den Funktionsumfang des Linux-Kernel 2.6 wird aber von Red Hat ständig aktualisiert.
[6] Jitter steht für Schwankungen in längeren Messungen eines anderen Wertes (hier Latenz)

doi: 10.2313/NET-2017-05-1_04

Im Folgenden werden für jedes Framework kurze Codebeispiele aufgeführt, die die Verwendung dieser veranschaulichen sollen. In allen Beispielen wurde die Fehlerüberprüfung vernachlässigt, um die Codeausschnitte möglichst kurz zu halten. Jeder Code sendet in einer Dauerschleife Pakete mit Daten aus einem Puffer. Der Puffer muss dabei kontinuierlich durch die Anwendung gefüllt werden.

```
fds.fd = open("/dev/netmap", O_RDWR);
strcpy(nmr.nm_name, "ix0");
ioctl(fds.fd, NIOCREG, &nmr);
p = mmap(0, nmr.memsize, fds.fd);
nifp = NETMAP_IF(p, nmr.offset);
fds.events = POLLOUT;
for (;;) {
  poll(fds, 1, -1);
  for (r = 0; r < nmr.num_queues; r++) {
    ring = NETMAP_TXRING(nifp, r);
    while (ring->avail-- > 0) {
      i = ring->cur;
      buf = NETMAP_BUF(ring, ring->slot[i].buf_index);
      ... store the payload into buf ...
      ring->slot[i].len = ... // set packet length
      ring->cur = NETMAP_NEXT(ring, i);
    }
  }
}
```

Listing 1: Beispiel mit netmap [2]

netmap nutzt als Datenstruktur einen Ringpuffer. Im obigen Ausschnitt wird diese Struktur genutzt, um den mit `NETMAP_TXRING()` erstellten Puffer zu füllen.
Hierzu stellt das Framework mit `NETMAP_BUF()` einen Puffer an einem angegebenen Platz im Ring zur Verfügung, welchen man anschließend mit seinen Daten füllen kann. Nach dem die Daten in den Puffer geschrieben wurden, und *netmap* die Größe der geschriebenen Daten mitgeteilt wurde, wird das Paket durch das Framework versendet. Anschließend kann mit der Funktion `NETMAP_NEXT()` der nächstgelegene Platz im Slot gewählt und der Prozess von Neuem begonnen werden.
In diesem Beispiel wurden vorwiegend Funktionen genutzt, welche nicht der POSIX-API entsprechen, sondern dem Nutzer möglichst viel Komfort bieten. Wie in der ersten Zeile zu sehen ist, nutzen auch diese Funktionen einen Dateideskriptor. Durch diesen Dateideskriptor können auch über die POSIX-API Pakete versandt werden.
netmap unterstützt auch Paketversand ohne Kopiervorgänge, beziehungsweise ohne wiederholtem Allozieren der Puffer. Dies geschieht durch die Wiederverwendung der bereits versendeten Puffer und einer darauf folgenden Zuweisung von `BUF_CHANGED` zu der Variable `ring->flags`. Dies kann auch genutzt werden, um Pakete ohne Kopiervorgang weiterzuleiten. Hierfür werden die Empfangspuffer mit dem Paket durch einen Puffer des Ringpuffers zum Versenden ausgetauscht. Anschließend wird obige Zuweisung ausgeführt und die Adapter senden die aktualisierten Puffer [2].

```
struct rte_mempool *mbuf_pool;
int ret = rte_eal_init(argc, argv);
mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", 32,
  MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
  rte_socket_id());
struct rte_eth_conf port_conf = port_conf_default;
rte_eth_dev_configure(0, 0, 1, &port_conf);
rte_eth_tx_queue_setup(0, 0, 512, rte_eth_dev_socket_id(0),
  NULL);
rte_eth_dev_start(0);
for (;;) {
  struct rte_mbuf *bufs[32];
  for (int i = 0; i < 32; i++)
    bufs[i] = rte_pktmbuf_alloc(mbuf_pool);
  ... store the payloads into bufs ...
  rte_eth_tx_burst(0, 0, bufs, 32);
}
```

Listing 2: Beispiel mit DPDK

DPDK bietet eine API, die mit wenigen Zeilen Code Pakete im Batch-Modus versenden kann. Im Beispiel verschickt das Framework gleichzeitig 32 Pakete, welche aus dem Puffer-Array `bufs` gelesen werden. Bevor der Sendevorgang beginnen kann, wird Anfangs das Framework initialisiert. Dies geschieht mit einem Aufruf von `rte_eal_init()`. Die für den Versand benötigten Puffer, werden durch das Framework in einem Puffer-Pool bereitgehalten. Dieser Pool wird mit `rte_pktmbuf_pool_create()` erstellt und später durch `rte_pktmbuf_alloc()` mit Puffern gefüllt. Nach der Erstellung des Pools wird für den Sendeprozess ein Netzwerkadapter mit der Funktion `rte_eth_dev_configure()` konfiguriert. Im Gegensatz zu *netmap* nutzt *DPDK* keinen Ringpuffer sondern eine Queue zum Versenden von Paketen. Diese Queue wird mit `rte_eth_tx_queue_setup()` initialisiert. Um die Vorbereitungen abzuschließen, muss der vorher konfigurierte Adapter mit `rte_eth_dev_start()` noch gestartet werden. Ab diesem Zeitpunkt ist ein Paketversand mit dem Kommando `rte_eth_tx_burst()` möglich.

```
struct onload_zc_iovec iovec;
struct onload_zc_mmsg mmsg;
onload_zc_alloc_buffers(fd, iovec, 1,
  ONLOAD_ZC_BUFFER_HDR_TCP);
mmsg.fd = fd;
mmsg.iov = iovec;
mmsg.msg.msghdr.msg_iovlen = 1;
for (;;) {
  ... store the payload in iovec.iov_base ...
  onload_zc_send(&mmsg, 1, 0);
}
```

Listing 3: Beispiel mit OpenOnload

Abschließend wird ein Beispiel mit *OpenOnload* betrachtet. Wie bei *netmap* wird auch bei diesem Framework darauf verzichtet, die POSIX-API in diesem Beispiel zu nutzen. Allerdings ist auch hier durch den Dateideskriptor `fd`, die Nutzung der API grundsätzlich möglich.
Ziel dieses Codeausschnittes ist es viel mehr, den Teil der API von *OpenOnload* kurz zu präsentieren, welcher dem Nutzer eine erleichterte Handhabung ermöglicht.
Durch Aufruf von `onload_zc_alloc_buffers()` werden dem Nutzer durch das Framework Puffer für Pakete zur Verfügung gestellt. Diese Puffer können durch die Anwendung gefüllt und mit dem Befehl `onload_zc_send()` versendet werden.
Anzumerken ist bei diesem Beispiel die Verwendung der API-Funktionen, welche Paketversand ohne interne Kopiervorgänge ermöglichen. Dies ist signalisiert durch die Buchstabenfolge `zc` in den Funktionsnamen. *OpenOnload* bietet des Weiteren auch die Möglichkeit mit einem Aufruf von `onload_zc_send()` mehrere Pakete an verschiedene Sockets zu senden. Ein Beispiel findet man in der Dokumentation von *OpenOnload* auf Seite 231f [7].

Ebenso wichtig ist neben der Minimierung des Programmieraufwandes für Entwickler die Transparenz im restlichen Umfeld des Betriebssystems. Möglicherweise kann ein Programm nicht passend abgeändert werden, um ein Framework zu nutzen. Wenn eine solche Applikation allerdings parallel zu beschleunigten Anwendungen benötigt wird, kann dies unter Umständen zu Problemen führen. *OpenOnload* bietet hier durch seinen Hybrid-Stack eine elegante Lösung, indem es automatisch den Kernelstack aktuell hält und somit auch nicht beschleunigten Applikationen den Empfang und das Versenden von Paketen über das Netzwerk ermöglicht [10]. *Netmap* und *DPDK* blockieren das Betriebssystem dage-

gen vollständig vom Zugriff auf einen Netzwerkadapter, sobald eine beschleunigte Applikation aktiv wird [6]. Daraus resultiert auch die Blockierung von nicht beschleunigten Anwendungen. Denn diese versuchen über die Funktionen des Betriebssystems auf den Netzwerkadapter zuzugreifen.

5. FAZIT

OpenOnload ist eine ausgereifte Alternative zu den anderen High-Speed Packet IO Frameworks. Die Verwaltung durch einen Hybrid-Stack bringt ein Maß an Transparenz, welches schwer zu übertreffen ist und gleichzeitig eine Sicherheit die anderen Frameworks in nichts nachsteht. Gleichzeitig kann *OpenOnload* in der Performance mit vollständigen Userspace-Frameworks mithalten. Einzig die fehlende Unterstützung von IPv6 und die Bindung an Solarflares Netzwerkadapter schränken den Anwender wirklich ein.

6. LITERATUR

[1] *OpenOnload*. `http://www.openonload.org/`. zuletzt besucht 12.12.2016

[2] Luigi Rizzo. 2012. *Netmap: a novel framework for fast packet I/O*. In Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC'12). USENIX Association, Berkeley, CA, USA, 9-9.

[3] Intel. *Data Plane Development Kit*. `http://www.dpdk.org/doc/guides/index.html`, zuletzt besucht 15.12.2016

[4] *Entwicklung des Datenvolumens im stationären Breitband-Internetverkehr im Festnetz in Deutschland von 2001 bis 2016 (in Millionen Gigabyte pro Jahr)*. `https://de.statista.com/statistik/daten/studie/3565/umfrage/datenvolumen-des-breitband-internetverkehrs-in-deutschland-seit-dem-jahr-2001/`. zuletzt besucht 12.12.2016

[5] *vpp-dev/netmap: Netmap - a framework for fast packet I/O*. `https://github.com/vpp-dev/netmap`. zuletzt besucht 21.12.2016

[6] S.Gallenmüller. Paul Emmerich. Florian Wohlfart. Daniel Raumer. Georg Carle. *Comparison of Frameworks for High-Performance Packet IO*. ANCS 2015. Mai 2015

[7] *Onload User Guide*. `https://support.solarflare.com/index.php/component/cognidox/?file=SF-104474-CD-21_Onload_User_Guide.pdf&task=download&format=raw&id=361`

[8] Jamal Hadi Salim. Januar 2005. *When NAPI Comes To Town*. In Proceedings of the UKUUG 2005 Linux Technical Conference.

[9] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. *Fast Userspace Packet Processing*. In Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems (ANCS '15). IEEE Computer Society, Washington, DC, USA, 5-16.

[10] Steve Pope, PhD, David Riddoch, PhD. *Introduction to OpenOnload-Building Application Transparency and Protocol Conformance into Application Acceleration Middleware*.

[11] Solarflare Communications, Inc. *Solarflare and OpenOnload*. `http://storage.dpie.com/downloads/solarflare/Solarflare_FRnOG_OpenOnload.pdf`

[12] Steve Pope. David Riddoch. Februar 2008. *OpenOnload A user-level network stack*. In Proc. of Google Talk 07.02.2008.

[13] Intel. TW Burger. März 2012. *Intel® Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices*.

[14] AMD. Februar 2015. *AMD I/O Virtualization Technology (IOMMU) Specification*

[15] Andrew Rybchenko. Oktober 2016. *[dpdk-dev] Solarflare PMD submission question*. `http://dev.dpdk.narkive.com/kICEGTdM/dpdk-dev-solarflare-pmd-submission-question#post1`. zuletzt besucht am 21.12.2016.

[16] ntop. *PF_RING*. `http://www.ntop.org/products/packet-capture/pf_ring/` zuletzt besucht 11.02.2017

[17] Solarflare. 2015. *Technology Brief: Application Note-OpenOnload 201509*.

[18] Arista. Solarflare. Bruce Tolley, PhD. *10Gb Ethernet: The Foundation for Low-Latency, Real-Time Financial Services Applications and Other, Latency-Sensitive Applications*

[19] RedHat. 2012. *Solarflare OpenOnload Performance Brief*.

[20] Concurrent. Joe Korty. März 2012. *Comparison of the real-time network performance of RedHawk Linux 6.0 and Red Hat operating systems*.

[21] Dominik Scholz. 2014. *A Look at Intel's Dataplane Development Kit*. In Proc. of Seminar Innovative Internettechnologien und Mobilkommunikation SS 2014.

Source Packet Routing in Networking (SPRING)

Adrian Reuter
Betreuer: M.Sc. Edwin Cordeiro
Seminar Future Internet WS2016
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: adrian.reuter@tum.de

ABSTRACT
This paper analyses the source routing strategy, which is an alternative to the shortest-path-first strategy. It depicts use cases for source routing and provides an insight into a new source routing mechanism currently developed by the IETF SPRING working group. The design requirements identified by the working group as well as the proposed architecture and implementational approaches are explained. The paper further presents alternative source routing solutions and compares them to the SPRING working group solution. The paper concludes with a prospect of challenges that the SPRING solution is likely to face in deployment.

Keywords
Source Routing, Source Packet Routing, Segment Routing, MPLS, SPRING, RPL, DSR, RH0

1. INTRODUCTION
Within internetworking infrastructure, the shortest path - respective to the metric in use - is the most common standard strategy for forwarding decisions. Routers interconnect separate networks and forward traffic from the source to the destination. Every router decides on its own where to steer a packet, according to its Routing Information Base (RIB) and its Forwarding Information Base (FIB). The information stored in these bases are established either manually or via routing protocols. Interior Gateway Protocols (IGP) such as OSPF or IS-IS are used to exchange routing information between routers within an Autonomous System (AS), whereas Exterior Gateway Protocols (EGP) such as BGP are used to communicate routing information between autonomous systems [1].

However, there are multiple scenarios in which a node may wish to determine a specific set of nodes that shall be traversed while delivering a packet to its destination, or even impose an explicit path through the network topology for reaching the destination. The strategy of imposing a partial or entire path on a packet is called *Source Routing* and can be a powerful mean towards efficient and programmable networks [2]. For this reason this paper will depict the manifold use cases for source routing and present source routing solutions that have been implemented or are currently in development.

Sections 1.1 and 1.2 provide an insight into mechanisms and protocols that are crucial for understanding the work in development by the SPRING working group. While section 2 concentrates on the source routing solution that is currently developed by the IETF, section 3 presents alternative solutions originating from academic research, standardisation organisations or industrial development.

1.1 IPv6 Extension Headers
The Internet Protocol Version 6 (IPv6) is located on the network layer (layer 3) of the ISO/OSI model and offers logical end-to-end addressing for network communication. Besides the standard header fields that most notably include source and destination address, IPv6 provides the ability to attach additional information to IP packets by so-called extension headers. Extension headers enable extended and optional functionalities for IP packets, such as fragmentation, routing options, authentication or encrypted encapsulation [3]. They consist of a 'Next Header' field indicating the subsequent header type, a 'Hdr Ext Len' field defining the length of the extension header, and varying header-specific data [4]. Extension headers immediately follow the IPv6 standard header and are announced by the 'Next Header' field of the preceding header. That means the standard header might announce an extension header in its 'Next Header' field, while the extension header announces another extension header (or a layer 4 protocol header) in its own 'Next Header' field [3].

1.2 Multiprotocol Label Switching (MPLS)
Conventional routers make their forwarding decisions according to a longest prefix match. This results in each and every router along the path to the destination deciding on its own where to route a packet; choosing the longest match between the destination IP address of the incoming packet and the network addresses with their prefix lengths stored in the Routing Information Base [1]. Multiprotocol Label Switching (MPLS) is used mostly in backbone networks, service provider networks or huge company networks, and enables routers within a MPLS-domain to forward packets only according to a prepended *label*. Such a label specifies the affiliation of a packet to a *Forwarding Equivalence Class (FEC)*, which is defined by RFC 3031 [5] as *"a group of IP packets which are forwarded in the same manner (e.g., over the same path, with the same forwarding treatment)"*.

A MPLS-enabled router at the border of a MPLS-domain analyzes incoming IP datagrams and assigns them to a FEC by prepending a 20-bit label. As consequence the network layer protocol and its addressing scheme is only analyzed once, that is when entering the MPLS-domain [5]. The la-

doi: 10.2313/NET-2017-05-1_05

bel is most commonly prepended with the help of a so-called *shim header*, a small header inserted between the layer 2 and layer 3 headers [5]. Other MPLS routers within the domain will forward the packet based on the label. They can also add further labels to the packet and thus create a label stack. Label meanings are exchanged between routers by dedicated protocols such as LDP and RSVP, or extensions to other protocols such as BGP [5].

2. SPRING WORKING GROUP

In order to advance the research and standardization of a flexible and universal source routing mechanism, the Internet Engineering Task Force (IETF) has formed a working group (WG) in 2013. This WG, called *Source Packet Routing in Networking (SPRING)*, is chartered to identify source routing use cases as well as defining the requirements and mechanisms for implementing, deploying and administrating source routing enabled networks [6]. The working group yet developed a new source routing mechanism called *segment routing* [7], which is discussed in section 2.3. It further introduced two implementational approaches [8, 9], which will be discussed in section 2.4 and 2.5. The working group is currently preparing their final document revisions for a technical review and adoption to IETF standards track [10].

2.1 Requirements and Design Goals

The SPRING WG charter [6] defines some fundamental design goals and general requirements for the new source routing mechanism to be worked on. With regard to IPv6 replacing IPv4 in near future, the working group agreed on not taking IPv4 into consideration and developing an IPv6-only based solution [6]. The new source routing mechanism is ought to be downward compatible with existing protocols and layers and should minimize modifications to existing architectures. This is a central requirement for an incremental and selective enrollment of the new mechanism in the context of existing network hardware resources and infrastructures [2]. To be able to traverse non-source-routed network sections, the new source routing solution needs to provide interoperability with conventional non-source-routed networks or subnets [6].

Furthermore intermediate routers shall be able to forward packets based on routing information attached to the packets themselves instead of per-path state information stored at those intermediate routers. That means the path (or a partial path) to be taken to reach the destination is encoded in the packet header, and is not decided by routers on the path. It is important to notice that the node imposing the source-routed path (in the following proceeding denoted as 'source') is not necessarily the originator of a packet, but might also be for example a router at the border of a source-routed domain [6]. After all, the SPRING WG solution must define a basic security concept to encounter common security issues, such as malicious packet injection or traffic amplification. This security concept might be enhanced by additional security mechanisms deployed by the operator, individually fitting the needs [2, 6].

2.2 Source Routing Use Cases

Source routing is a useful technique for various reasons. In a nutshell, source routing can be particularly used for tunneling network traffic, offers resiliency enhancing possibilities and allows for simplified traffic engineering. Of course further potential use cases exist and might show up as soon as source routing found wide adoption and a stable and universial standard mechanism has been released.

2.2.1 Traffic Engineering

Xiao et al. provide a concise definition of traffic engineering [11], stating that *"Traffic Engineering is the process of controlling how traffic flows through one's network so as to optimize resource utilization and network performance"*. Additionally to the optimization aspect, traffic engineering also allows for implementing service level agreements from a technical point of view, e.g fulfilling agreements between customer and internet service provider about guaranteed bandwidth, delay, stability or throughput [12]. Besides performance concerns, source routing can help implementing formal requirements or administrative policies for traffic transmission and traffic separation, e.g. governmental dictations or separation of security sensitive traffic flows in certain businesses [12].

It is obvious that traffic engineering is an indispensable measure in todays backbone networks, service provider networks and large enterprise networks that consist of many routers, redundant links and alternative paths to increase reliability, enhance performance and avoid congestion [13]. Using source routing, alternate paths to the same destination can be easily imposed to a packet without the need of stateful per-flow routing information established on intermediate nodes, which would elsewise be needed to decide where to forward a packet [12]. Load sharing and load balancing can be achieved by source routing traffic over different paths, even through non-parallel links and even if two distinct paths share the same costs (Equal Cost Multiple Path (ECMP) routing) [2].

2.2.2 MPLS Tunneling

Source Routing in combination with MPLS is a useful and efficient technique to build *Virtual Private Networks (VPN)*, that are transparent for users. IP-based tunneling mechanisms such as IPsec or L2TP encapsulate IP traffic within IP datagrams and thereby rely on the forwarding mechanism of the IP protocol themselves.

However, MPLS allows to tunnel IP traffic seemlessly without encapsulating it into a network layer protocol again, while providing a more direct access on path selection. Moreover internet service providers sometimes also offer *Virtual Private Wire Services (VPWS)*, meaning they connect spatially distributed customer networks on link layer basis by tunneling link layer traffic via their backbone networks [14]. Source routing, which can be implemented on MPLS data plane [2], enables operators to more efficiently create such tunnels and control the data flows and path selections in a direct manner [12].

2.2.3 Resiliency use cases

Source routing constitutes a base mechanism for increasing resiliency, offering fast rerouting capabilities by allowing imposition of alternative paths [2]. Resiliency can be improved by calculating backup paths through a network topology in order to face link failures or node failures on the designated

doi: 10.2313/NET-2017-05-1_05

path from source to destination [2, 15]. Path protection is a technique that augments resilience by calculating a complete disjoint secondary path from source to destination, i.e. using only disjoint intermediate nodes and links to reach the destination [15].

In contrast to path protection, resiliency can be also improved by either bypassing only a faulty link and then returning to the originally designated path, or by bypassing the entire faulty node and thus using an alternative shortest path to the destination [15]. Furthermore source routing can be a mean to bridge temporary microloops [2], which might occur during the short-lived convergence phase of the IGP protocol in use, as routing information is not consistent across all routers at this point time [15]. Source routing helps avoiding these temporary microloops by imposing an explicit path through the network topology to all packets that need to be forwarded during the convergence phase. The latter is possible because source routing can operate independently and regardless of inconsistent routes, metrics, and priorities that are being established during convergence phase and which are causing these loops [15].

2.3 Segment Routing

Segment routing is a new source routing mechanism developed by the IETF SPRING working group. It is based on so-called *segments* [7]. The SPRING architecture [7] defines that a segment represents *"an instruction a node executes on the incoming packet (e.g.: forward packet according to shortest path to destination, or, forward packet through a specific interface, or, deliver the packet to a given application/service instance)"*. A segment and its associated *Segment Identifier (SID)* is advertised within the segment routing domain with the help of the Interior Gateway Protocol (IGP) in use. Therefore the SPRING working group has defined extensions for the IGP protocols OSPF [16], OSPFv3 [17] and IS-IS [18]. With the help of these extensions, those protocols are able to carry the necessary segment routing signaling information. Segment routing introduces three major types of segments [19, 7]:

- IGP-Prefix Segments
- IGP-Node Segments
- IGP-Adjacency Segments

Each of these segment types are discussed in the following sections. The term *ingress node* identifies the node at which a packet enters the segment routing domain, whereas *egress node* identifies the node at which a packet exits the segment routing domain.

2.3.1 IGP-Node Segment

An IGP-node segment has global scope and thus is identified by a globally unique SID. Each node is assigned a SID and advertises its nodal segment via the IGP protocol [20]. Global scope in this context means that all nodes within a segment routing domain add an entry in their Forwarding Information Base for the instruction associated with that segment [12]. The node identified by the node-SID is always reached by the shortest path, which is determined by the IGP algorithm [7]. That means an ingress node can impose a source route to a packet by specifying another node to be traversed by prepending the correspondent node-SID to that packet.

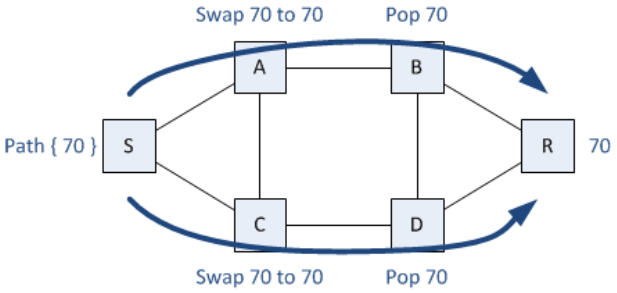


Figure 1: Node Segments [20]

Figure 1 shows an exemplary scenario: Node R advertised its node-SID 70 to all other nodes within the domain. Node S can instruct incoming packets to traverse node R by prepending the node-SID 70 to it. Hence the packet will be either forwarded via the path {S,A,B,R} or {S,C,D,R}, depending on which of both paths have been investigated as the shortest path. Intermediate nodes do not change the prepended SID, thus symbolically swapping it from 70 to 70, except for the last node. The last node on the path towards R is directly connected to R and thereby can remove the SID as this information is not needed anymore [20].

2.3.2 IGP-Prefix Segment

In fact, a nodal segment is a special case of an IGP-prefix segment, as a nodal segment represents a specific node by advertising a prefix of full address length [19]. In general a node can advertise the network prefixes it is attached to with the help of prefix segments, assigning a global segment identifier to each of them (referred to as prefix-SID) [7, 19]. Prefix segments are principally treated and forwarded the same way as explained in section 2.3.1 for nodal segments, with the difference that a certain prefix-SID is only advertised by those nodes that are attached to the respective (sub)network and thus can advertise a route to it within the IGP domain [19]. Prefix segments consequently also have global relevance within the segment routing domain [7].

2.3.3 IGP-Adjacency Segment

An IGP-Adjacency segment in turn has only local scope and thus is identified by a node-locally unique SID. A adjacency segment can be assigned to a specific unidirectional link that is directly attached to a node [7]. This type of segment is likewise advertised within the whole segment routing domain via the IGP protocol in use, but is only installed into the Routing Information Base (RIB) of remote nodes [21]. That means remote nodes can use the SIDs for imposing a route that steers a packet over specific links, but do not install an forwarding entry in their own Forwarding Information Base (FIB) [21]. The latter is only done by the node that advertises the adjacency segment [7].

Multiple SIDs of different types can be stacked and all SIDs together compose the path to be traversed. The imposed

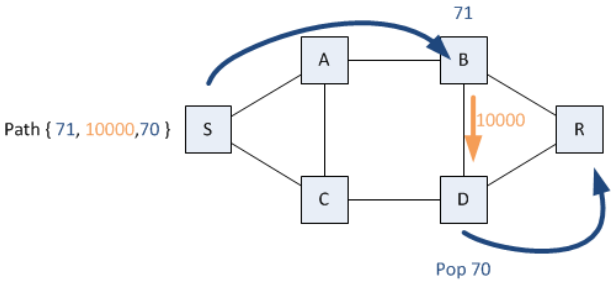


Figure 2: Node and Adjacency Segments [20]

path can either be fully explicit and complete, or define portions of the path while the regular shortest-path algorithm determines the rest of the path [7, 20]. Figure 2 shows combined usage of an adjacency and nodal segments: Node B installed an adjacency segment with SID 10000 in its FIB and advertised it to all other nodes. Additionally node B is also allocating a nodal segment with SID 71. Node R is also allocating a nodal segment with SID 70. An ingress node S can steer packets over the path {S,A,B,D,R} to reach node R by prepending the SIDs {71,10000,70}. Firstly, SID 70 steers packets to node B via shortest-path-first, SID 10000 then causes packets to be forwarded through the directly attached link to node D, and node D is forwarding packets towards R because of SID 70 [20].

2.4 Implementation via MPLS

The implementation of segment routing using the MPLS data plane is pretty straight forward as the forwarding plane is kept unmodified [9]. Since MPLS is a label switching mechanism, it already offers labels that can be used to represent and encode the SIDs [9]. MPLS also offers to stack labels, which allows to stack multiple segment identifiers. It also defines appropriate operations to process and to manipulate the label stack, such as push, pop and swap [5]. The currently active segment to be processed is always considered to be the top of the label stack, with the next segments below accordingly [9]. The main difference between pure MPLS and MPLS for segment routing is, that segment routing does not depend on any additional label distribution protocols such as LDP or RSVP and thus reduces operational complexity. Only the IGP (OSPF or IS-IS) is needed, which is in use either way [9].

2.5 Implementation via IPv6

The motivation for an additional IPv6-only based implementation of the segment routing mechanism is, that some network operators cannot or wish not to deploy MPLS in their IPv6 network [22]. An IPv6-only implementation can be in favor of an easier network administration, a lack of MPLS-enabled hardware or simply better scalability then offered by MPLS label [22]. For more details, this IETF draft "IPv6 SPRING Use Cases" [22] explaines some IPv6-only use cases of segment routing.

Segment routing functionality is added to IPv6 by a new routing extension header of type 4, called *Segment Routing Header (SRH)*. As illustrated in Figure 3, the SRH carries a list of SIDs [8]. SIDs are encoded as IPv6 addresses instead of labels as it is the case when using MPLS. The IPv6 address of a segment routing node serves as node-SID, whereas an adjacency-SID can be any global or local IPv6 address that is not already in use [8]. When reaching the ingress node of a segment routing domain, the original IPv6 datagram is encapsulated with an outer IPv6 header and the SRH [8]. The currently active SID is always copied into the destination address field of the new outer standard IPv6 header, and is further located within the segment list contained in the SRH by the index in the *Segments Left* field [8]. The *Policy List* fields in the SRH are optional and might for example indicate the ingress router at which the packet entered the segment routing domain or the egress router at which the packet is ought to leave the segment routing domain [8].

2.6 Security Considerations

The segment routing architecture assumes a basic trust model: Any node imposing a segment routed path to a packet is legitimate to do so [2]. Hence the operator of a segment routing enabled network has to ensure that all participating nodes within the segment routing domain are trustworthy and are not compromised by malicious evildoers [8]. Furthermore the IPv6-only implementation offers the opportunity to authenticate the segment routing header by an optional HMAC signature field. Consequently, within domains that strictly protect and apply the pre-shared secret key for HMAC computation, no attackers can impersonate as legitimate segment routing nodes [8]. Moreover, RFC 7785 [2] explicitly mandates that a segment routing implementation *"MUST NOT expose any source-routing information when a packet leaves the trusted domain"* and should filter incoming packets from outside the domain that carry segment routed paths [2].

3. ALTERNATIVE SOURCE ROUTING SOLUTIONS

Besides the IETF SPRING working group's standardization efforts, several other source routing solutions exist. A selected set of wellknown and commonly used mechanisms is presented in this chapter. However the heterogeneity of these techniques with regard to efficiency, scalability and maintainability highlights the need of a universal solution. The development of the latter is subject to the SPRING working group.

3.1 Obsolete IPv6 RH0 Extension Header

The Internet Protocol Version 6 defines a source routing mechanism that can be optionally applied by an routing extension header of type 0 (short: RH0) [3]. The extension header allows to define an arbitrary list of non-multicast IPv6 addresses that need to be transitted before reaching the last address of the list, which is representing final destination [3]. The length of this list is only limited by the *Maximum Transmission Unit (MTU)* and the resulting maximum packet size [23].

This routing extension header of type 0 has been deprecated by the IETF due to security concerns and thus is filtered by the majority of routers and firewalls [24]. The fact that the list of addresses can contain arbitrary (non-multicast) entries allows for addresses to appear multiple times. As worst

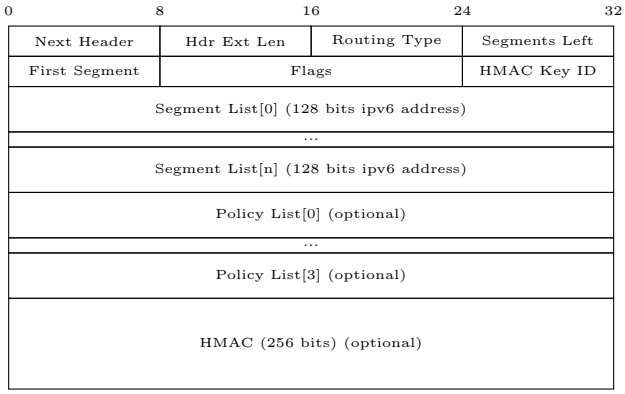


Figure 3: IPv6 Segment Routing Header

case this circumstance can be abused to keep packets travelling between two nodes, consuming bandwidth, switching capacity and processing power on all nodes along this cyclic path [24]. The routing header can thereby be used by attackers to compose *Denial of Service (DoS)* attacks with high efficiency; injected traffic is amplified many times as RH0 enables packets to be routed back and forth [24].

3.2 MPLS with RSVP or LDP

Huge networks that are operated and administrated by a central organizational unit, such as internet service provider networks, extensive company networks or content delivery networks, are often optimized and strengthened by traffic engineering policies. Up to now MPLS is the method of choice to accomplish traffic engineering via source routing [22]. Dedicated protocols such as the *Label Distribution Protocol (LDP)* and *Resource Reservation Protocol (RSVP)* are necessary to communicate label meanings and establish per-flow states on all nodes along a path [25]. Each unidirectional flow needs a label-switched path to be configured on MPLS nodes (called tunnel), i.e. new flow-dependent labels are installed on all MPLS routers on the path from the head-end towards the tail-end router [26]. Both protocols, LDP and RSVP, are used for building up and maintaining such tunnels, but signalization with LDP is limited to the IGP-based shortest-path-first routes, whereas RSVP uses a constrained SPF-algorithm and thus allows for more sophisticated and explicit path configuration [25].

3.3 Routing Protocol for Low-Power and Lossy Networks (RPL)

The *Routing Protocol for low-power and lossy networks (RPL)* is a routing protocol based on the distance-vector algorithm. It has been developed for networks that contain components which are limited in memory, bandwidth, energy and computational power [27]. It only supports IPv6 and helps reducing routing complexity for low-power routers by using the source routing paradigm. As result routers do not need to maintain extensive routing information bases as the path is already encoded in the IPv6 packet. Therefore RPL defines an IPv6 routing extension header of type 3 which carries a list of all next hop addresses needed to reach the final destination [27].

RPL only allows for strict hop-by-hop source routing and tunnels traffic by encapsulating the original incoming IPv6 datagram into a second outer IPv6 header (and its extension header), if the router is not the originator of the packet itself. If the latter is the case, the packet does not need be encapsulated into a second IPv6 header and the routing extension header is directly added to the original IPv6 header [27]. The destination IP address of the (outer) IP header represents the next hop to be visited and is switched to the next address upon reaching the designated next hop. It is important to note that RPL source routing headers have only significance within the RPL domain and must not be carried into other RPL domains [27].

3.4 Dynamic Source Routing (DSR) Protocol for Wireless Ad Hoc Networks

The *Dynamic Source Routing (DSR)* protocol is a self-organizing protocol that is well suited for wireless networks that operate adhoc and without designated infrastructure [28]. Topologies of such networks typically contain nodes located on opposite ends of the network and hence being out of direct range to each other. Such nodes are dependent on other nodes in between to forward packets originated by or destined for them [29]. Moreover, wireless adhoc networks often face high node mobility, thus including nodes that change their location within the network topology as well as occasionally quitting and entering the network [29].

DSR is a protocol that is self-adapting to topology changes by determining on-demand which paths are currently available towards the destination. A node discovers the path(s) to a target on-demand by a broadcasting a *Route Request* message to all neighboring nodes within transmission range. The request contains an ID as well as a list of IP addresses that were previously visited (initially empty) [28, 30]. Receiving nodes either discard the packet because they have received a request with the same ID before, broadcast it again within their transmission range while appending their IP address to the list of intermediate hops, or respond to it with a *Route Reply* message because they are the target of the request. The Route Reply is containing a copy of the list of intermediate hops. The final list that is sent with the Route Reply is used by the initiator of the request for imposing source routes to the packets destined for the target node [28, 30]. Discovered routes are cached in the routing information base and can be used for future packets until the path gets invalid and packets cannot be delivered. In this case the old route is removed from the routing table and a known alternative route is used, or a new route is discovered [28]. Various versions and extensions of DSR exist, offering additional quality properties and optimizations with regard to security [31, 32], energy efficiency [33], node-disjoint paths [34] and many more aspects.

4. DISCUSSION

Comparing the different source routing approaches presented above, the heterogeneity of these techniques with regard to efficiency, scalability and maintainability is remarkable. MPLS using LDP or RSVP for label distribution especially lacks easy maintainability due to complex protocol formats and complex interaction and synchronization of multiple

protocols [35, 36]. In addition RSVP lacks good scalability due to its point-to-point concept [37].The obsolete IPv6 RH0 extension header has no importance for today's networks any longer, as it was officially deprecated by the IETF and is filtered by most routers and firewalls. However it is mentioned in this proceeding because other solutions were inspired by it and learnt from that negative example with regard to security. The Dynamic Source Routing protocol is a well-suited solution for wireless adhoc networks and is also efficiently supporting node mobility [28]. Due to its on-demand approach the signaling overhead is reduced to only those routes that are actually requested [30]. Furthermore no periodic updates flood the network. The signaling overhead can be further reduced by caching not only routes retrieved from node-specific requests but also analyzing all Route Reply messages that have been provoked by other nodes. Moreover one single Route Request message discovers many alternative routes to the destination if present [30]. This results in excellent efficiency for topologies with low node mobility, while signaling overhead and outdated cache states rise with increasing node mobility [28].Segment Routing benefits the most of its seemless integration into both existing MPLS infrastructure and IPv6 networks. Because of its extensions for OSPF and IS-IS, segment routing requires no additional protocol other than the IGP that is already in use [9]. Therefore segment routing reduces the complexity of the source routing architecture and allows for simplified administration and maintenance of source routed network domains [38]. Providing the opportunity to steer traffic over specific links permits to use segment routing for efficient traffic engineering purposes. Furthermore the IPv6 implementation of segment routing offers authentication and integrity protection of the imposed source route via HMAC [8], which is not offered by MPLS, RPL or DSR by default.

In a nutshell, all mechanisms that encode the source route in packets have in common that the overhead per packet increases with the route length. On the other hand, mechanisms that do not encode the source route in packets but maintain per flow states on intermediate nodes tend to scale worse and require more memory and processing power on source-routing enabled network components.

5. CONCLUSION

With todays extensive and increasing usage of network infrastructure, source routing will become a key technology for large-scale networks in oder to optimize traffic flows and implement traffic policies. Especially with regard to content delivery networks and the associated tremendous amout of data to be exchanged, source routing will more and more become an important mean for an efficient allocation of infrastructure resources. The growth of giants like Youtube, Amazon, Netflix and many other multimedia streaming services and IPTV services indicate an outrageous network load that will increase even more and needs to be accommodated in future. Segment Routing is a promising technique that shows potential to establish an universal standard for source routing. Segment Routing seems to be an appealing technique for producers of networking hardware and network operators. The industrial interest in a unified and standardized source routing solution is quite obvious, as the IETF SPRING working group experiences broad support of huge companies such as Cisco, Nokia, Juniper and some more, which attend the working group's meetings or even send associates to contribute to the working group's activities.

The success of segment routing will largely depend on the security of this technology, because this has already been a deal-breaker in the past (just recall the abandoned IPv6 RH0 extension header). The future will show whether network operators succeed in preventing abuse through malicious attackers. Since backbone and provider networks have always been worthwhile targets for attackers as they offer the potential to paralyze or spy a large portion of the internet traffic, one can be sure that hackers will sound every security flaw that comes with segment routing. Considering the tremendous and continuously growing sizes of internet service provider networks or data center architectures, operating secure segment routing will be a challenging task. Especially with regard to an attacker from the inside of a segment routing domain, who is not restrained by the assumed basic trust model, but is quite likely due to many staff members, security will be a challenge to cope with.

6. REFERENCES

[1] L. L. Peterson and B. S. Davie, *Computer networks: a systems approach*. Elsevier, 2007.

[2] S. Previdi, C. Filsfils, B. Decraene, S. Litkowski, M. Horneffer, and R. Shakir, "Source Packet Routing in Networking (SPRING) Problem Statement and Requirements." RFC 7855 (Informational), May 2016.

[3] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification." RFC 2460 (Draft Standard), Dec. 1998. Updated by RFCs 5095, 5722, 5871, 6437, 6564, 6935, 6946, 7045, 7112.

[4] S. Krishnan, J. Woodyatt, E. Kline, J. Hoagland, and M. Bhatia, "A Uniform Format for IPv6 Extension Headers." RFC 6564 (Proposed Standard), Apr. 2012.

[5] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture." RFC 3031 (Proposed Standard), Jan. 2001. Updated by RFCs 6178, 6790.

[6] A. Retana and S. Bryant, "Charter: Source Packet Routing in Networking," 2013. `https://datatracker.ietf.org/doc/charter-ietf-spring/`; last accessed on 2016/12/20.

[7] C. Filsfils, S. Previdi, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture," 2016. `https://tools.ietf.org/html/draft-ietf-spring-segment-routing-10`; last accessed on 2016/12/19.

[8] C. Filsfils, S. Previdi, B. Field, and I. e. a. Leung, "IPv6 Segment Routing Header (SRH)," 2015. `https://tools.ietf.org/html/draft-previdi-6man-segment-routing-header-08`; last accessed on 2016/12/20.

[9] C. Filsfils, S. Previdi, B. Decraene, S. Litkowski, and R. e. a. Shakir, "Segment Routing with MPLS data plane," 2016. `https://tools.ietf.org/html/draft-ietf-spring-segment-routing-mpls-05`; last accessed on 2016/12/18.

[10] SPRING working group, "Status report for SPRING WG meeting on 2016-11-17," 2016. `https://www.ietf.org/proceedings/97/slides/slides-97-spring-0_ietf97_spring-wg-status-00.ppt`; last accessed on 2016/12/18.

[11] X. Xiao, A. Hannan, B. Bailey, and L. M. Ni, "Traffic Engineering with MPLS in the Internet," *IEEE network*, vol. 14, no. 2, pp. 28–33, 2000.

[12] Cisco Systems Inc., "Segment Routing: Prepare Your Network for New Business Models," 2015. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/application-engineered-routing/white-paper-c11-734250.html; last accessed on 2016/12/14.

[13] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao, "Overview and Principles of Internet Traffic Engineering." RFC 3272 (Informational), May 2002. Updated by RFC 5462.

[14] Juniper Networks Corp., "Understanding VPWS," 2015. https://www.juniper.net/documentation/en_US/junos14.2/topics/concept/vpws-overview.html; last accessed on 2016/12/20.

[15] C. Filsfils, S. Previdi, B. Decraene, and R. Shakir, "SPRING Resiliency Use Cases," 2016. https://tools.ietf.org/html/draft-ietf-spring-resiliency-use-cases-08; last accessed on 2016/12/16.

[16] P. Psenak, C. Filsfils, R. Shakir, H. Gredler, W. Henderickx, and J. Tantsura, "OSPF Extensions for Segment Routing," 2015. https://tools.ietf.org/html/draft-ietf-ospf-segment-routing-extensions-10; last accessed on 2016/12/20.

[17] P. Psenak, C. Filsfils, R. Shakir, H. Gredler, W. Henderickx, and J. Tantsura, "OSPFv3 Extensions for Segment Routing," 2016. https://tools.ietf.org/html/draft-ietf-ospf-ospfv3-segment-routing-extensions-07; last accessed on 2016/12/19.

[18] S. Previdi, C. Filsfils, H. Gredler, S. Litkowski, B. Decraene, and J. Tantsura, "IS-IS Extensions for Segment Routing," 2016. https://tools.ietf.org/html/draft-ietf-isis-segment-routing-extensions-09; last accessed on 2016/12/20.

[19] S. Salsano, L. Veltri, L. Davoli, P. L. Ventre, and G. Siracusano, "PMSR-Poor Man's Segment Routing, a minimalistic approach to Segment Routing and a Traffic Engineering use case," *arXiv preprint arXiv:1512.05281*, 2015.

[20] Y. El Fathi, "Introduction To Segment Routing," 2013. http://packetpushers.net/introduction-to-segment-routing/; last accessed on 2016/12/20.

[21] D. Singh, "Yet Another Blog About Segment Routing - Part 1," 2015. http://packetpushers.net/yet-another-blog-about-segment-routing-part-1; last accessed on 2016/12/18.

[22] J. Brzozowski, J. Leddy, M. Townsley, C. Filsfils, and R. Maglione, "IPv6 SPRING Use Cases," 2016. https://tools.ietf.org/html/draft-ietf-spring-ipv6-use-cases-07; last accessed on 2016/12/20.

[23] B. Curtin, "Internationalization of the File Transfer Protocol." RFC 2640 (Proposed Standard), July 1999.

[24] J. Abley, P. Savola, and G. Neville-Neil, "Deprecation of Type 0 Routing Headers in IPv6." RFC 5095 (Proposed Standard), Dec. 2007.

[25] Juniper Networks Corp., "Understanding the RSVP Signaling Protocol," 2013. https://www.juniper.net/documentation/en_US/junos12.1x47/topics/concept/mpls-security-rsvp-signaling-protocol-understanding.html; last accessed on 2016/12/20.

[26] D. Singh, "Yet Another Blog About Segment Routing - Part 3," 2015. http://packetpushers.net/yet-another-blog-segment-routing-part3-sr-te/; last accessed on 2016/12/18.

[27] J. Hui, J. Vasseur, D. Culler, and V. Manral, "An IPv6 Routing Header for Source Routes with the Routing Protocol for Low-Power and Lossy Networks (RPL)." RFC 6554 (Proposed Standard), Mar. 2012.

[28] D. B. Johnson, D. A. Maltz, J. Broch, *et al.*, "DSR: The dynamic source routing protocol for multi-hop wireless ad hoc networks," *Ad hoc networking*, vol. 5, pp. 139–172, 2001.

[29] D. Johnson, Y. Hu, and D. Maltz, "The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4." RFC 4728 (Experimental), Feb. 2007.

[30] B. Awerbuch, A. Mishra, and J. Hopkins, "Dynamic Source Routing (DSR) Protocol," *Johns Hopkins University, US*. http://www.cs.jhu.edu/~cs647/dsr.pdf; last accessed on 2016/12/20.

[31] F. Kargl, A. Geiß, S. Schlott, and M. Weber, "Secure Dynamic Source Routing," in *HICSS*, 2005.

[32] T. Jiang, Q. Li, and Y. Ruan, "Secure dynamic source routing protocol," in *Computer and Information Technology, 2004. CIT'04. The Fourth International Conference*, pp. 528–533, IEEE, 2004.

[33] M. Tarique, K. E. Tepe, and M. Naserian, "Energy saving dynamic source routing for ad hoc wireless networks," in *Third International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt'05)*, pp. 305–310, IEEE, 2005.

[34] J. Wu, "An extended dynamic source routing scheme in ad hoc wireless networks," *Telecommunication Systems*, vol. 22, no. 1-4, pp. 61–75, 2003.

[35] M. Jork, A. Atlas, and L. Fang, "LDP IGP Synchronization." RFC 5443 (Informational), Mar. 2009. Updated by RFC 6138.

[36] S. Kini and W. Lu, "LDP IGP Synchronization for Broadcast Networks." RFC 6138 (Informational), Feb. 2011.

[37] P. Lapukhov, "Scaling MPLS Networks," 2010. http://blog.ine.com/2010/08/16/scaling-mpls-networks/; last accessed on 2016/12/19.

[38] A. Korzh, "Segment-Routing." https://www.enog.org/presentations/enog-6/201-SR_ENOG.pdf; last accessed on 2016/12/20.

SQL, noSQL or newSQL – comparison and applicability for Smart Spaces

Christoph Rudolf
Advisors: Stefan Liebald, M.Sc. and Dr. Marc-Oliver Pahl
Seminar Innovative Internet Technologies and Mobile Communication (WS 16/17)
Chair for Network Architectures and Services
Department of Informatics, Technische Universität München
Email: christoph.rudolf@tum.de

ABSTRACT

In this paper, we analyze database architectures in terms of their applicability as a data storage for distributed data in smart spaces. As a concrete example, the paper takes the *Distributed Smart Space Orchestration System (DS2OS)* which acts as a management system for smart spaces and implements a peer-to-peer network. We develop key requirements for a database solution based on the data characteristics of DS2OS. These requirements allow us to analyze representatives of database architectures regarding their applicability. We conclude that the simple data models provided by NoSQL databases are suitable for modeling the desired data structure. Based on their performance and features, we propose *PostgreSQL*, *Redis* and *InfluxDB* as suitable solutions for DS2OS.

Keywords

NoSQL, NewSQL, database types, DS2OS, IoT, smart spaces

1. INTRODUCTION

Smart spaces are real-world spaces with embedded devices that capture data about their environment and control aspects of it [14]. The data that is shared in smart spaces originates from distributed sources connected to each other. This offers challenges regarding the storage of data handled by a common middleware. The *Distributed Smart Space Orchestration System (DS2OS)* is such a middleware and is used as a representative to asses different paradigms for database management regarding their suitability as a data storage in smart spaces. This paper starts by introducing special characteristics of data in DS2OS. Further, it gives an overview over database architectures. We then develop requirements to be met by a suitable database solution and select representatives of each architecture type for evaluation.

2. DATA STRUCTURES IN THE DS2OS

The *Distributed Smart Space Orchestration System (DS2OS)* is a management system for smart devices. It was started in the Ph.D. thesis of Marc-Oliver Pahl [14] and is actively developed at the Chair of Network Architectures and Services at the Technical University of Munich[1]. The system contains a middleware for the uniform abstraction of smart spaces. It targets the heterogenic field of today's smart devices and makes them manageable by a central software.[16]

[1] `www.ds2os.org`

While this paper focuses specifically on the data structure used in DS2OS, we start by introducing key architecture components which are necessary for understanding the data structure. The central element and middleware of DS2OS is the *Virtual State Layer (VSL)*, a peer-to-peer system built of so called *Knowledge Agents (KA)* as its peers [16]. The VSL is self-organizing in a sense that peers automatically locate and connect to each other. In addition to that, the VSL handles synchronization of data between agents. In a real-world scenario, these interconnected agents run on computing devices in different physical locations of a smart space. They provide an interface between a diverse set of services and the VSL. Services can, for example, manage a smart device and encapsulate its complexity, collect sensor data or output information to the user.

One of the major aspects of a system like DS2OS is the design of a suitable data model for all possible services and devices. This is especially important to allow for previously unsupported devices to be introduced into the system. Therefore, domain-specific data has to be added dynamically.[16] To do so, DS2OS provides a directory of *context models* that can be extended by developers. A context model is a data structure that holds properties of a service and acts as an interface. A standard context model for services which are, for example operating a lamp, is desired to be adopted by many services for lamps to decouple service implementation from the specific devices. Context models are currently stored in an XML-format.[15]

The context model of a service is instantiated when a service initially connects to a KA. The context models consist of "hierarchically structured typed key-value pairs (...) with additional management metadata" [15] and can be thought of as a tree data structure that allows to address each attribute, similar to files in a filesystem. Figure 1 visualizes the interconnected agents, their services, the data as well as sample attributes for a lamp as service S_1.

All information depicted in Figure 1 is known to every KA in the VSL as meta data. However, only the KA that is directly connected to a service stores the actual data. Other agents have to actively query the respective KA for this data by referring to its path in the hierarchic structure (e.g. `/KA1/S1/switchedOn/`) which forms a unique identifier. They can also subscribe to an address and be notified of changes. These notifications are sent by the KA in charge of storing the data. Storing the data as well as the structure

doi: 10.2313/NET-2017-05-1_06

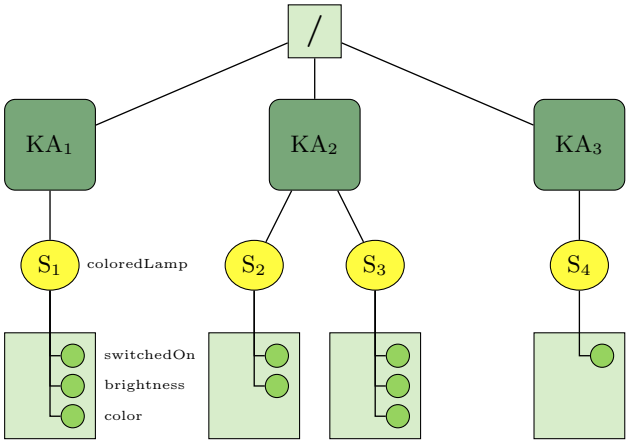


Figure 1: Visualization of the DS2OS data structure known to each KA.

is currently implemented by using the relational database *HSQLDB* with a database schema of three tables:

structure : {[address, type, readers, writers, restriction, cachedParameters]}
version : {[address, timestamp, version]}
data : {[address, timestamp, value]}

The address inside the tree structure references specific entries in all three tables. All addresses are associated with a semantic data type in the context of the VSL that is independent of the type in the database.

The `structure` table is responsible for storing the complete structure depicted in Figure 1 and additional metadata like restrictions and permissions. The `version` table receives new entries for every change made to a value, as the `data` table is not updated in case of a value being changed. Instead, a new entry with a timestamp is added, in addition to the new version in the other table. The version update via insert does also happen for all prefixes of the changed address up to the service level. For this reason, separated tables help to insert versions for all parent nodes without duplicating their values. By inserting new versions, the database for DS2OS has to handle mainly insert operations.

3. DATABASE ARCHITECTURES

Due to new database architectures, the field of *database management systems (DBMS)* has been diversified over recent years. While relational databases are prevalent in the majority of all use cases, unstructured data calls for new approaches [11]. Not all DBMS can be assigned solely to a single architecture type of databases, as many provide features that are characteristics of other groups. This section provides a brief overview over current database architectures.

3.1 Relational database

Relational databases are currently the most widespread type of databases [6], with well-known DBMS like *PostgreSQL* and the briefly mentioned *HSQLDB* used in DS2OS.

Their idea was proposed as early as 1970. With this kind of databases, data is stored in tables which model relations in a mathematical sense. Each table models an n-ary relation, by specifying n columns with each storing a certain type of information for all entries. The column definition provides the scheme of the table. The data itself is stored as row entries in form of n-tuples.[3] This design approach makes relational databases adaptable to many application domains. They allow to define certain columns as unique key values and to create references between relations. If utilized correctly, this reduces redundancy of data.

Relational databases provide powerful querying capabilities based on relational algebra. Relational algebra defines a set of operations working on relations. Every output generated by a query combining relations with their operations is a new relation in itself. This enables complex nested queries. It is within the responsibility of the user or an optimizer integrated into a DBMS to make them as efficient as possible. The features of relational algebra are implemented by SQL which acts as a *de facto* standard query language for modern relational databases.[5]

Relational databases usually do well in terms of concurrency and reaction to failures as they provide ACID transactions. By being ACID compliant, transactions are supposed to fulfill the following principles [10]:

- ***A**tomictiy:* transaction are considered as a single unit
- ***C**onsistency:* results are consistent database states
- ***I**solation:* no side-effects among parallel transactions
- ***D**urability:* their results are persistent

This is necessary if a series of critical changes is made that can only be allowed to complete fully or have no effect at all. While transactions are no inherent attribute of relational databases, it usually distinguishes them from newer approaches. Whether or not transactions are needed is subject to the application domain.

3.2 NoSQL

While relational databases are adaptable to many scenarios, not all data models fit well with a relational schema. Relational databases are inflexible when the schema has to be changed due to redesigns in the underlying application. In addition to that, their sophisticated features, like ensuring data consistency, add overhead which is sometimes unnecessary and critical in contexts of massive amounts of data being handled. This has become an issue for online services over the recent years, which are in need of database systems that can be distributed.[7, 9]

NoSQL databases solve these problems with different approaches to data organization and an increased focus on distributability. The term NoSQL does not describe a specific group of databases, instead, it is a collective term for systems that break the classic relational approach.[8] Without the relational basis, NoSQL databases do not provide a common query language. In order to improve the scalability, many NoSQL databases abandoned ACID transactions. Subsequently, we discuss the most important types of NoSQL databases.

doi: 10.2313/NET-2017-05-1_06

3.2.1 Key-value stores

This kind of NoSQL database models all data as simple key-value pairs. Values are stored as plain byte-arrays. usually without a data type specified on the database side. While this is limiting in terms of data structuring, the storage is extremely adaptable, as changes to the application do not require the database to be adapted. It also does not require `null` values like they appear in relational models. Complex queries are not possible due to the lack of interpreted data, however, because of the flat and simple structure, this is not needed. Instead, simple `get` and `set` commands in conjunction with unique keys are the only way to read and write data to the store. Many implementations, such as *Redis* [6], work as in-memory databases with optional mechanisms for saving to a persistent disk. Being in-memory allows them to have a much faster but, in terms of size, limited storage. *Redis* in particular does also support data types.[8, 9]

3.2.2 Document-oriented database

Document-oriented databases are related to key-value stores, but add a means of structuring to them by allowing to group key-value pairs into documents. The keys then only have to remain unique inside the scope of the document. Each document receives a unique identifier as well. Queries to fetch data are improved beyond simple `get`-operations by also allowing to query documents based on their properties inside. The values associated with the keys are not opaque byte arrays like for key-value stores. Documents are allowed to be nested further and support basic datatypes like lists, strings and numeric values. The features are similar to the ones being expressed by JSON syntax which leads to many document-oriented DBMS using JSON or at least a JSON-like data format. There is no restriction on the schema of documents which allows adding and removing entries to and from, possibly heterogenous, documents. Utilizing the XML document standard is also possible and allows queries with *XQuery*, a query language that aims to fulfill the same role as SQL in the relational world.[9]

3.2.3 Column-oriented database

This type of databases is similar to key-value stores in a sense that key-value pairs are stored without any interpretation of the values inside the database. Like for key-value stores, additional logic regarding relationships and types of the values is shifted to the application. The main difference of this database type to key-value stores is that it deviates from being row oriented in its physical organization. Instead, all values of each column are grouped together. This is beneficial in cases where not all values of a single row are of interest. Due to grouped columns, their values are also stored very close in memory. This optimizes the performance in cases where all rows and only few columns of each one are needed, as there will be less page faults and therefore less slow hard disk access.[9, 8] The concept was initially introduced by Google's Bigtable and is not limited to NoSQL, as relational data can also be organized column-oriented.

3.2.4 Graph-based

Graph-based databases are optimized for data which emphasizes the relationship between data objects. Their primary elements for data modeling are nodes and edges which may be associated with key-value pairs to store additional properties. Querying relies on different graph traversal algorithms like breadth-first search to find single matching nodes or depth-first search for shortest paths. While graph structures can be represented in a relational database, graph traversal cannot be expressed in SQL which leads to decreasing performance for larger data sets [8]. Examples are social networks (e.g. Twitter's FlockDB) or storing location data.

3.2.5 Time series database

This type of database focuses on handling time series data. This covers all data with one or multiple values being stored at continuous time intervals while one is still being interested in past values for processing them. Typical scenarios in which time series appear are statistical data, logging or sensor data. A time series is identified by a name, includes a timestamp and values. This could basically be modeled by a relational database if it is feasible to create a new table for each time series. By putting more than one series into a table, one would have to define the superset of all values as its scheme and deal with many `null` values or make the values opaque to the database which hurts querying. Time series databases are specifically built for handling this data efficiently and are especially useful if the amount of data is very large.[12]

3.2.6 Object-oriented databases

According to the *db-engines.com* ranking [6], object-oriented databases are currently among the least used databases. Their distinct advantage over relational databases is that they are capable of modeling concepts like inheritance and polymorphism. This is especially useful if the application using the database utilizes object orientation as well. Therefore, object-relational mapping (ORM), which is often used to translate relation data into objects for usage in the application, is not needed, thus, removing an additional layer of complexity when accessing the database.

However, there are disadvantages which account for the limited acceptance of object-oriented DBMS. Due to their diverse structure depending completely on the domain model they are used for, they cannot rely on relational algebra and cannot create new objects by joining existing ones in queries. In addition to that, they lack a standard query language, like SQL for relational databases.[2] This kept object-oriented databases from getting increasing market shares. Instead, using relational databases and ORM is used by most object-oriented applications with databases.

3.3 NewSQL

The term NewSQL describes new database architectures, deviating from the way relational DBMS are implemented. In contrast to NoSQL databases, they aim to maintain characteristics of relational databases, like SQL as a query language, support for the relational model and ACID transactions. At the same time, they provide additional performance, scalability and distributability. This is done by leveraging modern hardware and deploying improved algorithms which haven't been available yet, when older existing DBMS had been designed. NewSQL databases rewrite the core of the database system from scratch to remove legacy code that hinders distributability and performance due to its assumptions being outdated.[8] By rewriting the very basis of a DBMS, they utilize modern multi-core architecture

doi: 10.2313/NET-2017-05-1_06

and support clustering in a native way compared to older RDBMS which are missing these features or had them added in a much later stage of their product lifecycle.

The main concepts of NewSQL have been introduced by Stonebraker et. al. [19] in a paper that also introduced the implementation *H-Store* that provides ACID transactions and excels at *online transaction processing (OLTP)*. The paper also points out that a single architecture is not sufficient in all use cases and that multiple but very optimized DBMS for each field of application are preferable.

Operating in-memory is often mentioned as a key property of NewSQL, making relational databases like *SAP Hana* which supports this feature also a NewSQL database to some extent. Other examples are *Google Spanner* and *VoltDB*

4. CRITERIA FOR COMPARISON

The differences among the wide variety of databases introduced in Section 3 show that the most suitable choice for a database depends on the needs of the application at hand. In order to determine a fitting type or even a concrete DBMS software for usage within DS2OS, we have to define the criteria that are important for our use case. In the process, we identify criteria that are quantifiable and can be determined by measurements, like for example performance values. In addition to the measurable criteria, there are requirements which *must* be fulfilled and other properties that are considered to be beneficial but *may* not be fulfilled. We distinguish these as *hard* and *soft* requirements.

General criteria for evaluation without a specific use case are presented in related work [9, 20] and cover attributes like scalability, query support, broad data model support and distributability. These are only partly applicable in our case, as they are very general or not of importance in our special case. Nevertheless, ideas for evaluation criteria can be drawn from related work.

4.1 Hard requirements

H1: The database must be able to store the data presented in Section 2. It is not necessary to model the same relational schema. However, the database must support basic key-value pairs associated with a timestamp. The additional versioning of data is currently modeled in a separate database table, however, it would be beneficial to be handled natively (see Section 4.2). The same goes for access control to certain service data.

H2: As DS2OS is currently written in Java, the DBMS of choice has to provide an API accessible from this programming language. This does not limit the field to databases that directly provide Java bindings, as an HTTP API or a connection via ODBC is also perfectly usable from Java.

H3: The current implementation of DS2OS in combination with *HSQLDB* as a database does use transactions. This is important when data is initially inserted or deleted, as this affects not only the data entry itself, but also the version stored in a separate table. Unless the proposed DBMS of choice eliminates the separation between data, version and structure completely, further support for ACID transactions is required. However, a completely different data handling could in fact lift this requirement.

4.2 Soft requirements

Many of the subsequent properties are either currently missing or are handled inside the application, because the current database does not offer support on the particular issue. It is preferable to have native support for these features instead of an implementation in Java. However, due to necessary code changes to the VSL, each of the subsequent requirements has to be put into perspective regarding how *invasive* it is in regard changes necessary to the DS2OS existing code base. With both, benefit and code changes considered, a score (✓or ✓✓) indicating its importance is given.

S1: Versioning is currently handled by the application layer of DS2OS. The relational database schema is designed to allow storing versioning information for each value. A solution like this is feasible for a future choice of database, but a native solution directly integrated within the database is preferred. We attribute such DBMS additional points during the assessment. As the current implementation encapsulates the handling of versions completely within the database wrapper, there are no further changes to the DS2OS coming with this feature. **Bonus: ✓✓**

S2: As the relational *HSQLDB* does not support being distributed directly, the distribution of data is handled on the application layer by the KA in charge of the data. Therefore, the current implementation of DS2OS provides application layer code for notifying peers of changes and ensuring that only structural data is shared. Many NoSQL databases provide native support for distribution over multiple nodes which removes complexity from the application layer. However, as this is currently directly implemented in the VSL, the refactoring of the existing code would be rather complex. Another downside to take into consideration, is that distribution on a database level imposes the usage of the same database for all agents in the network. Mixing databases among agents is currently possible due to the handling on a higher level. Hecht and Jablonski separate distributability into the support for partitioning data among multiple nodes and the replication of data [9]. In our scenario, replication is needed only for structural data. Partitioning has to be controlled to ensure that values are confined to a specific agent. **Bonus: ✓**

S3: Access control is currently available and implemented by storing which agents have read and write access to an address. The application layer processes data queries from remote Knowledge Agents and uses the stored information to grant or deny access. A native support would eliminate complexity from the agents and simplify the data model. The desired mechanism is that a KA can query data under the permissions of a peer. This is a more complex problem than the one addressed by standard role-based permissions of most DBMS. Native support on the database side should also allow querying whether or not permission is given without fetching the actual data. The amount of changes to the given code basis are considered as being in between those of the last two requirements, with smaller changes outside the immediate database wrapper. **Bonus: ✓**

doi: 10.2313/NET-2017-05-1_06

S4: Besides access control to sensor data through its managing KA, there is currently little additional security, like encryption of the data, provided. Confidentiality of the data *at rest* or even *in memory* is desirable as the possible damage coming from malicious changes to sensor data in smart spaces can result in actual physical damage. Encryption for data *in transit* is already provided by DS2OS. **Bonus: ✓✓**

S5: DS2OS allows agents to subscribe to data managed by another peer to be notified of changes. A support for such a messaging system directly inside the database is beneficial as it would reduce computational cost for the KA when processing changes. However, changes to the code outside of the database wrapper are necessary. **Bonus: ✓**

4.3 Measurable criteria

The subsequent criteria are measurable metrics regarding the performance of databases.

C1: The insert performance of the DBMS is important, as DS2OS uses mostly insert requests. Because of versioning, new entries are written for changing values. In nested data structures the number of inserts for a changing value is large as all parents up to the service node get a new version.

C2: One of the aspects a different database can improve, is the read performance. This is important due to possibly many services requesting data in a productive smart environment. Because of the tree-like structure being stored in relational database tables, querying a node and all its child elements results in many different rows to be fetched. The current database has to rely purely on string comparison of addresses to filter for the correct entries to return.

C3: Deleting is important to prevent the database from growing to large. Old versions of data are deleted if the total number of entries exceeds a certain threshold. This means that once this threshold is surpassed for an entry, every insert does also include the removal of old data. Therefore both insert and delete commands are highly important for the new database, while the performance for updates is negligible.

5. ASSESSMENT OF DBMS

Due to the large amount of existing DBMS, we pick representatives for each of the categories introduced in Section 3. Afterwards, we conduct a preliminary based on hard requirements **H1**–**H3** to narrow down the representatives before comparing them in regard of the soft requirements **S1**–**S5** and the quantifiable criteria **C1**–**C3**.

5.1 Selection of DBMS

Table 1 lists representative DBMS for each database type. The DBMS *PostgreSQL*, *Redis*, *MongoDB*, *Cassandra* and *Neo4j* are picked based on db-engines.com's DBMS ranking of popular DBMS [6]. This is reasonable based on the assumption that popular databases are under active and continuing development as well as field-tested and optimized to a degree where using them in production can be recommended. In addition to that, all selected databases are open source to fit into DS2OS which aims to act as an "enabler for a software maker culture" [16]. Therefore, it is not desired to introduce a commercial database solution.

Table 1: Representatives of each database type for further evaluation.

Database type	Representative DBMS
Relational	HSQLDB, PostgreSQL, VoltDB
Key-value stores	Redis
Document-oriented	MongoDB, Elasticsearch
Column-oriented	Cassandra
Graph-based	Neo4j
Time series database	InfluxDB
Object-oriented	ZeroDB

With *HSQLDB*, we include the current database of DS2OS for comparison. *VoltDB* is included as a in-memory NewSQL database. Note that NewSQL is not a distinct category as its representatives can implement any architecture. *Elasticsearch* is special as it is technically a search engine but can function as a document-oriented database due to storing schema-free JSON. It emphasizes on distributability by offering features like automatic clustering. The relatively unknown *ZeroDB* is included due to its unique capabilities in terms of encryption. *InfluxDB* is the most prominent time series database [6] which works in clusters of many instances. Querying can be done in a SQL-like query language that returns JSON data, with very sophisticated capabilities regarding time based queries.

5.2 Rating based on hard requirements

With the total of 10 DBMS picked, we can assess their features in regard to the hard requirements **H1**–**H3**. If one or more of these requirements are not met, the respective DBMS is eliminated from further evaluation.

Table 2: Compliance of DBMS with our hard requirements (see Section 4.1).

DBMS	H1	H2	H3
HSQLDB	✓	✓	✓
PostgreSQL	✓	✓	✓
VoltDB	✓	✓	✓
Redis	✓	✓	✓
MongoDB	✓	✓	(✗)
Elasticsearch	✓	✓	(✗)
Cassandra	✓	✓	✗
Neo4j	✓	✓	✓
InfluxDB	✓	✓	(✗)
ZeroDB	✓	✓	✓

H1: Table 2 shows that all DBMS are capable of handling the relatively simple data structure of DS2OS. Even the time-series DBMS *InfluxDB* is applicable in that regard as it allows to store strings in contrast to other time-series databases like *RDRTool*. Time-series databases, alongside graph-based databases, are the only ones where special restrictions of a concrete DBMS could hinder the modeling of DS2OS's data.

H2: No specific type of database is likely to have issues with being coupled to a Java application. The database systems listed provide either a Java API or allow access over the *JDBC (Java Database Connectivity)* interface. *ZeroDB*

doi: 10.2313/NET-2017-05-1_06

is a small exception, as it is based on the object-oriented DBMS *ZODB* which can only be used with Python. *ZeroDB* mitigates this issue by providing an API server that forwards queries to the database on behalf of the client.

H3: The requirement for ACID transactions can be lifted if the DBMS allows for a different approach without the need for transactions. This is promising in the case of *MongoDB*, *Elasticsearch* and *InfluxDB*. The column-oriented *Cassandra* is still in a sense relational and would have to model the DS2OS data like it is done currently (see Section 2). Therefore we eliminate *Cassandra* from further evaluation.

5.3 Rating based on soft requirements

The remaining databases are analyzed regarding the soft requirements **S1**–**S5**. Table 3 summarizes this evaluation. The bonus values for each soft requirement are given partially based on how elaborate the feature provided by the DBMS is on the specific regard. Unless stated otherwise, the information on available features is taken from the product's documentation.

Table 3: Compliance of DBMS with our soft requirements (see Section 4.2).

DBMS	S1	S2	S3	S4	S5	Σ
HSQLDB				✓		**1**
PostgreSQL			✓	✓	✓	**3**
VoltDB					✓	**1**
Redis		✓			✓	**2**
MongoDB		✓		✓	✓	**3**
Elasticsearch	✓	✓	✓			**3**
Neo4j			✓			**1**
InfluxDB	✓✓					**2**
ZeroDB				✓✓		**2**

The narrow field in terms of score and the highest value of 3/7 indicates that there is no ideal solution that performs exceptionally well.

S1: The first requirement revolves around versioning of data. The time series database *InfluxDB* excels naturally as versioning is a key concept of this database type. *Elasticsearch* is the only other DBMS that provides dedicated versioning features by an internal integer that can be atomically incremented [13]. For all other database solutions, versioning like it is currently done with *HSQLDB* via a database table, is the only option.

S2: Distributability, in terms of replication and partitioning, is natively supported by three candidates. *Redis* can partition its key space. Each of the nodes in the resulting cluster holds a portion of all keys. Queries for a certain key can be redirected to the node storing the correct portion. *MongoDB* provides a similar mechanism with *sharded clusters*. This is also the way *Elasticsearch* handles distribution of data. *Neo4j* does also allow to create clusters, but only in its enterprise version. Based on requirement **S2**, it is necessary to enforce that certain values are only stored on a particular host. All three candidates can solve this by using tags. For *MongoDB* this is called *Tag Aware Sharding*. *Elasticsearch* supports it as *Shard Allocation Filtering*.

S3: A permission system is beneficial for DS2OS if it allows to take the role of another KA when querying data. Both *PostgreSQL* and *Elasticsearch* provide features to query as a certain database user. The complete permissions of a user can also be queried separately. *Neo4j* models permissions with special edges that can be taken into consideration even if querying as a different user.

S4: As a database that focuses on security, *ZeroDB* provides the most advanced encryption features. It is the only assessed DBMS that does not even hold decrypted data in its memory. Other DBMS like *HSQLDB*, *PostgreSQL* and *MongoDB* provide encryption features for data at rest, but hold plain data in memory.

S5: A native subscription mechanism is provided by four of the DBMS. *PostgreSQL*, *Redis* and *MongoDB* all deploy a mechanism where the database actively published notifications to all client applications on inserts, updates and the removal of data. Access to this feature is possible from Java clients in all cases. *VoltDB* does not have a publish-subscribe mechanism, but its export functionality can be used to send live updates to an external application [21].

5.4 Evaluation of measurable criteria

The measurable performance criteria **C1**–**C3** focus on write, read and delete operations. Due to the consistent versioning, update operations are hardly used in DS2OS. To narrow down the field of databases for performance evaluation, we only consider candidates that earned at least 2 points in the evaluation of Section 5.3 and the currently used *HSQLDB*. The remaining databases are *HSQLDB*, *PostgreSQL*, *Redis*, *MongoDB*, *Elasticsearch*, *InfluxDB* and *ZeroDB*.

As existing work on the comparison of databases is sparse, we use a combination of the existing measurements and our own experiments. This covers the candidates in a way their performance can be put into relation. This way, not all databases are measured together. Instead, we compare the performance transitively to make a reasonable statement about the databases.

An existing evaluation based on the *Yahoo! Cloud Serving Benchmark (YCSB)* covers *MongoDB*, *Redis* and *Elasticsearch* regarding their performance for insert (**C1**) and read (**C2**) commands [1]. Their execution time for different sizes of data is visualized in Figure 2 and Figure 3 respectively.

The comparison shows that *Elasticsearch* lacks in terms of insert performance compared to the other DBMS, but excels when reading large data. We also conclude that *Redis* has the best overall performance for reading and writing. Due to limited hardware used for this benchmark, the absolute values are not comparable to subsequent tests.

To create additional comparison options beside the existing work, we conduct our own measurements including *HSQLDB*, *PostgreSQL* and *Redis*. This does help to put *MongoDB* and *Elasticsearch* in perspective, as they have been compared to *Redis*. All measurements are conducted on a desktop system (Intel® Core™ i5-2520M CPU @ 2.50GHz, 8 GB RAM, Ubuntu 16.04) by using either a Python client with the respective DBMS' API, or, in case of *HSQLDB*, a Java appli-

doi: 10.2313/NET-2017-05-1_06

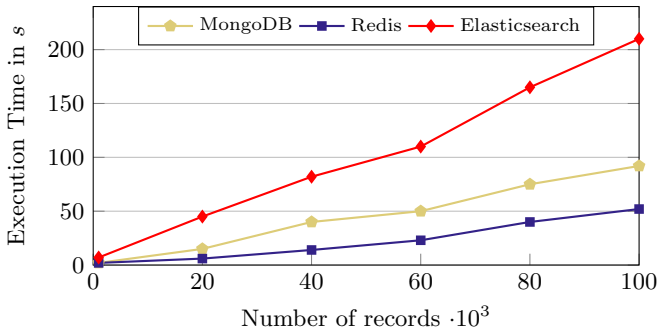


Figure 2: Insert performance of *MongoDB*, *Redis* and *Elasticsearch* (adapted from [1])

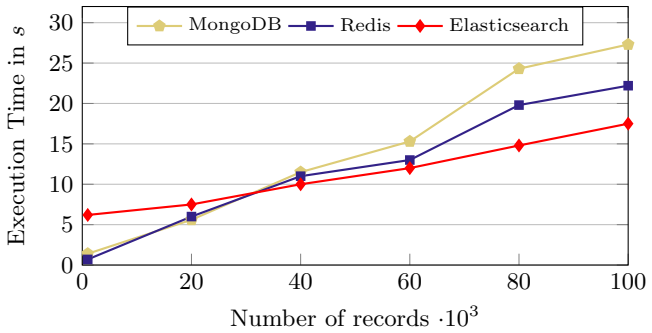


Figure 3: Read performance of *MongoDB*, *Redis* and *Elasticsearch* (adapted from [1])

cation that works like the data access wrapper in DS2OS. The values shown in the subsequent figures are the average out of 5 repetitions for each data set size.

The insertion speed (see Figure 4) is measured for different sizes of data. These inserts show that *HSQLDB* gets slower with larger data sets. The amount of data already in the database did not affect this for any DBMS in our test. As seen in Figure 2, *Redis* performs very well for large data.

Both, reading (see Figure 5) and deleting data (see Figure 6), was measured for different record sizes out of a constant data set of 100000 elements. For *HSQLDB*, we observed a high fluctuation in terms of the performance. Therefore, the number of repetitions for each selection was increased to 10.

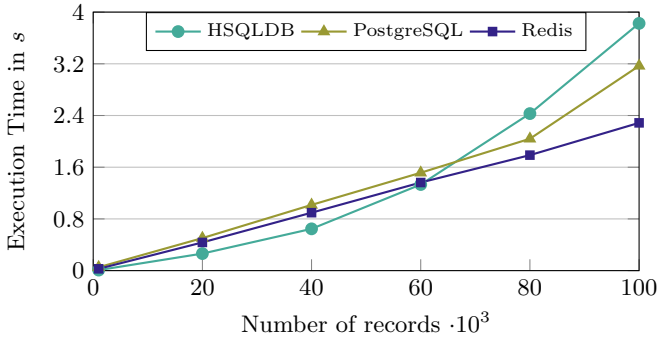


Figure 4: Insert performance of *HSQLDB*, *PostgreSQL* and *Redis*

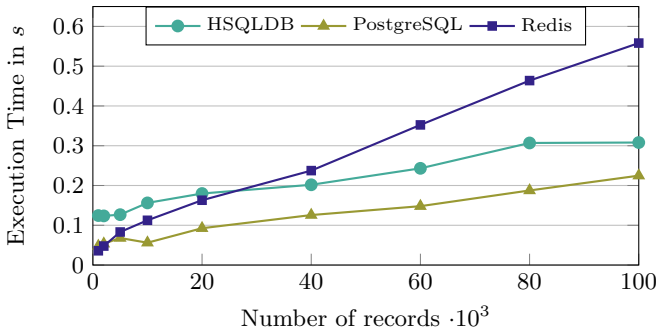


Figure 5: Read performance of *HSQLDB*, *PostgreSQL* and *Redis*

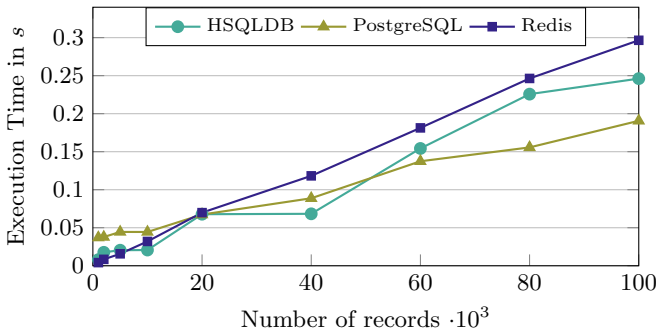


Figure 6: Delete performance of *HSQLDB*, *PostgreSQL* and *Redis*

We observe that *Redis* behaves the most predictable and is linear in regard to the time it takes for increasing data. Besides the high fluctuation in *HSQLDB's* performance, we observe that *Redis* does well for small records up to 5000, as the two RDBMS seem to have an initial overhead. *Redis* is preferable for reading smaller record numbers while *PostgreSQL* is a better overall choice. These differences in behavior based on the size of the data does not allow to put all DBMS into an order for each criteria **C1**–**C3**.

The newer and lesser known *InfluxDB* is compared to *MongoDB* and *Elasticsearch* by two white papers. The comparison with *MongoDB* is based on a data set depicting monitoring data of 100000 different values. Every value is updated every 10 s for 6 h which results in a total of 216 million values. This data is very suitable for time series databases which leads to *InfluxDB* outperforming *MongoDB* by a factor of 27 when inserting. It is also observed that *InfluxDB* needs considerably less disk storage for the data. In terms of reading data, both DBMS performed equally with *MongoDB* getting ahead if concurrency is introduced. [18]

A similar benchmark is conducted for *Elasticsearch*. The data set has 10000 different values updating every 10 s over one day which results in a total of 86.4 million values. For this data, *InfluxDB* is 8 times faster than *Elasticsearch* when inserting it. *InfluxDB* is also faster on querying by a factor of 4 which increases for larger data sets. [17] It is important to note that both papers regarding *InfluxDB* have been released by the developers themselves and are likely to have chosen data sets which are beneficial to *InfluxDB's* architecture.

doi: 10.2313/NET-2017-05-1_06

For *ZeroDB*, there is no quantifiable information on its performance in regard to other databases. However, due to its architecture, it shifts encryption to the client as the database does only handle encrypted data. This way, a single query needs multiple messages, as clients receive an encrypted tree structure, decrypt it and resend a more specific query to get a certain node of the tree. This is repeated until the final value is retrieved. In combination with the Python wrapper for data access from a Java application (see Section 5.2), the DBMS is by design slower than *ZODB* on which it is based on. *ZODB* is able to compete with *PostgreSQL* for very large data stores [4].

6. CONCLUSION

We conclude that the area of NoSQL covers a wide variety of interesting database architectures. In cases like DS2OS where the data is a large set of equally structured data with very little relations, lightweight storing mechanisms like the ones introduced by key-value stores (see Section 3.2.1) or document-oriented databases (see Section 3.2.2) can be used. In our scenario, the data structure can be handled by all categories of databases. By picking popular representatives for each category we are able to narrow down the field and pick suitable candidates for introduction into DS2OS.

Based on their performance and the features they implement in regard to the soft requirements (see Section 4.2), we suggest three candidates worth of further consideration.

PostgreSQL as a more sophisticated and slightly faster alternative to the currently deployed *HSQLDB*. As a RDBMS, it can be introduced with very little changes to the current system.

Redis as a simple key-value store that is very fast on large insert blocks and scalable due to its simple schemaless data storage that inherently supports distributed systems. It is proposed instead of the slightly faster *HSQLDB* as it is more compliant to the soft requirements

InfluxDB as a highly performant solution that implements the versioning as a central concept and looks promising for future improvements as it has barely reached version 1.0 as of now.

7. REFERENCES

[1] Y. Abubakar, T. S. Adeyi, and I. G. Auta. Performance Evaluation of NoSQL Systems using YCSB in a Resource Austere Environment. *International Journal of Applied Information Systems*, 7(8):23–27, September 2014. Published by Foundation of Computer Science, New York, USA.

[2] H. Alzahrani. Evolution of Object-Oriented Database Systems. *GJCST*, pages 37–40, 2016.

[3] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970.

[4] R. Compaan. ZODB Benchmarks revisited. `http://www.upfrontsystems.co.za/Members/roche/where-im-calling-from/zodb-benchmarks-revisited`, Mar. 2008. Accessed: 2016-12-17.

[5] C. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 8 edition, 2003.

[6] DB-Engines Ranking. Popularity Ranking Of Database Management Systems. `http://db-engines.com/en/ranking`, Nov. 2016.

[7] C. Hallenbeck. NoSQL, OldSQL, NewSQL, In-Memory & SAP HANA. `https://blogs.saphana.com/2015/05/19/nosql-oldsql-newsql-memory-sap-hana/`, May 2015. Accessed: 2016-11-18.

[8] G. Harrison. *Next Generation Databases: NoSQL, NewSQL, and Big Data*. Apress, 2015.

[9] R. Hecht and S. Jablonski. NoSQL evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336–341, Dec 2011.

[10] A. Kemper and A. Eickler. *Datenbanksysteme - Eine Einführung*. Oldenbourg, 7 edition, 2009.

[11] N. Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, Feb 2010.

[12] B. Leighton, S. J. D. Cox, N. J. Car, M. P. Stenson, J. Vleeshouwer, and J. Hodge. *A Best of Both Worlds Approach to Complex, Efficient, Time Series Data Delivery*, pages 371–379. Springer International Publishing, Cham, 2015.

[13] B. Leskes. Elasticsearch Versioning Support. `https://www.elastic.co/blog/elasticsearch-versioning-support`, June 2013. Accessed: 2016-12-10.

[14] M.-O. Pahl. *Distributed Smart Space Orchestration*. Dissertation, Technische Universität München, München, 2014.

[15] M.-O. Pahl and G. Carle. Crowdsourced Context-Modeling as Key to Future Smart Spaces. In *Network Operations and Management Symposium 2014 (NOMS 2014)*, May 2014.

[16] M.-O. Pahl, G. Carle, and G. Klinker. Distributed Smart Space Orchestration. In *Network Operations and Management Symposium 2016 (NOMS 2016) – Dissertation Digest*, May 2016.

[17] T. Persen and R. Winslow. Benchmarking InfluxDB vs Elasticsearch for Time-Series. Technical report, InfluxData, Inc., San Francisco, CA, September 2016.

[18] T. Persen and R. Winslow. Benchmarking InfluxDB vs MongoDB for Time-Series Data, Metrics and Management. Technical report, InfluxData, Inc., San Francisco, CA, September 2016.

[19] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[20] S. Tiwari. *Professional NoSQL*. EBL-Schweitzer. Wiley, 2011.

[21] VoltDB, Inc. VoltDB Export – Connecting VoltDB to Other Systems. `https://www.voltdb.com/blog/voltdb-export-connecting-voltdb-to-other-systems`, Aug. 2014. Accessed: 2016-12-11.

P4 Compiler & Interpreter: A Survey

Henning Stubbe
Betreuer: Sebastian Gallenmüller, Dominik Scholz
Seminar Future Internet WS2016–2017
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: stubbe@in.tum.de

ABSTRACT

Software-Defined-Networking (SDN) provides new possibilities to configure and manage large scale networks. However, most SDN control protocols limit the possibilities of this approach since they only provide a limited set of protocols and are understood only by a fraction of the available hardware. The Domain-Specific Language (DSL) P4 was developed to mitigate this issue. Designed with the intent to allow a brief description of a switch's behavior, the aim is to enable network owners to develop their own application oriented software for a programmable switch. To be able to execute the developed description on the hardware either a compiler or an interpreter is required, which translates P4 into an executable program. In the past two years, since P4 has been introduced, different tools emerged. This paper will conduct a survey on the different options available.

Keywords

Compiler, Domain-Specific Languages, P4, SDN

1. INTRODUCTION

With increasing size of network structures the necessity to simplify the management in these networks increases. The introduction of Software Defined Networking (SDN), that is the separation of the function a switch provides into the forwarding and control plane, worked towards this need. The forwarding plane, responsible for deciding which rules are applicable to a packet and then acting accordingly, remains part of the switch. Contrary to that, the control plane, responsible for describing and populating the rules the forwarding plane uses to decide, can be outsourced to a central point. This enables simpler configuration options, since modification of the desired network behaviour requires only the controller to be updated. The control plane then makes the change available in the whole network by deploying them to the forwarding plane. Due to the centralized approach previously costly problems, e.g. the calculation of a spanning tree for the network, can be solved more efficiently since the control plane is aware of the network structure.

A protocol that is used to configure SDN-enable devices is OpenFlow [11]. This open standard describes a set of properties the forwarding and the control plane must have as well as how they should interact with each other. Hence, it is possible to combine arbitrary hardware adhering to this standard. Thus network providers gain more flexibility when combining different hardware into their new centralized managed network. The catch on this approach is that current OpenFlow standard — the OpenFlow Switch Specification 1.5.1 [11] — only describes a fixed set of protocols for which rules the forwarding plane has to obey can be expressed. Those protocols are implicitly defined by the list of supported flow match fields as described in Section 7.2.3.7 of the specification [11]. This introduces the problem that custom inhouse solutions that might be desired by network owners cannot be realized since either a protocol used is not available or the custom protocol stack is not supported by OpenFlow.

Bosshart et al. [4] propose to enable network operators to describe the structure of packets that will traverse their network as well as which rules can be applied to them. This allows using separated forwarding and control planes while overcoming the issue of limited protocol support. As the language P4 which Bosshart et al. [4] introduce is a high-level language, tools are required producing a program executable by the hardware target that should later execute it. Since those tools have to abstract from the specific hardware to the hardware independent P4 language they have to cope with this by either being intended for specific hardware or other means. In Section 2 this paper will provide an introduction to the concepts and ideas of P4. Subsequently available solutions with their take on the challenges and implications of the abstraction done by P4 will be addressed in Section 3. Finally, in Section 4, the paper summarizes the current state in its conclusion.

2. BACKGROUND: P4

The Programming Protocol-Independent Packet Processors language — abbreviated P4 — is a Domain-Specific Language (DSL) intended to be used for description of the package processing capabilities of programmable switches. As a DSL focuses on describing rules that can be applied to packets. This allows P4 programs to be more concise when specifying the behavior of a switch. Targeting network operators, Bosshart et al. phrased the design goals of P4:

1. *Reconfigurability in the field.* Modification of the behavior of a switch must remain possible once the switch is permanently installed. I.e. switch behavior changes, e.g. by publishing new match-action rule entries as allowed by OpenFlow must be possible.
2. *Protocol independence.* P4 attempts to make no assumption on which protocols might be used in which combination. In fact P4 tries to create the possibility

doi: 10.2313/NET-2017-05-1_07

to define and integrate new protocols formats whenever desired.

3. *Target independence.* A program written in P4 cannot require features of special hardware, but instead is usable on any hardware for which a runnable translation of P4 can be created. While P4 imposes requirements on the capabilities of the hardware in general, it does not demand the presence of e.g. a fixed instruction set.

Even though P4 is motivated by the inability of OpenFlow to use custom protocols, it borrows a few concepts to describe the packet processing. (Bosshart et al. [4] suggest, that P4 might even be used as proposal how the next version of OpenFlow could look like.) Processing of packets is done by performing actions on the packets based on values of the header fields. As in OpenFlow the mapping from header field value tuples to actions to perform is created during runtime by a control plane, not at compile time. The processing of packets in P4 can be divided into four major phases:

1. *Parsing of the packet.* When receiving a packet, it first must be translated into a representation that can be processed in the next phases. The parser is based on a finite state machine generated from the underlying P4 program.

2. *Apply match-action table to ingress.* The representation of the received object now entered the ingress pipeline. In this phase it is possible to match on the different header fields of the received packet and execute almost arbitrary rules on it. Additionally, the switch can decide which egress pipeline should later process the packet. For this the P4 program can access additional information — metadata — e.g. on which hardware port the packet arrived. If necessary, the potentially modified packet can be resubmitted to enter the ingress pipeline again.

3. *Apply match-action table to egress.* Similar to the ingress pipeline the egress pipeline allows execution of rules based how the parsed header fields. Note that neither submission to another egress pipeline nor resubmits can be used here.

4. *Deparsing.* To be able to send the packet to the wire it has to be deparsed based on its current state. In $P4_{14}$ the deparser is generated automatically from the parsed object.

Therefore, in order to describe the behavior of a switch in P4 each of these phases must be described. Statements of a P4 programs influence different parts of the processing pipeline. Which program parts impacts which phase is depicted in Figure 1. The parse graph required to specify the parser and deparser in P4 is described by a set of headers that might occur, a header to start parsing with and conditions that express which header is expected next in which case. Figure 2 shows a P4 (version 1.0.3) program which will serve as example in this section. It first describes the fictional header type `ether_t` which consists of an 48 bit long field called `src` — the source of the packet — and a second field

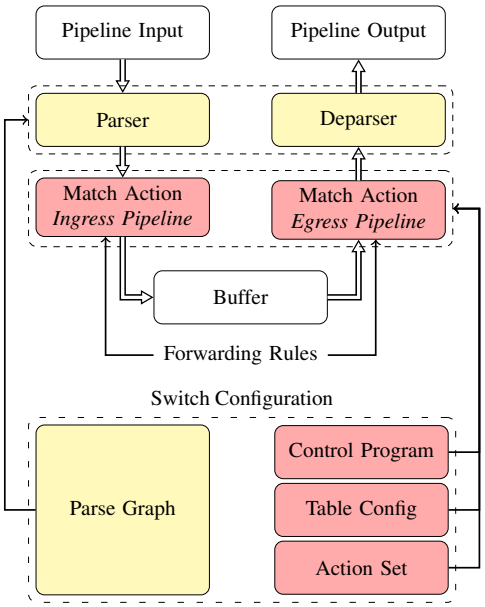


Figure 1: Packet processing as done in P4.

with the same size `dst` — the destination. Then the header named `ether` is declared to be of type `ether_t`. The `parser` lines subsequently instruct the switch to read the outermost header as `ether` and treat the rest of the packet as payload that does not require parsing.

Finally, the packet is added to the ingress pipeline. The remaining part of the program is used to describe the available actions, tables where they occur and which header fields these tables depend on, as well as which tables are part of which pipeline. The field `standard_metadata.egress_spec` describes the port on which the packet should be sent out after deparsing.

Translation tools are required to execute the program, since P4 as a high-level language is not executable. Two options for such a translation exist, either the program is compiled once or interpreted on every execution. While the former method allows a broader range of optimizations the latter

```
header_type ether_t {fields {src: 48; dst: 48;}}
header ether_t ether;
parser start {return ether;}
parser ether {extract(ether); return ingress;}
action forward_ether(out) {
 modify_field(standard_metadata.egress_spec, out);
}
table forward {
 reads {ether.src: exact; ether.dst: exact;}
 actions {forward_ether; drop; no_op;}
}
control ingress {apply(forward);}
control egress {}
```

Figure 2: Discard or forward packet to chosen outgoing port based on source and destination address.

doi: 10.2313/NET-2017-05-1_07

can reduce the time between development and runtime. This makes interpretation more suitable for the development process, and compiling the program preferred for deployment respectively.

The P4 language specification has different versions, which are names with a three digit, dot separated version identifier. Increments in the rightmost digit indicate an update or addition to the last standard. An increase in the middle digit indicates a larger language change. All currently known language version identifier have one as the leftmost digit. Currently the standards 1.0.3 (released 2016–11–03) is said to be the most common, but the standard 1.1.0 (released 2016–01–27) is also available. The main features introduced in 1.1.0 were support for types, strong typing and the ability to state an order in which actions should be executed [17, sec. 17.2]. The P4 version 1.2 has currently draft status. As the current versioning scheme of P4 has caused confusion the versions P4 1.0.0 up to 1.0.3 are referred to as $P4_{14}$, since P4 1.0.0 was released 2014–09–08. The draft P4 1.2 is known as $P4_{16}$.

In the slides of Budiu [6] the differences between $P4_{14}$ and the $P4_{16}$ draft are listed. Additionally the $P4_{16}$ specification draft [18, sec. 3] briefly outlines which change will occur. A major change is that, while $P4_{14}$ specified that the deparser constructed from the given parser, $P4_{16}$ drops this idea and requires the programmer to specify how the object shall be deparsed. This change is motivated by the fact, that there might exist multiple possibilities how the object can be deparsed, e.g. IPv4 in IPv6 or vice versa [17, sec. 6]. A second major difference is that in $P4_{16}$ more targets specific information into P4 by suggestion that a separate file should be used which lists functions provided by the target. Also, one or more core files are envisioned, that shall provide common routines and thus can serve as library for other programs. $P4_{16}$ will also add annotation support and reduce the number of keywords from more than 70 to less than 40. According to aforementioned presentation [6] it will be possible to convert a $P4_{14}$ to a comparable $P4_{16}$ program.

3. COMPILER & INTERPRETER

To receive an executable file from a P4 program either a compiler or an interpreter is required. The next subsections will discuss open source options available wrapped up by a subsection dedicated to proprietary solutions.

3.1 Reference Implementation

To make a reference translator available, the P4 Language Consortium published the P4 compiler `p4c-behavioral` [14]. It will be discussed in the next section in more detail. For a couple of reasons `p4c-behavioral` is now replaced by the "behavioral model" — `bmv2` [13]. The motivation for this change as well as the properties of `bmv2` then follow subsequently.

3.1.1 p4c-behavioral

Developed as first reference compiler for P4 `p4c-behavioral` which is written in C implements all features of the P4 specification. It takes a P4 program as input and generates a valid C program based on this input. To do so it depends on `p4-hlir` — a program to create a High-Level Intermediate Representation of P4 according to the source code given. p4c-hlir [15] is written in Python and provides a target independent P4 parser. On successful parse the result is accessible as Python object hierarchy. By utilizing `p4-hlir`, `p4c-behavioral` can focus on generating correct C code. This code can then be complied for the intended target. As a result, additional optimization by the C compiler is possible. The Apache 2 licensed source code can be found at [14].

After completion of the implementation of `p4c-behavioral` a few issues regarding its design were discovered. It was seen as hassle to be required to compile twice (from P4 to C and from C to binary) in order to produce an executable. Further the generated C code from `p4c-behavioral` was deemed to be hard to understand and thus discouraging people from looking at it. Finally, the underlying switch model in the old compiler is fixed and assumes the presence of two pipelines: one ingress and one egress pipeline. This assumption stands in contrast to the paradigm of P4 not to require any hardware properties. Hence it motivated a change in order to fully support P4.

3.1.2 Behavioral Model

Addressing some issues with `p4c-behavioral` the new so called behavioral model [13], also called `bmv2`, was developed. The behavioral model is, in contrast to the previously introduced `p4c-behavioral` an interpreter. To run a P4 program with `bmv2`, the P4 source code must first be compiled to a JSON file which then, combined with the P4 file, serves as input for the interpreter. The JSON output is generated by `p4c-bm` [16] which also can generate program dependent C++ code. This program dependent code is either an Apache Thrift [1] or an nanomsg [21] based mechanism to enable communication between the control plane and the forwarding plane on the switch.

As `p4c-behavioral`, the new interpreter is released under the Apache 2 license. It fully supports the P4 specification and can be integrated into mininet [12].

3.1.3 P4C

The language P4 is developing and the next version $P4_{16}$ will be released in the future. It is written in C++ and provides different backends. Depending on the P4 input the program can either be compiled to be used with the `bmv2`, or as input for a compiler that can compile C code to eBNF — extended Berkeley Packet Filter. The code released under Apache 2 of [19] is currently rated as alpha quality and thus at the moment is not ready for production.

3.2 P4@ELTE

Similar to the reference implementation of the P4 Language Consortium the compiler P4@ELTE by Laki et al. [9] is not focused on a specific hardware target. Instead, the aim is to provide a target independent compiler. To be able to do so they identified which functions required for P4 are hardware dependent and hardware independent respectively. Providing the independent functions with their compiler the requirements to successfully implement a compiler for a new target is reduced to describe the hardware dependent functionality. This hardware dependent implementation is also

called Hardware Abstraction Layer (HAL). [9] currently provides one implementation of such a Hardware Abstraction Layer that makes use of Intel's DPDK [7] and thus can be used with any network interface which supports C code compatible with said library[1]. For input parsing P4@ELTE makes use of `p4-hlir`. The compilation steps are summarized in Figure 3.

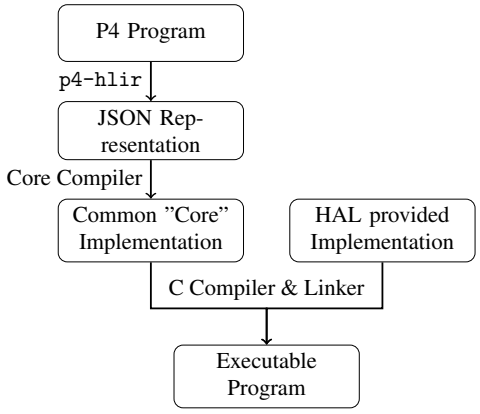


Figure 3: Compilation process of P4@ELTE.

As Laki et al. [8] describe, the approach taken by P4@ELTE to offload the hardware specific details into a library has the benefit that the actual compiler implementation is simpler, as determining properties of the target is not necessary. They argue that the modularity introduced by this separation of concerns increases the maintainability of the compiler. In addition, the fact that the Hardware Abstraction Layer is independent of the P4 programs compiled and thus does not require recompilation if the program changes, is mentioned as advantage of P4@ELTE.

Reduced performance compared to a hardware dependent compiler which can make use of hardware specific optimizations is seen as drawback. In fact, it is difficult to implement any optimization which introduces a constraint on the protocol or hardware used since it may not hold in all cases. The author concludes with the note that implementing the Hardware Abstraction Layer introduces the difficulty to find a suitable abstraction that includes as many hardware targets as possible while avoiding reducing the number of applicable optimizations. The implementation of P4@ELTE was released under the Apache 2 license [10].

3.3 P4FPGA

As the name already suggest P4FPGA [24] differs from the programs mentioned so far regarding the target hardware. The idea of P4FPGA is to provide a tool that eases the development of FPGA based switches. This is done by translating from P4 to Bluespec System Verilog, a hardware description language. For this to work P4FPGA integrates as backend into `p4c`, thus receiving the intermediate representation generated from P4 program parsed by the compiler.

While the full P4 language is supported by P4FPGA, the language P4 itself does not suffice to fully describe how the resulting system should behave. Wang et al. [24] note, the runtime implementation is out of scope for the P4 specification and hence has to be described by other means. This implies that e.g. the description of the Memory Management Unit must be done by other means. The source code of P4FPGA, which is released under the Apache 2 license can be found at [20].

[1] A list of supported network interface controller is maintained at `http://www.dpdk.org/doc/nics`.

3.4 PISCES

While all previous projects focused on hardware targets, the work of Shahbaz et al. [22] — PISCES — focuses on software switches. Bringing to mind that switching between virtual machines hosted on the same physical machine often also involves new or custom protocols, [22] argue that there is a need for an inexpensive way to implement these protocols. Inexpensive in CPU cycles when executed as well as in time to develop and maintain. To address this Shahbaz et al. modified Open vSwitch, introduced by Pfaff et al. [3], to be usable in conjunction with a P4 program. An important aspect in these changes is the effort to remove any protocol dependency — all conclusions based on assumptions how the arriving packet will look like — from the Open vSwitch implementation.

One implication of this is that the concept of micro-/megaflow caches in Open vSwitch has to be revisited. Micro-/megaflow caches utilize the idea that flows of packets resemble each other in certain field, e.g. source and destination address, ports, etc. Since it is no longer ensured that IP addresses or ports fields are present in the header to match on, the justification of this caching mechanism must be verified. After reviewing Shahbaz et al. [22] come to the conclusion, that this matching algorithm does not have to be modified. It is enough to enable the control plane to manage which fields of the header shall be used to match on.

As they were not required for the supported protocols, not all primitives P4 offers were implemented in Open vSwitch and hence had to be added:

- Since protocols described can have any desired header format Shahbaz et al. [22] enhanced Open vSwitch with functions that allow prepending and removal of a header.
- The incremental checksum modification of Open vSwitch was augmented to also support explicit checksum computation, if desired by the programmer.
- Comparison of header fields must be expressed as bitwise equality test, in Open vSwitch. On the other hand P4 enables the programmer to request such tests for fields of arbitrary length. Hence an implementation of that feature, utilizing the available comparison function was implemented to provide that feature.

Some of these additions, e.g. the question of how the checksum should be computed or if inline editing should be used, require knowledge about the protocol which the compiler does not have. To overcome this problem Shahbaz et al. [22] introduced new annotations that allow the developer in these situations to inform the compiler which action should be performed.

Equipped with the modified code they wrote a compiler to generate C code, that makes use of Open vSwitchs functionality, from a P4 program. Compiling those two sources together results in an Open vSwitch derivate with support for an arbitrary protocol stack.

Removing the protocol dependent optimizations from Open vSwitch causes in average a longer execution time per packet, when implementing the same protocol stack as the original version supported. Trying to be competitive with the original Open vSwitch version Shahbaz et al. [22] implemented optimizations to increase the throughput of PISCES:

- With the annotations to the P4 program the compiler can now decide when to compute header checksums or if the use of incremental checksum computation is advised. By delaying or using an incremental computation the compiler can potentially reduce the processing time for the packet.
- When modifying header fields, there are two options. Either the packet is modified inline before all rules are applied, or after all rules are applied post-pipeline. If the modification of the header fields involves changes of the header size, then it might be beneficial to delay the modification to avoid superfluous memory operations. With annotations to the P4 program a developer can influence when PISCES applies modifications to packets.
- The PISCES parser makes use of the knowledge which fields must be parsed to be able to make a forwarding decision and thus is able to reduce parsing time.
- When modifying fields, Open vSwitch can make use of the protocol dependence, i.e. it is known which fields are supposed to remain unchanged and thus do not need to be checked before applying an action. The P4 compiler does not have this knowledge, but can deduce from the P4 program which header fields might have changed and thus need to be checked. This reduction of checks increases the processing speed.
- By analyzing the program the compiler might be able to combine certain modification rules into one.
- Open vSwitch uses its domain specific knowledge to divide its lookup into stages, each stage uses an additional layer of the ISO/OSI stack for the match. While the ordering of layers is not inferable from the program, annotations were introduced that allow to specify an ordering.

When comparing PISCES to Open vSwitch in an extensive benchmarking, Shahbaz et al. [22] come to the conclusion that an implementation of the features of Open vSwitch in P4 was about 40 times shorter (measured in lines of code) than the original implementation with almost the same performance. In their conclusion the authors note, that P4 features that imply state on the target are not implemented, e.g. counter, since this would require a larger modification of Open vSwitchs caching model.

3.5 Proprietary Solutions

Even though P4 is a comparably new language, a couple of companies exist that offer hardware which can be used in combination with P4. The lack of accompanying papers makes it difficult to compare these to the scientific work presented in the previous sections. Nevertheless, it is worthwhile to mention them here, since they as well provide a possibility to use P4 on different targets.

3.5.1 Xilinx SDNet

The company Xilinx, Inc. provides SDNet [5], a development environment which can be used to manage hardware sold by them. To compile a P4 program for their hardware they utilize the `p4-hlir` to parse the source code and make use of the returned python object. A mapper built by them then constructs Xilinx PX code from the result of the previous step, which finally is compiled to firmware by Xilinx SDNet. Xilinx PX is a forwarding plane programming language which, similar to P4 describes the packet processing parsing, one match-action table and deparsing.

3.5.2 Netronome SDK

Netronome indicates that they implemented a compiler that allows execution of programs written in P4 1.0 on Netronome iNIC devices [23]. Extending this compiler to be P4 1.1 compatible is anticipated as effortless by Netronome, since expected syntax changes do not influence the optimization process performed on an intermediate representation of P4.

3.5.3 Barefoot Capilano

A third company providing a P4 compiler for their hardware, called Barefoot Tofino, is Barefoot Networks, Inc. [2]. Barefoot Capilano creates an Integrated Desktop Environment that includes a P4 compiler to create firmware for their hardware.

4. CONCLUSION

In this paper a survey on different methods to translate programs written in P4, a DSL to describe the behavior of programmable switches, was conducted. Each of the programs was discussed regarding the translation process, the translation target as well as unique properties.

Most of the programs available are compilers, but an interpreter intended for development of P4 programs exists. The focus regarding the translation targets are programmable hardware switches with PISCES as notable exception that compiles to a customized Open vSwitch. All implementations support P4 1.0 — the current released version of P4. Apart from `p4c` which is intended to be the reference compiler for the next major version of P4.

It is interesting to see that many of the available tools use the frontend provided by the P4 Language Consortium and hence are able to reduce the effort associated with writing the compiler and interpreter respectively. Providing said frontend might reduces the effort required by developers to support new P4 versions, since provided that the abstract representation remains the same no changes to the backend are required. The question how the different projects cope with the change and how they develop over time might motivate another survey in the future.

5. REFERENCES

[1] Apache Software Foundation. Apache Thrift. https://thrift.apache.org/, 2015. visited 2016–12–21.

[2] Barefoot Networks. The World's Fastest & Most Programmable Networks. Whitepaper, Barefoot Networks, https://barefootnetworks.com/media/white_papers/Barefoot-Worlds-Fastest-Most-Programmable-Networks.pdf, 2016. visited 2016–12–21.

[3] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, Santa Clara, CA, 2015. USENIX Association.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, 2014.

[5] G. Brebner. Programmable Target Architectures for P4. 2nd P4 Workshop by Stanford/ONRC, 2015. http://sched.co/4eqF. visited 2016–12–21.

[6] M. Budiu. Migration guide P4. Technical report, Barefoot Networks, 2016. https://github.com/p4lang/p4c/blob/master/docs/migration-guide.pptx. visited 2016–12–21.

[7] Intel Corporation. Intel Data Plane Development Kit (DPDK), 2016. http://dpdk.org/. visited 2016–12–21.

[8] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel. High-Speed Forwarding: A P4 Compiler with a Hardware Abstraction Library for Intel DPDK. http://p4.elte.hu/publications/p4-ws-2016.pdf, 2016. visited 2016–12–21.

[9] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel. High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 629–630, New York, NY, USA, 2016. ACM.

[10] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel. Retargetable compiler for the P4 language. https://github.com/P4ELTE/p4c, 2016. visited 2016–12–21.

[11] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.5.1*, 2015.

[12] Open Networking Laboratory. Mininet: An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/, 2015. visited 2016–12–21.

[13] P4 Language Consortium. Behavioral Model (bmv2). https://github.com/p4lang/behavioral-model, 2014. visited 2016–12–21.

[14] P4 Language Consortium. p4c-behavioral. https://github.com/p4lang/p4c-behavioral, 2015. visited 2016–12–21.

[15] P4 Language Consortium. p4c-hlir. https://github.com/p4lang/p4-hlir, 2015. visited 2016–12–21.

[16] P4 Language Consortium. Preprocessor for the P4 behavioral model. https://github.com/p4lang/p4c-bm, 2015. visited 2016–12–21.

[17] P4 Language Consortium. *The P4 Language Specification, Version 1.1.0*, 2016.

[18] P4 Language Consortium. *The P4 Language Specification, Version 16 (Draft 2016–12–16)*, 2016.

[19] P4 Language Consortium. p4c. https://github.com/p4lang/p4c, 2016. visited 2016–12–21.

[20] P4FPGA Project. P4 Bluespec Compiler. https://github.com/hanw/p4fpga, 2016. visited 2016–12–21.

[21] A. Roussel, A. Fabijanic, A. Brem, A. Jonsson, A. Starks, A. Santogidis, A. Degtiarov, B. McCroskey, B. Zentner, B. Mitchener, B. Bigras, C. Salzenberg, D. Beck, D. Ochtman, D. Fang, D. Crawford, D. Socolobsky, E. Chevalier, E. R. Berthing, E. Wies, F. S. Mathieu, G. Roberts, G. D'Amore, G. Diethelm, G. Gupta, H. Saito, H. Lieberman-Berg, I. Weber, I. Pechorin, I. Vachkov, J. R. Dunaway, J. Foster, J. Ammous, K. Schiess, K. Lein-Mathisen, L. Barbato, M. Mendez, M. Ellzey, M. Sustrik, M. Howlett, M. Drechsler, M. John, M. Koppanen, N. Desaulniers, N. Hillegeer, N. Soffer, Örjan Persson, O. Timperi, P. Colomiets, P. Kapyshin, R. Brunno, R. Sciuk, R. Killea, R. G. Jakabosky, S. Avseyev, S. Kovalevich, S. Nikulov, S. Velmurugan, S. Strandgaard, S. Mihai, S. Atkins, S. McKay, S. Wallace, T. Besset, T. Peters, V. Guerra, Y. Luo, and Z. Boszormenyi. nanomsg. http://nanomsg.org/, 2016. visited 2016–12–21.

[22] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA, 2016. ACM.

[23] J. Tönsing. P4/PIF + C Programmable Intelligent NICs: Requirements and Implementation Notes. 2nd P4 Workshop by Stanford/ONRC, 2015. http://sched.co/4epr. visited 2016–12–21.

[24] H. Wang, K. S. Lee, V. Shrivastav, and H. Weatherspoon. P4FPGA: High-Level Synthesis for Networking. 2016.

Verifiable Secret Sharing Mechanisms - A Survey

Voggenreiter Markus
Betreuer: Dr. Matthias Wachs
Seminar Innovative Internettechnologien und Mobilkommunikation WS16/17
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: markus.voggenreiter@tum.de

ABSTRACT
In this paper, we show the state of the art technology in key escrow mechanisms. Secure communication over the internet as well as inside an intranet is based on the encryption of sent data. Hence the keys used for encryption are not only precious for the user but also for economical institutions. As soon as these keys are inaccessible the encrypted data can not be recovered, which might result in a loss of essential information. Therefore a way to regain these keys is fundamental for any institution. In this context Key Escrow provides a simple way to provide this desirable infrastructural feature. This paper will introduce several mechanisms to escrow keys and try to give a suitable solution for modern networks. Consequently key escrow in general will be explained and the currently popular mechanisms will be shown divided by the approach used in the infrastructure. These approaches will be explained on protocol layer and checked for the practicability. Finally it should give a conclusion over all protocols.

Keywords
Key Escrow, PKI, IBE, RIKE, CARIBE

1. INTRODUCTION
A fundamental part of modern communication is the security of communication. Therefore almost every entity communicating over the internet or any insecure channel uses cryptography to encrypt their information. Except for the positive outcome of pseudosecure communication, this trend also generates new issues. Assuming that any kind of key was used for protecting the data, all participating entities have the same preconditions for encryption and decryption of data. While the communication is protected, the generated data can only be read with the key intended to do so. This creates a long term problem with accessing all kind of data after the user key is lost. Especially when the data is critical, regaining it is essential.
In an institutional environment, adding, changing and deleting keys is a daily procedure. As soon as a key gets deleted, even though still necessary or an employee leaves, without transferring the decryption keys, the data encrypted with these keys is irrecoverably overwritten. Especially universities or high-tech industry deal with critical data and make use of cryptography for messaging and data transfer. The consequences of not having the ability to decrypt the data again, can result in a loss of money, time but also in legal actions. This problem can be solved by setting up key recovery or key escrow during the generation of such a key. This means the keys can be recovered after they are lost and therefore the encrypted data can be recovered. There are several approaches, which escrow the keys, based on, whether the keys are symmetrical or asymmetrical, which kind of key management is used and which special properties are expected.
Key Escrow in general has quite a bad reputation, since the expression also refers to a means to an end to give governments access to encrypted communication of their inhabitants. Already in 1990 the US government proposed the 'Clipper Chip', which generated a user key from a built in master key and sent the halved user key to two entities [1] . This enabled the access to encrypted data for the relevant US authorities. In modern escrow the underlying principle remained the same, but the Use Case changed. The use of key recovery/escrow in companies is not based on surveillance but on the ability of decrypting critical information encrypted with no longer existing keys.

In this paper several Key Escrow mechanisms will be discussed with focus on asymmetrical encryption in decentralized infrastructures. These will be rated based on several predefined requirements.

2. DEFINITIONS AND REQUIREMENTS
First of all several basic crypto mechanisms and encryption standards must be defined. Modern key management systems base on two major structures, the symmetrical and the asymmetrical encryption. These will be described as well as the criteria, used to benchmark the later presented mechanisms.

2.1 Symmetrical Encryption
Symmetrical encryption is a method to encrypt data sent between two entities with a single key. For this purpose the entities agree on a key first, which later is used to encrypt and decrypt the messages. An example for this would be the blockcipher AES [2], where the data is encrypted with in several rounds with the same key. Since this kind of encryption uses one key for every communication partner, it is not common for a scalable network. Due to this the work focuses on the usage of asymmetrical encryption.

2.2 Asymmetrical Encryption
The other encryption method, the asymmetrical encryption, is in terms of key management contrary to our first method. This technique uses two keys per user, where one key is used

doi: 10.2313/NET-2017-05-1_08

to encrypt data and the second key to decrypt data. In contrast to the symmetrical encryption these two keys are sufficient to communicate with all entities inside the system, since the encryption key of another person is used to encrypt data meant for him. The asymmetrical encryption is the common method for key management in larger systems. For the understanding of this paper the knowledge of basic asymmetrical encryption standards is necessary. Therefore the state-of-the-art techniques RSA and ElGamal will be presented.

2.2.1 RSA

RSA is based on the factoring problem, which results from the generation of the keys[3]. Every entity in the system owns two keys a private key (SK_{entity}) and a public key (PK_{entity}), which are used to encrypt and decrypt any message sent to them. In order to do so, the public key of the user, who should be able to decrypt the message/data must be used. These keys are derived from the following algorithm.
1. $n = p * q$ with $p \neq q$ and p, q are prime
2. $\varphi(n) = (q-1) * (p-1)$
3. e : coprime to $\varphi(n)$ with $e < \varphi(n)$
4. $e * d + k * \varphi(n) = 1$ and derive d from it
5. $PK_{user} = (n, e)$ and $SK_{user} = (n, d)$

The encryption is done with the public key:
1. $ENC(m) = m^e \ mod \ n = c$

The decryption with the private key:
1. $DEC(c) = c^d \ mod \ n = m$

2.2.2 ElGamal

ElGamal is based on the discrete logarithm problem, since it is based on the idea of the Diffie-Hellman protocol. It also relies on the basic idea of private and public keys, but these are generated differently than in RSA.
1. G of order q with generator g
2. $f \in \{1, ..., q-1\}$ with $ggT(f, q) = 1$
3. $h = g^f$
4. $PK_{user} = (G, g, q, h)$ and $SK_{user} = (f)$

The encryption is done with the public key:
1. $m' = m \in G$
2. $y \in \{1, ..., q-1\}$ with $ggT(h, y) = 1$
3. $c_1 = g^y$
4. $c_2 = m' * h^y$
5. $c = (c_1, c_2)$

The decryption with the private key:
1. $c_3 = c_1^f$
2. $m' = c_2 * c_3^{-1}$ with c_3^{-1} as inverse of c_3 in G
3. $m \Leftarrow m'$ with $\Leftarrow$ as reconversion

The ElGamal algorithm is less common in modern system than RSA. Therefore this paper will focus on infrastructures with RSA keys to escrow.

2.3 Requirements

The usage of Key Escrow comes along with several problems and unwanted features. To provide a fair comparison of the different mechanisms the must-have requirements and the good-to-have requirements are presented and explained.

Must-have:
1. The keys may not be sent in clear text. Otherwise it could be accessed by an attacker listening to the network.
2. It must be impossible for the user to use another key for decryption than the one escrowed. The user might send no key at all or doesn't use the key, which makes the escrowed key invalid.
3. It must be possible to generate new keys effortless. This means a new key should not force the whole system to reload.

Good-to-have
1. The key should be stored on more than one entity to reduce the risk of an entity cheating and decrypting the data. This also reduces the risk of a system loss, when one entity is infected.
2. A proof should be generated for the escrowed key, so that every user can verify the recoverability of the key.
3. The mechanism should be capable of being integrated into an existing system.
4. There should solely be the entity in need to escrow and the entity storing the keys. No third party should be required.
5. The entity storing the keys should not able to impersonate the user storing the keys. Since the user stores his private key, used to sign messages the entity is able to sign any message as the user with the private key.
6. Keys should be exchangeable effortless. Whenever a key is compromised the key pair of the user must be replaced. This should be doable as effortless as possible.

3. KEY ESCROW IN CPKIS

The main property of centralized public key infrastructures (CPKIs) is the centrally generated key pair. Whenever a user requires a key the central entity has to generate a key pair and send it to the user. This has to happen over a secure channel, to ensure the integrity and confidentiality of the keys. The key escrow in a CPKI happens directly at the central entity generating the key, depending on how the key was generated.

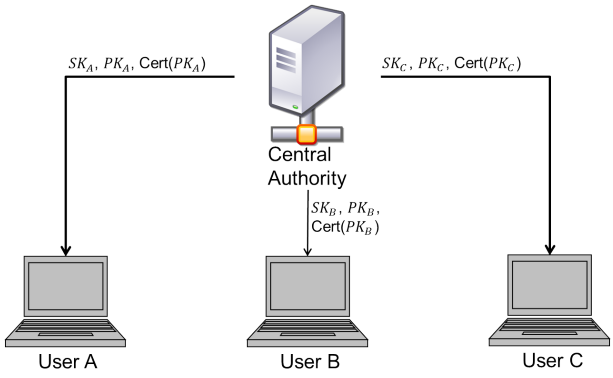


Figure 1: Key distribution in CPKIs

3.1 Private Key Storage

When generating an asymmetric key pair, the relevant part for decrypting data is the private key. Therefore the storage of each user's private key can be a solution. In this scenario the necessary entities are the key database and the key generator. The keys are generated and sent to the database as well as the user. These must be done over a secure channel to provide confidentiality and keep the key secret. In this scenario the key escrow fully relies on the database and the

doi: 10.2313/NET-2017-05-1_08

connection to it. Thus the database is a valuable target, for an attack.

3.2 Master Key Storage

When a master key is in use, the keys are getting computed based mainly on an identifier of the user and the master key. Therefore the key pair can be recovered by re-computing the user keys. This provides a way of key recovery, where only the master key must be escrowed.

3.3 Problem with Key Escrow

The main problem of key escrow in a CPKI is the access of the keys. Since the keys are stored at a single entity, everybody having administrative rights on it has normally access to the keys. Further accessing the keys does not require the cooperation of several entities, which would intercept on a cheating entity, wanting to access the key on its own, but since only one entity can access the key storage, a single entity is sufficient to gain the plaintext of all messages encrypted with the associated keys. Since the central authority generates all keys, it can impersonate any user in the system and read any encrypted message, without the user noticing.

4. KEY ESCROW IN DECENTRALIZED PKIS

Decentralized public key infrastructure(DPKIs) allow every user to generate his own key pair. This makes it, compared to a centralized PKI more flexible and usable, since the users are independent of a key generator and only need to send the public key to the centralized authorities in order to enable the certification, while the private key of the user stays with him. Whereas key escrow in a decentralized PKI is difficult, since the private key must be transmitted to one or more entities in order to escrow it. Therefore in DPKIs no escrow mechanism is part of the basic protocol.

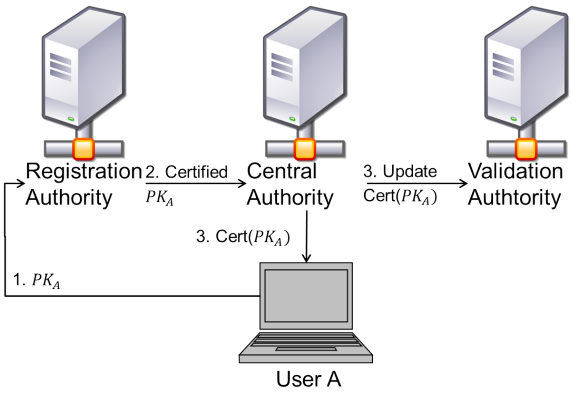


Figure 2: Key distribution/Certification in PKIs

4.1 Fair Encryption of Keys

A practical approach to a publicly verifiable proof that an entity involved in our cryptosystem is able to recover the secret key of our user, was proposed by Poupard and Stern [4]. In their scenario every user owns a pair of keys, from which the secret key is sent to an entity, that is not involved in process of verification or encryption [4, p. 172]. In this context the term "fair encryption" was used for this cryptosystem. Since their protocol operates on asymmetric encryption, especially with modern RSA, it can be used in a decentralized PKI.

4.1.1 Preconditions

To use the proposed "fair encryption" on modern RSA keys several cryptographical assumptions are made. First of all the entity used to store the secret key has a public key pair (N,G) in the Paillier cryptosystem [5]. Second user 1 has a RSA key pair (SK,PK) and wants to escrow SK. Since the user created SK and PK as RSA keys from $n = p * q$ with p and q as prime numbers (usually they must be different, but this protocol does not check the dissimilarity of them). A collision-resistant hash function like H()=SHA-256 must be available on all clients for the proof.

4.1.2 Key Escrow

The conditions for a fair encrypted secret key SK are furthermore a ciphertext Γ and a proof of fairness[4, p. 178]. First the user computes
$x = n - \varphi(n) = p + q - 1$,
where $\varphi()$ is Euler's totient function. To make the encryption probabilistic a random number should be added into the ciphertext. Therefore a random number $u \in \mathbb{Z}_N{}^*$ is selected. Now the Γ is calculated the as follows:
$\Gamma = G^x * u^N mod\ N^2$
Since the public RSA key of our user consists of the RSA modulus n and a number e coprime to $\varphi(n)$, the entity already knows our modulus n. Now the user sends Γ to the entity. The combination of n, Γ and the Parllier secret key of our entity can factor n in q and p and from these factors the private key SK from our user can be computed via
$e * d + k * \varphi(n) = 1$, solve for d[4].

4.1.3 Proof of validity

A main characteristic of this protocol is the non-interactive key storage entity. This means any validation is done by the user itself. The typical proof consists of user 1, who wants to proof the key and user 2, who wants to see, whether the key can be recovered. Thus both users first have to agree on randomly chosen integers
$z_i \in \mathbb{Z}_n^*$ for i=1,...,K.
Then user 1 has to compute the proof:
Choose:
$(r_i)_{i=1,...,l} \in_R [0, A[^l$ and $(v_i)_{i=1,...,l} \in_R \mathbb{Z}_N{}^{*l}$
Generate:
$t = ((G^{r_i} v_i^N mod N^2)_{i=1,...,l}, (z_j^{r_i} mod n)_{i=1,...,l;j=1,...,l}$ and
$(e_1, ..., e_l) = H(t, N, G, (z_j)_{j=1,...,K}, n)$ and $\{\ y_i = r_i + e_i * x$ and $y_i' = u^{e_i} * v_i \mod N\ \}$ for $i = 1, .., l$
Now user 1 sends the proof $((y_i, y_i', e_i)_{i=1,...,l})$. This proof is verified by user 2:
Check $0 \leq y_i < A\ for\ i = 1, ..., l$
Generate:
$t' = ((G^{y_i} * y_i'^N / \Gamma^{e_i}\ mod\ N^2)_{i=1,...,l},$
$(z_j^{y_i - e_i n}\ mod\ n)_{i=1,...,l;j=1,...,K})$
Check $(e_1, ..., e_l) = H(t', N, G, (z_j)_{i=1,...,K}, n)$
If the equality of the last line is given, the key is recoverable by the entity with N and G as public key[4, p. 183].
In this proof several variables were used. Poupard and Stern made a proposal for them in the year 2000. Since then the recommended key sizes increased to a minimum of 2048 [6], these sizes should be adapted. Since we use SHA-256 as

doi: 10.2313/NET-2017-05-1_08

our hash function the values of e are between 0 and 2^{256} . The probability of success in breaking the protocol is $1/B^l$, which would be in our case $1/2^{256l}$. That makes the length l=2 proposed in 2000 still usable. The variable A is a number smaller than n and larger than $x * 2^{256} * 2$. Since the key length of RSA should be 2048, $A < 2^{2048}$. Thus a number greater 2^{1300} can be sufficient. In case of K the used integer has no real effect on the security. In 2000 the value 3 was proposed and since it only changes the amount of random numbers used in the protocol any number higher than 5 should be sufficient[7, compare with].

4.2 Auto-recoverable Cryptosystems

Another way of escrowing keys with a verifiable proof was presented in the year 2000 by Young and Yung. They dealt with the topic of auto-certifiable and auto-recoverable cryptosystems [8] in general and published an implementation of it based on ElGamal [9] in 1999. This was expanded on RSA in 2000. This cryptosystem escrows the private RSA key of a user and gives proof on the recoverability. This scheme works on one escrow agent as well as on a group of escrow agents, since it basically just encrypts the private key. The protocol consists of four steps. The first one is the generation of the RSA key pair. The second one is the generation of the certificate, which provides the data for the recovery to the escrow agents and a proof for anybody else. The third one is the verification of the certificate and the fourth one the recovery of the private key by the recovery agents. In this example the non-interactive version of this protocol is presented.

4.2.1 Preconditions

For the initialisation, the generation, the verification and the recovery of the key several functions need to be defined. These variables might be initialized during the setup of the system and give an overview.
ENC(r,s,E) encrypts the plaintext r using the random value s with the key E.
H(x) is a ideal hash function, hashing x with range Z_n^*.
H"(x) is a random oracle hash function, hashing x.
"E_i" is the public key of the escrow agent i.
"e" is one part of the public key of the user, that wants his key escrowed.
"n" is the RSA module of the public key.
"d" is the private key of the user, that needs to be escrowed.
"$a_i \in_R Z_{\varphi(n)}$" is a random number, chosen by the user.
"$s_{i,1}$ *and* $s_{i,2}$" are the two random numbers for the encryption, chosen by the user.
"t_i *and* t_i'" are variables used during the certification. They either are generated randomly or given by the verifier.
"P" is the computed proof by the user.
$\lambda(n)$ is the Carmichael function.
"m" is the amount of escrow entities.
"k(m)=$\omega(log\ m)$" is the length of the transcript.
"δ" is an additional variable to increase the size of the transcript independent of m [10, p. 329-334].
For the key recovery it is essential, that n consists of two different prime numbers. As protocol to validate this Young and Yung propose [12].

4.2.2 Generation

In this step the RSA keys are created. This is the standard process of RSA key generation, where n is the product of two distinct prime-numbers p and q, e is a number coprime to $\varphi(n)$ and larger 0 and d is the modular multiplicative inverse of $e(mod\varphi(n))$.[10, p. 332] Afterwards the variables e, n and d are initialized.

4.2.3 Key Certification

To construct the proof of escrow, we need the RSA key pair of our user. The output is a proof P, with which the escrow can be verified.[10, p. 330]
1. Initialization
First the initialization takes part, then some randomly based values are generated:
$P = (n), t_0 = H(n)$
$for\ i = 1\ to\ \delta k(m)$:
$t'_{i-1} = H(t_{i-1})\ ;\ t_i = t'^{e}_{i-1}\ mod\ n$
Now the encryption of the data, later relevant for key recovery is done and added to the end of P. In this example the key used for encryption is just E. This could also be a set of keys from several escrow entities $E_1, ..., E_m$[10, p. 331]:
$for\ i = 1\ to\ \delta k(m)$:
$a_i \in_R Z_{\varphi(n)}$;
$s_{i,1}$ *and* $s_{i,2}$ randomly chosen;
$v_i = t_i^{a_i}\ mod\ n$
$C_{i,1} = ENC(a_i, s_{i,1}, E)$;
$C_{i,2} = ENC(d - a_i\ mod\varphi(n), s_{i,2}, E)$;
$add(v_i, C_{i,1}, Ci, 2)$ to P [10, p. 332].
2. Partially Proof
At last the actual "proof" is generated, since the user needs the following to verify the correct encryption of the data, which is made probabilistic by a random oracle.
val = H"(P);
set $b_1, ..., b_{\delta k(m)}$ as the least significant bits of val ($b_i \in \{0, 1\}$)
$for\ i = 1\ to\ \delta k(m)$:
$a_{i,1} = a_i; a_{i,2} = d - a_i\ mod\ \varphi(n)$; $z_i = (a_{i,j}, s_{i,j})$ with $j = 1 + b_i$
add z_i to the end of P
3. Entire Proof
Now the entire proof is generated and has the form:
P = $(n, (v_1, C_{1,1}, C_{1,2}), ..., (v_{\delta k(m)}, C_{\delta k(m),1}, C_{\delta k(m),2})$, $z_1, ..., z_{\delta k(m)})$ [10, p. 333]

4.2.4 Verification

Since the verifier knows
P = $(n, (v_1, C_{1,1}, C_{1,2}), ..., (v_{\delta k(m)}, C_{\delta k(m),1}, C_{\delta k(m),2})$, $z_1, ..., z_{\delta k(m)})$ where $z_i = (a_{i,j}, s_{i,j})$ with $j = 1 + b_i$
he can extract n from P. This implies he can generate $t_1, ..., t_{\delta k(m)}$ from the known public key e and the hash function H(), similar to the key certification. Further he can compute $b_1, ..., b_{\delta k(m)}$,
as he can hash the first part of P:
$P_2 = (n, (v_1, C_{1,1}, C_{1,2}), ..., (v_{\delta k(m)}, C_{\delta k(m),1}, C_{\delta k(m),2}))$ with the function H"(). Now several values are checked:
First of all some general values need to be verified. These are the basic variables used for the generation of the proof.
$t_i \in Z_n^*$ and $a_{i,1+b_i} < n$ for $1 \leq i \leq \delta k(m)$.
Now the actual verification takes place.
$C_{i,1+b_i} = ENC(a_{i,1+b_i}, s_{i,1+b_i}, E)$,
where the values from P_2 are compared with z_i.
$t_i^{a_{i,1+b_i}} = (t'_{i-1}/v_i)^{b_i} v_i^{1-b_i}\ mod\ n$,

doi: 10.2313/NET-2017-05-1_08

where $t_i^{a_{i,1+b_i}}$ can be computed from our first step of verification and the first part of z_i. The second part of the equation is given either just by P_2 or by P_2 and t_i'. As long as these equations and the basic variables are correct the key is correctly escrowed. [10, p. 333].

4.2.5 Recovery

First of all the escrow agents need to decrypt $C_{i,1}$ and $C_{i,2}$, to extract the corresponding plaintext either a_i or $d-a\ mod\varphi(n)$. Now d_i' is computed as the plaintext from $C_{i,1}$ plus the plaintext of $C_{i,2}$. Further K_i is generated via $K_i = ed_i' - 1$. In this way K_i is created for each i from 1 to $\delta k(m)$ [10, p. 333]
Now it is possible to factor n given K_i with a Las Vegas algorithm [11, p. 10]. This algorithm is run on K_i for all i.

5. IDENTITY BASED ENCRYPTION

The basic idea behind IBE, abbreviation for Identity Based Encryption, is to encrypt messages with an identifier instead of a key. This scheme differs in several points from a PKI, but the main advantage of an IBE scheme is the built in escrow mechanism. Therefore the usage of IBE instead of PKI to enable key escrow is a noteworthy option.

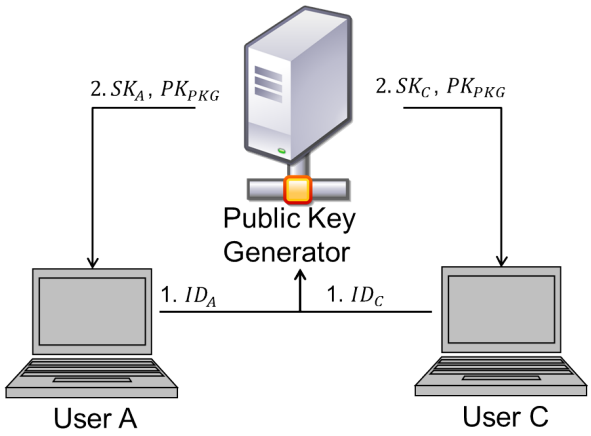


Figure 3: Key distribution in IBEs

5.1 IBE in General

The basic IBE scheme consists of an amount of users n and a private key generator PKG [13]. Each user has an identifier the so called ID. Based on this ID the user can request a corresponding private key from the PKG. The PKG owns a private and a public key, whereas the public key is rather a set of system parameters. For the usage of the encryption scheme (IBE.E) the following four basic algorithms are the framework as described by Boneh and Franklin in 2003:
1. Setup, which the PKG runs once to derive the public and the private key (MK) from a security parameter k, where the public key is generally called "params" meaning system parameter.
2. Extract, which is run by the PKG each time a new user requests a private key and giving the PKG his ID. It takes MK, params and the user's ID to generate a corresponding private key (d), that is given to the user.
3. Encrypt, which is run each time a user wants to send a message encrypted. This algorithm takes the ID of the receiver, the message to send and params to encrypt the message.
4. Decrypt, which the receiver of an encrypted message must run, to decrypt it. This takes the encrypted message, the ciphertext, params and d of the receiver and outputs the plaintext message.[14]
As one may notice the critical spots are the connection between the user and the PKG and the risk of a compromised PKG.

5.2 Key Escrow in IBEs

One of the main features of identity based encryption is the automated key escrow. Since the user gets his private key from the PKG, the PKG has two different strategies how to escrow the private keys of the user.

5.2.1 Key Escrow with Master Key

Due to the fact, that the PKGs private key (MK) is used to generate the users private key, while as only other variables the static values params and ID are used, the private key can be recovered by running the algorithm again. This strategy is the most comfortable, since the amount of data to store and the effort to restore a key is minimum. On the other hand the MK is the single point of failure in the escrow system. When this key is lost, the whole system beneath this PKG can not be escrowed anymore, aside from the fact that if it was stolen, the attacker has full control over the system.

5.2.2 Key Escrow by Storing Keys

To assign the key escrow to a third entity, an option is to store the computed private keys of the user in a database, which is not necessarily on the same system as the PKG. This increases the required storage capacity, keeps the effort at a minimum and distributes the critical data, seen from the side of key escrow, on two systems.

5.2.3 Problems with Key Escrow

The main problem concerning key escrow in IDE schemes is, the access of escrowed keys. Since in both shown ways of escrowing keys, these keys can be accessed by a entity and do not require a cooperation of several entities(compared to PVSS mechanisms). So a special kind of four-eyes principle must be implemented if this kind of security is required.

5.3 Cascade-realized Identity Based Encryption

The basic protocol of IBE consists of one single encryption scheme, used on one PKG. As seen earlier this creates several problems with the security of the key escrow. This issue could be solved by splitting the PKG authority into a group of entities, which was proposed as a cascade realized identity based encryption scheme(CARIBE) by Hale, Carr and Gligoroski [15].
In this scheme n different PKGs cooperate to generate keys, encrypt and decrypt the messages based on possibly n different encryption schemes. This is realized by encrypting the messages under a cascade realisation, where the ciphertext of the first encryption is the plaintext of the second encryption and so on.

doi: 10.2313/NET-2017-05-1_08

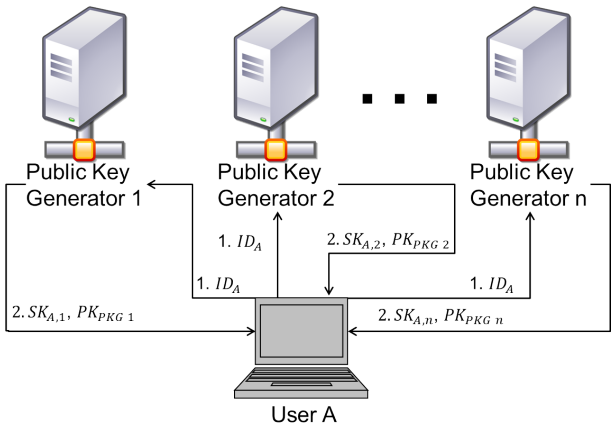


Figure 4: Key distribution in CARIBE

5.3.1 Algorithms

The algorithms are based on the same framework as described in [5.1], but differ in the amount of variables used in every step. Further there can be n different encryption schemes, used during the setup[15, p. 8-9]. The used function calls refer to the specific function used in scheme i, e.g. $Setup_i$ is the function $Setup$ of the encryption scheme i:

1. Setup, where n security parameter (k) are taken and n params and master keys MK_i are generated:
$for\ i \in \{1, ..., n\}$:
$(params_i, MK_i) = Setup_i(k_i)$ end
return $(params_1, ..., params_n\}, \{MK_1, ...MK_n\})$

2. Extract, which takes n params and MKs and the ID of one user and computes the corresponding n private keys d_i:
$for\ i \in \{1, ..., n\}$:
$d_i\ = Extract_i(params_i, MK_i, ID)$ end
return $d_1, ..., d_n$

3. Encrypt, where message m is encrypted using the ID of the user and n params to generate the ciphertext:
$c_2 = Encrypt_1(params_1, ID, m)$
$for\ i \in \{2, ..., n\}$:
$c_{i+1} = Encrypt_i(params_i, ID, c_i)$
end
$c = c_{n+1}$; return c

4. Decrypt, where the ciphertext, n private keys of the receiving user (d_i) and n params are used to produce the plaintext:
$c_n = c$; i=n
$for\ i > 0$:
$c_{i-1}\ = Decrypt_i(params_i, c_i, d_i)$; i=i-1
end
$m = c_0$; return m

As one can see the messages are cascaded en-/decrypted with n keys, n params under n schemes(not necessarily different).

5.3.2 Problems with Key Escrow

The main problem mentioned in [5.2.3] was the access of the keys. This problem has been solved by giving each PKG only access to one layer of encryption. If using the MK of each PKG, n different entities must be requested to decrypt the message. Otherwise if the keys are not stored in a single database, also n databases must be requested for the access. Solely if all user keys are stored in one database the problem keeps remaining. The only missing property would be the ability to regain access to the encrypted data, even though $PKG_x \subset \{PKG_1, ..., PKG_n\}$ is compromised or the keys are lost.

5.3.3 Problems with CARIBE

There are several problems going along with a cascade realisation of IBE. First of all the length of the ciphertext increases, since several encryptions are run successively. This problem was declared as not imposing, as modern IBE schemes are less expanding and the transmission rates are fast enough to deal with this size of data[15, p. 12]. Another problem is the new amount of keys. These must be stored securely at the client, but first of all need to be transmitted to the client. The usage of CARIBE increases the amount of necessary secure channels to transmit data in advance. Since IBE at all does not deal with the problem of key distribution, this is no CARIBE specific problem. Further the performance of en- and decryption is problematic, since these must be run n times. At last cascade cryptography is seen as CCA insecure, as any user can decrypt and re-encrypt the outermost encryption, when in possession of the key. But the possession of this key is only an assumption and not realistic for this case [15, p. 9].

5.4 Risks of IBE

IBE on its own has few considerable security risks. The connection between user and PKG must be highly secure, since the private key of the user is sent over this channel. This general problem of PKG can only be solved based on the use case. Also a compromised key is a problem, since the identifier is the public key and therefore the identifier of this user must be changed in order to grant security, but these identifiers are normally static values(like email addresses). Since the PKG generates all keys, it can on the one hand impersonate every user in his system and on the other hand can decrypt all messages. Because of these enormous privileges, the PKG is a worthwhile attack target. As soon as the PKG is compromised the whole system is compromised and can no longer be used.

5.5 RIKE

A way to integrate IBE into PKI was proposed in 2012[16]. The key management infrastructure was called RIKE and integrates a identity based encryption into a already existing PKI to enable key escrow in the PKI. This is basically done by hashing the user certificate and the value computed in this way is the new public key of our user in the IBE scheme.

5.5.1 Algorithm

The preconditions for implementing RIKE is an existing PKI with the user key pair (PK_u, SK_u) for every user and an CA with identity ID_{ca} and keys, which signs the users public key and returns a certificate $Cert(U) = SIGN_{SK_{ca}}(ID_{ca}, PK_u)$ signed by the private key of the CA. The integration of the IBE requires several values, which are equal to the data used for an IBE. This comprises the PKG, with the system parameters params, the private key of the generator MK and the standard functions Encrypt, Decrypt and Extract. These function are the same as in a basic IBE scheme[16, p.

52].
1. Initialization, where params and MK is generated, similar to [5.1/1.] and signs the new certificate $CERT(CA, params) = SIGN_{SK_{ca}}(ID_{ca}, PK_{ca})$ and adds params as extension[16, p. 62/63]. This certificate is sent to every user, so that every user has params and the certificate of the CA.
2. Key Generation, where the PKG produces the second private key of the user from the hash of the users certificate $SK_{u2} = Extract(MK, params, H(Cert(U)))$ comparable to [5.1/2.]. Now the second, escrowed secret key gets sent to the user. The user now has (SK_u, PK_u), which is the not escrowed pair and the new $(SK_{u2}, (PK_{u2} = H(Cert(U))))$ as the escrowed key pair.
3. Usage, can be done from now on, since the IBE scheme is now implemented. The encryption and decryption works just as in a normal IBE scheme with the escrowed keys, except for the fact that the certificate of the user is checked first. This grants a fast way of revoking compromised keys in the IBE scheme. Also signing can be realized by using the not escrowed private key (SK_u). That solves the problem of the almighty PKG, because it can no longer impersonate any user of the system.[16, p. 52/53]

5.5.2 RIKE based on different structures

The larger the cryptographic environment is, the more entities are needed to manage the users. This tends to result in a increasing number of CAs. To deal with this, RIKE is also compatible to a hierarchical CA structure in [16, p. 54-56]. Usually there are several different CAs cross-signing each other in order to ease the validation of certificates for users. This can also be implemented by RIKE, which is described in [16, p. 56-59].

5.5.3 Key Escrow in RIKE

RIKE builds an IBE inside a PKI, so the key escrow properties apply to the system. But this only applies to the keys generated by the PKG, not to the existing asymmetrical keys. Therefore the use of the first key pair has to be forbidden for encryption to enable full escrow.

6. OTHER KEY ESCROW OPPORTUNITIES

There are further ways of escrowing keys in different cryptosystems. Since every of these schemes are comparable complex to the others given in this paper, they will just be explained briefly.

6.1 SE-PKI

Self escrowing public key infrastructures deal with key escrow by intervening into the key generation at the user. The user generates his key pair decentralized, but with the public part of trapdoor information provided by a central entity. This connects the public and the private key under a master scheme. Then he computes a proof, which everybody can verify. In case of a lost key, the users private key can be recovered by the private part of the trapdoor information by a central authority. The key recoverability can be distributed between several entities.[17]

6.2 PVSS

A rather universal approach are the publicly verifiable secret sharing mechanisms. These mechanisms allow to share a secret among several participants and generates a proof for the recoverability of this secret. [19] This proof can be verified by any user of the system. As mentioned in [18] the usage of these properties enable key escrow in any infrastructure. Instead of a secret, the private key of the user is divided into shares and sent to the escrow entities. Any user of the system can verify the recoverability of the key. As soon as the key must be recovered the entities having the shares must collaborate to recover the key. This is a quite universal solution to the problem of Key Escrow.

7. EVALUATION

This paper presented a solution for different key infrastructures. At last these mechanisms will be compared and rated based on our requirements. The first must-have can not clearly be fulfilled by any mechanism. The transmission of keys must always be done in a secure way, whether using encryption or manual techniques. Solely the Fair Encryption and the auto-recoverable cryptosystems directly refer to this problem, as they send the encrypted private key to the central entity. Therefore this requirement will be seen as a challenge, which must be solved by other means.
1. CPKI
The first two mechanisms used in a centralized key infrastructure fulfil all must-haves, but not all other requirements. They fail in 1, 3 and 5. Since the keys are solely generated centrally 2 is not required. Further the keys can only be exchanged by requesting such a change, but there are no major changes required in the system. All in all this technique fulfils most of our requirements, but the required infrastructure makes it highly impractical and hence limited useful.
2. Fair Encryption
The Fair encryption fulfils all must-haves, since a valid proof requires the usage of the public key. This protocol fails in 1 and 5. Especially the non-interactive proof executed solely by the users of the system makes this protocol convenient. The only drawback is the usage of the Pallier cryptosystem, which is not supported by every system. Therefore this can be integrated into any system, but only with mentionable effort, which makes this method a possible but not the best solution.
3. Auto-Recoverable Cryptosystem
The Auto-Recoverable Cryptosystem also fulfils all our must-haves. Since it is possible to increase the number receiving entities, all of the optional requirements can be fulfilled. Since it also does not require any specific encryption it can be integrated into any system. Since the protocol itself is more than 15 years old, the protocol should be investigated for possible flaws before using it.
4. IBE
The next protocol is the usage of IBE properties. This Escrow mechanism is similar to the first two mechanism, since it uses the centralized generation to escrow the key. So it actually does fulfil all must-haves, but fails in 1, 3, 5 and 6. Changing keys is complex, since the system is based on identifiers. Further it is a completely different infrastructure than the common PKI, which makes it incompatible.
5. CARIBE
The IBE based protocol CARIBE solves several problems of this IBE scheme. It is conform to all our must-haves and

also achieves more optional properties than IBE. It only fails with 3, since the distribution of the keys on different entities solely leaves the problem of compatibility.

6. RIKE

This problem can be solved by the last protocol RIKE, which complies all must-haves, but fails with 1. It gains the compatibility by using an existing PKI, which leaves the problem of a single entity controlling the keys.

8. CONCLUSION

Key escrow in a nutshell is a very complex topic. Not only that most studies try to get rid of key escrow instead of improving it, but modern key escrow often relies on a lot assumptions. On the other hand most key escrow mechanisms can be integrated in every key scheme and is therefore highly usable. Any presented algorithm can be integrated in a public key infrastructure, which is the common cryptosystem. A suitable solution for a PKI system, would be the integration of RIKE in combination with CARIBE. Most mentioned problems would not appear in this line-up and the solely drawback is the management of the different entities and keys.

9. REFERENCES

[1] Han-Wei Liu. 2016. Inside the Black Box: Political Economy Behind the TTP's Encryption Clause, Journal of World Trade, pages 13/14.

[2] Joan Daemen and Vincent Rijmen. 2002. The Design of Rijndael. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[3] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21, 2 (February 1978), 120-126. DOI=http://dx.doi.org.eaccess.ub.tum.de/10.1145/359340.359342

[4] Guillaume Poupard and Jacques Stern. 2000. Fair encryption of RSA keys. In Proceedings of the 19th international conference on Theory and application of cryptographic techniques (EUROCRYPT'00), Bart Preneel (Ed.). Springer-Verlag, Berlin, Heidelberg, 172-189.

[5] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In Proceedings of the 17th international conference on Theory and application of cryptographic techniques (EUROCRYPT'99), Jacques Stern (Ed.). Springer-Verlag, Berlin, Heidelberg, 223-238.

[6] Cryptographic Key Length Recommendation, `https://www.keylength.com/en/8/`, 18.12.2016.

[7] Guillaume Poupard and Jacques Stern. 2000. Short Proofs of Knowledge for Factoring. In Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography (PKC '00), Hideki Imai and Yuliang Zheng (Eds.). Springer-Verlag, London, UK, UK, 147-166.

[8] Adam Young and Moti Yung. 1999. Auto-recoverable Auto-certifiable Cryptosystems (A Survey). In Proceedings of the International Exhibition and Congress on Secure Networking - CQRE (Secure) '99, Rainer Baumgart (Ed.). Springer-Verlag, London, UK, UK, 204-218.

[9] Young, Adam L. and Moti Yung. Auto-Recoverable Cryptosystems with Faster Initialization and the Escrow Hierarchy. Public Key Cryptography (1999).

[10] Adam Young and Moti Yung. 2000. RSA-Based Auto-recoverable Cryptosystems. In Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography (PKC '00), Hideki Imai and Yuliang Zheng (Eds.). Springer-Verlag, London, UK, UK, 326-341.

[11] Hinek, M. Jason. 2009. Cryptanalysis of RSA and its variants. Boca Raton: CRC Press. url=https://books.google.de/books?id=LAxAdqv1z7kC,

[12] Z. Galil, S. Haber, M. Yung. Minimum-knowledge Interactive Proofs for Decision Problems. In SIAM J. of Computing, (4), pages 711?739, 1989.

[13] Clifford Cocks. 2001. An Identity Based Encryption Scheme Based on Quadratic Residues. In Proceedings of the 8th IMA International Conference on Cryptography and Coding, Bahram Honary (Ed.). Springer-Verlag, London, UK, UK, 360-363.

[14] Dan Boneh and Matthew Franklin. 2003. Identity-Based Encryption from the Weil Pairing. SIAM J. Comput. 32, 3 (March 2003), 586-615. DOI=http://dx.doi.org/10.1137/S0097539701398521

[15] Hale, Britta, Christopher Carr and Danilo Gligoroski. CARIBE: Adapting Traditional IBE for the Modern Key-Covetous Appetite. IACR Cryptology ePrint Archive 2015 (2015): 1035.

[16] Nan Zhang, Jingqiang Lin, Jiwu Jing, and Neng Gao. 2012. RIKE: using revocable identities to support key escrow in PKIs. In Proceedings of the 10th international conference on Applied Cryptography and Network Security (ACNS'12), Feng Bao, Pierangela Samarati, and Jianying Zhou (Eds.). Springer-Verlag, Berlin, Heidelberg, 48-65. DOI=http://dx.doi.org/10.1007/978-3-642-31284-7_4

[17] Paillier, P., Yung, M.: Self-Escrowed Public-Key Infrastructures. In: Song, J.S. (ed.) ICISC 1999. LNCS, vol. 1787, pp. 257?268. Springer, Heidelberg (2000)

[18] Adam Young and Moti Yung. 2001. A PVSS as Hard as Discrete Log and Shareholder Separability. In Public Key Cryptography: 4th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2001 Cheju Island, Korea, February 13–15, 2001 Proceedings. Springer Berlin Heidelberg.

[19] Markus Stadler. 1996. Publicly verifiable secret sharing. In Proceedings of the 15th annual international conference on Theory and application of cryptographic techniques (EUROCRYPT'96), Ueli Maurer (Ed.). Springer-Verlag, Berlin, Heidelberg, 190-199.

ISBN 978-3-937201-55-9
DOI 10.2313/NET-2017-05-1
ISSN 1868-2634 (print)
ISSN 1868-2642 (electronic)