HAW HAMBURG

BACHELORTHESIS
Soufian Ben Afia

Erkennung von IP-Spoofing für UDP-basierte Protokolle mittels Rückfragen

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Soufian Ben Afia

Erkennung von IP-Spoofing für UDP-basierte Protokolle mittels Rückfragen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Schmidt
Zweitgutachter: Prof. Dr. Klaus-Peter Kossakowski

Eingereicht am: 10. Februar 2022

Soufian Ben Afia

Thema der Arbeit

Erkennung von IP-Spoofing für UDP-basierte Protokolle mittels Rückfragen

Stichworte

IP-Spoofing, DDos, Datenanalyse, Tshark, UDP Probing, IT-Sicherheit

Kurzzusammenfassung

IP-Spoofing wird heutzutage als Waffe benutzt um vorallem UDP-basierte Protokolle anzugreifen. Es ermöglicht reflection and amplification Angriffe, die derzeitige Internet-Infrastruktur von Services wie Amazon oder Github stark bedrohen. Die Erkennung von falschen Quelladressen würde dazu beitragen, die Situation zu verbessern. Bei UDP handelt es sich um ein verbindungsloses Protokoll, bei dem die IP-Quelladressen (Internet Protocol) nicht überprüft werden. In diesem Arbeit versuchen wir, ein Rückfragemechanismus einzuführen, um IP-Spoofing auf UDP-basierte Protokollen zu erkennen. Dazu haben wir exemplarisch zwei Protokolle, die auf UDP basieren ausgesucht, um den Rückfragemechanismus einzubetten und zu testen...

Soufian Ben Afia

Title of Thesis

Detection of IP spoofing for UDP-based protocols through queries

Keywords

IP-Spoofing,DDos, data analysis, Tshark, UDP Probing, IT-Security

Abstract

IP spoofing is nowadays used as a weapon to attack especially UDP-based protocols. It enables reflection and amplification attacks, which heavily threatens current Internet infrastructure of services like Amazon or Github. The detection of false source addresses would help to improve the situation. UDP is a connectionless protocol that does not

check IP (Internet Protocol) source addresses. In this work, we attempt to introduce a querying mechanism to detect IP spoofing on UDP-based protocols. For this purpose, we have chosen two protocols based on UDP as examples to embed and test the querying mechanism. ...

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

UDP ist ein verbindungsloses Protokoll, das keine IP-Quelladressen überprüft. Sofern das Protokoll der Anwendungsschicht keine Gegenmaßnahmen einsetzt, kann ein Angreifer die Quell-IP-Adresse leicht fälschen [34]. Hinzu besitzt UDP kein Drei-Wege-Handshake wie bei TCP, um eine gültige Verbindung herzustellen. Wenn die Quell-IP-Adresse vieler UDP-Pakete mit der IP-Adresse des Opfers gefälscht wird, kann es zu einem reflektierten DDoS-Angriff kommen, um gefälschte Anfragen zu generieren.

Aus diesem Grunde wird diese Arbeit die Erkennung von IP-Spoofing auf UDP-basierten Protokollen behandeln. Dazu wird ein Rückfragemechanismus eingeführt, um eine Adressvalidierung der Gegenseite durchzuführen. Der Rückfragemechanismus wird exemplarisch anhand QUIC und DNS implementiert, getestet und anschließend evaluiert. Das QUIC-Protokoll wurde bereits mit dem Rückfragemechanismus designt. DNS ist ein typisches Amplifikation-Angriffsziel, das von Hackern mittels IP-Spoofing durchgeführt wird, um gefälschte Anfragen zu generieren.

Die Gliederung der Arbeit teilt sich in sechs Kapitel auf. In Kapitel 2 wird näher auf die notwendigen Grundlagen eingegangen. Im Zuge dessen wird erklärt, was IP-Spoofing ist und die relevanten Arbeiten diesbezüglich werden geschildert, bevor auf die für diese Arbeit besonders relevanten Aspekte und die Eigenschaften der einzelnen Protokolle eingegangen wird. Kapitel 3 schafft ein Verständnis zum Konzept, um den Rückfragemechanismus in den einzelnen Protokollen umzusetzen. Die konkrete Umsetzung des Konzepts wiederum bildet den Gegenstand von Kapitel 4. Darüber hinaus werden die verwendeten Werkzeuge vorgestellt. Anschließend werden im 5. Kapitel die Ergebnisse präsentiert. Mit einem Fazit sowie Ausblick schließt die Arbeit ab.

2 Grundlagen

Im folgenden Kapitel werden Begriffserklärungen sowie ein Überblick über die einzelnen UDP-basierten Protokolle geboten, um eine gute Basis für ein hinreichendes Verständnis von der vorliegenden Arbeit zu erlangen.

2.1 IP-Spoofing

IP-Spoofing bedeutet, dass ein Absender bzw. ein Angreifer die Quell-IP-Adresse eines IP-Paket Headers durch eine gefälschte IP-Adresse ersetzt, um entweder die Identität des Absenders zu verbergen, sich als ein anderes Computersystem auszugeben oder beides. Mit anderen Worten ist ein Paket gespooft, wenn ein IP-Paket mit einer gefälschten Quelladresse oder hinter dem kein Dienst zu jenem Zeitpunkt steht, versendet wird [23].

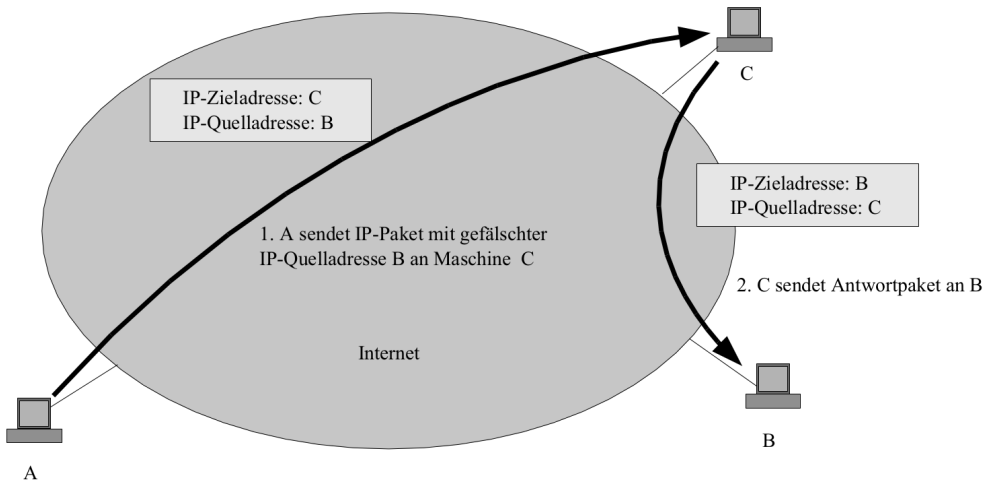

Abbildung 2.1: Reflexionsangriff mittels IP-Spoofing [23].

In Abbildung 2.1 wird der Ablauf eines Reflexionsangriff mittels IP-Spoofing dargestellt. Host A sendet ein IP-Paket mit einer gefälschten IP-Adresse von Host B an Host C. Dabei verbirgt Host A seine Identität und gibt sich als Host B an. Host C empfängt das Paket und wird mit der gefälschten IP-Quelladresse B hinters Licht geführt. Antworten von C werden anstatt nach A an die IP-Adresse B gesendet und erreichen damit A nicht. Der beschriebene Ablauf wird auf Schicht 3 des Referenzmodells der Open Systems Interconnection (OSI) ausgeführt [23]. Weiterhin ist IP-Spoofing aus den folgenden Gründen schwer zu erkennen:

- Wir können den Angreifer auf der anderen Seite des Endpoints nicht daran hindern, uns Pakete mit gefälschten Quelladressen zu senden, außer der ISP des Angreifers entscheidet IP-Adressen zu filtern.

- Das IP-Protokoll spezifiziert keine Methode zur Validierung der Quelle des Pakets. Ein Angreifer kann die Quelladresse beliebig nach seinen Wünschen fälschen.

- Da wir unseren Schwerpunkt auf UDP-basierte Protokolle setzen, ist IP-Spoofing umso schwerer, da UDP ein verbindungsloses Protokoll ist.

Die Implikationen bzw. die Folgen von IP-Spoofing und die Schwierigkeit, es zu erkennen, machen IP-Spoofing gefährlich und somit zu einem Problem. Potenzielle Folgen sind etwa das Umgehen von Firewalls oder DDoS-Attacken wie gegen Github mit 1,7 Tbit/s. Dieser Angriff verwendete einen Amplifikation-Vektor in Memcached UDP Server [25][31].

Es gibt verschiedene Arten von Spoofing, um eine Identität vorzutäuschen. ARP-Spoofing ist bspw. eine Art von Angriff, bei dem ein Angreifer gefälschte ARP-Nachrichten über ein lokales Netzwerk sendet. Dies führt dazu, dass die MAC-Adresse eines Angreifers mit der IP-Adresse eines legitimen Computers oder Servers im Netzwerk verknüpft wird. Unsere Arbeit wird sich allerdings nur auf die Erkennung von IP-Spoofing auf UDP-basierten Protokollen einschränken [23].

2.2 Erkennung von IP-Spoofing

Bestehende Ansätze zum Umgang mit gefälschten IP-Quellen Adressen gibt es bereits und werden im Folgenden kurz angeschnitten. Die Ansätze können als solche in **hostbasierte Methoden** und **routerbasierte Methoden** klassifiziert werden. Die hostbasierten Methoden werden auf einem Host implementiert und zielen darauf, gefälschte Pakete an

einen Host zu erkennen. Die routerbasierten Methoden werden wiederum auf Routern implementiert, um gefälschte Pakete zu erkennen, bevor sie das Ziel erreichen [16].

Der Vorteil der **hostbasierten Methoden** liegt in der einfachen Implementierung auf den Host, ohne dass die gesamte Netzwerkinfrastruktur geändert werden muss. Andererseits liegt der Nachteil darin, dass die gefälschten Pakete erst erkannt werden, wenn sie den Host erreichen und nicht davor. Die **routerbasierten Methoden** sind in ihrer Bereitstellung kompliziert und aufwendig, erkennen jedoch gefälschte IP-Pakete sehr schnell [16].

2.2.1 Hostbasierte Methoden

2.2.1.1 Hop-Count Filtering

Der Ansatz folgt der Analyse von Time-to-Live-Feld im IP-Header. Dabei wird die Hop-Count Information aus dem TTL-Feld des IP-Headers abgeleitet. Es wird ein Mapping zwischen IP-Adressen und deren Hop-Counts erstellt. Der Zustand des Mappings hält der Server, um gefälschte IP-Pakete von legitimen zu unterscheiden. Der Nachteil besteht jedoch darin, dass die Anzahl der Hops aufgrund von Routing-Änderungen variieren kann. Dies führt beim Filtern aller Pakete zu falsch-positiven Ergebnissen. Dagegen ist der Vorteil, dass es auf dem Host einfach zu implementieren ist, ohne die gesamte Netzwerkinfrastruktur zu ändern [38].

2.2.2 Routerbasierte Methoden

2.2.2.1 Network Ingress/Egress Filtering

Im Ingress Filtering werden IP-Quelladressen durch den Internet-Service-Provider aussortiert. Dadurch werden lediglich IP-Adressen in dem eigenen Netz geroutet, die mit der Liste, die auf dem Filter verwaltet werden, übereinstimmen. Egress Filtering steuert hingegen den Datenverkehr, der versucht, das Netzwerk zu verlassen. Mit anderen Worten, es werden nur IP-Adressen nach außen geroutet, die dem eigenen Netz angehören. Die Methode erfordert die Installation der Filterung bei jedem ISP, was es schwer bis unmöglich macht, es zu deployen. Auch wenn das Filtering bei jedem ISP aufgesetzt wäre, können IP-Adressen im LAN weiterhin gespooft werden [17].

2.2.3 IP-Spoofing-Messungen

2.2.3.1 IP-Spoofer-Projekt – CAIDA

Das Spoofer-Projekt, das von CAIDA betreut wird, misst in regelmäßigen Abständen die Fähigkeit eines Netzwerks, Pakete mit gefälschten IP-Quelladressen zu senden und zu empfangen. Dabei werden die Ergebnisse in regelmäßigen Abständen als Berichte und Visualisierungen erstellt. Die Daten des Projekts stammen von freiwilligen Teilnehmern, die einen von CAIDA entwickelten aktiven Probing-Client im Hintergrund laufen lassen [4].

2.3 UDP

Angreifer, die IP-Spoofing verwenden, um ihre Identität zu fälschen, nutzen die Tatsache aus, dass UDP ein zustandsloses Protokoll ist. Dies bedeutet, dass ein Angreifer leicht ein UDP-Anfragepaket fälschen kann, indem die IP-Adresse des Angriffsziels als UDP-Quell-IP-Adresse angegeben wird. Auf der anderen Seite ist TCP zustandsorientiert und erfordert einen 3-Wege-Handshake, um die Verbindung aufzubauen, infolgedessen es erschwert wird, die IP zu fälschen als in UDP. Zweitens besitzt TCP einen Mechanismus wie SYN-Cookies, um IP-Spoofing-Angriffen entgegenzuwirken [34][15].

2.3.1 QUIC

QUIC ist ein verbindungsorientiertes Protokoll, das auf der Grundlage von UDP entwickelt wurde. Es bedarf einer relativ geringen Latenz, um einen Verbindungsaufbau herzustellen [21]. Obwohl QUIC ein verbindungsorientiertes Protokoll und bereits mit Mechanismen designt ist, reflective amplification attacks einzuschränken, wurde QUIC bereits Opfer von Flooding-Attacken, ähnlich wie bei TCP SYN-Floods [29]. Im Durchschnitt ist das Internet vier QUIC-Floods pro Stunde ausgesetzt, und die Hälfte dieser Angriffe findet gleichzeitig mit anderen gängigen Angriffsarten wie TCP/ICMP-Floods statt [29].

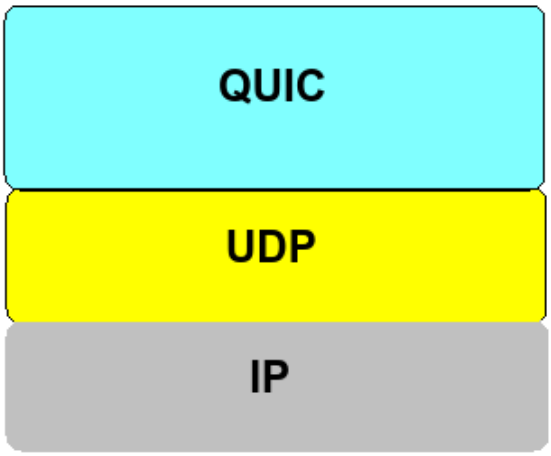

Abbildung 2.2: QUIC Protokoll Stack

Im Gegensatz zu einer Kombination von TCP und TLS, wo jedes Protokoll sequentiell nacheinander seinen Handshake durchführt und zwei Roundtrips verbraucht, integriert QUIC das TLS 1.3 Protokoll in nur einem Roundtrip. Beim Standard TLS + TCP benötigt TCP einen Handshake, um eine Verbindung zwischen Server und Client aufzubauen. TLS benötigt darüber hinaus noch einen eigenen Handshake, um sicherzustellen, dass die Verbindung gesichert ist. Demgegenüber benötigt QUIC nur einen einzigen Handshake, um eine sichere Verbindung aufzubauen [35].

2.3.1.1 QUIC Header

In QUIC kommunizieren Client und Server durch QUIC-Pakete. Letztere enthalten wiederum Frames, die Steuerinformationen und Anwendungsdaten zwischen Endpunkten übertragen. QUIC-Pakete werden in UDP-Datagrammen gekapselt und übertragen. Alle QUIC-Pakete sind entweder in einen Short Header oder in einen Long Header einzuordnen. Eine genauere Beschreibung des Short Headers ist nicht notwendig, da diese Art von Header im späteren Verlauf der Arbeit und für unseren Algorithmus zur Erkennung von IP-Spoofing irrelevant wird [21].

Das Long-Header-Paket wird während des Verbindungsaufbaus benutzt und verfügt – wie in der nachfolgenden Abbildung aufgeführt – über folgende Struktur:

Abbildung 2.3: Long Header Packet Format [21]

```
Long Header Packet {
 Header Form (1) = 1,
 Fixed Bit (1) = 1,
 Long Packet Type (2),
 Type-Specific Bits (4),
 Version (32),
 Destination Connection ID Length (8),
 Destination Connection ID (0..160),
 Source Connection ID Length (8),
 Source Connection ID (0..160),
 Type-Specific Payload (..),
}
```

Wie in Abbildung 2.3 zu sehen ist der Header Form Bit für den Long Header bereits auf 1 gesetzt. Der Payload des Long Headers enthält eine Folge von Frames und können wie eine Art atomare Einheit begriffen werden, die verschickt werden können. Das Header-Format wird nur in Paketen in der Verbindungsaufbauphase verwendet. Anschließend wird in QUIC zum Short-Header-Format gewechselt und entsprechend ändert sich der Header Form Bit zu 0. Das Long Packet Type enthält den Pakettyp und umfasst folgende Pakettypen [21].

Type	Name
0x00	Initial
0x01	0-RRT
0x02	Handshake
0x03	Retry

Tabelle 2.1: Long Header Packet Types [21].

Die Tabelle 2.1 enthält in der linken Spalte den Long Packet Type, bestehend aus zwei Bits. Die Pakettypen Initial und Retry werden für den späteren Verlauf der Arbeit von Bedeutung sein, um den Algorithmus zur IP-Spoofing in QUIC umzusetzen. Die beiden anderen Pakettypen werden nicht weiter behandelt

2.3.1.2 QUIC Frames

QUIC-Pakete können einen oder mehrere Frames enthalten, die in der Payload hintereinander angeordnet sind.

Abbildung 2.4: Generic Frame Format [21]

```
Frame {
  Frame Type (i),
  Type-Dependent Fields (..),
}
```

In Abbildung 2.4 ist das generische Frame-Format zu sehen. Jeder Frame beginnt mit einem Frame-Typ, der seinen Typ angibt, gefolgt von zusätzlichen typabhängigen Feldern. Es gibt in QUIC einige Frames, aber für unseren Fall ist der CONNECTION CLOSE-Frame von besonderer Bedeutung. Ein Host sendet einen CONNECTION CLOSE-Frame vom Typ=0x1d, um seinen Peer zu benachrichtigen, dass die Verbindung geschlossen wird. Im Kapitel 3.2.1 werden wir das genannte Frame in ein Initial-Paket kapseln und für unseren Setup benutzen [21].

2.3.1.3 Initial Paket

Ein Initial-Paket verwendet den Long Packet Type, wie in Tabelle 2.1 beschrieben einen Typ-Wert von 0x00. Es trägt die erste kryptografische Nachricht vom Client zum Server, um den Key Exchange durchzuführen. Es werden ebenso Acknowledgements in beide Richtungen gesendet [21].

Abbildung 2.5: Initial Paket Format [21]

```
Initial Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 0,
  Reserved Bits (2),
  Packet Number Length (2),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Token Length (i),
  Token (..),
  Length (i),
  Packet Number (8..32),
  Packet Payload (8..),
}
```

In Abbildung 2.5 ist der Header Form Bit für das Initial Paket ebenfalls bereits auf 1 gesetzt, da es ein Typ des Long Header ist. Das Token-Feld in dem Initial-Paket wird zuvor durch ein Retry-Paket bereitgestellt, welches der Server generiert. Weiterhin wird die Token-Länge des Token-Felds in Byte angegeben. Dieser Wert ist 0, wenn kein Token vorhanden ist. Der Initial-Pakettyp wird zudem dazu verwendet, um einem vom Server gesendeten Retry-Paket zu antworten; siehe 2.3.1.4. Die Antwort enthält ein Token, der zuvor von einem Retry-Paket bereitgestellt worden ist [21].

2.3.1.4 Retry Paket

Der Server kann eine Adressüberprüfung anfordern, indem es ein Retry-Paket an den Client sendet, das ein Token enthält. Das Retry-Paket verwendet einen Long Packet Header mit ein Typ-Wert von 0x03. Das Format des Retry Paket ist in Abbildung 2.6 zu sehen.

Abbildung 2.6: Retry Paket Format aus [21].

```
Retry Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 3,
  Unused (4),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Retry Token (..),
  Retry Integrity Tag (128),
}
```

Es trägt im Feld **Retry** Token ein vom Server erstelltes Adressvalidierungs-Token. Es wird von einem Server verwendet, der einen Retry durchführen möchte. Wenn ein Client ein Initial-Paket sendet, kann der Server, bevor ein kryptografisches Handshake durchgeführt wird, eine Adressvalidierung von der Quelle anfordern, indem er ein Retry-Paket sendet, das ein Token enthält. Die Quelle muss für jeden Verbindungsversuch höchstens ein Retry-Paket akzeptieren und verarbeiten. Der Client antwortet auf ein Retry-Paket mit einem Initial-Paket, welches das bereitgestellte Retry-Token enthält, um den Verbindungsaufbau fortzusetzen [21]. Im weiteren Verlauf der Arbeit (Kapitel 3.2) wird die Retry-Option in QUIC für unseren Algorithmus zunutze gemacht, um die Adressvalidierung durchzuführen und zu evaluieren.

2.3.1.5 QUIC Handshake

QUIC bietet im Unterschied zu anderen Transportprotokollen mehrere Handshake Varianten an. (siehe Abbildung 2.7).

- Das sogenannte 1-RTT Handshake muss vor dem 0-RTT durchgeführt werden, da die Quelle dem Ziel bekannt gemacht werden muss. Daten können nach einem Roundtrip vom Client zum Server übertragen werden [21][36].

- Das 0-RTT-Handshake, funktioniert nur, wenn zuvor eine Verbindung zu einem Host hergestellt wurde. Daten können bereits sehr früh zu Beginn vom Client zum Server übertragen werden, bevor der Handshake abgeschlossen ist [21].

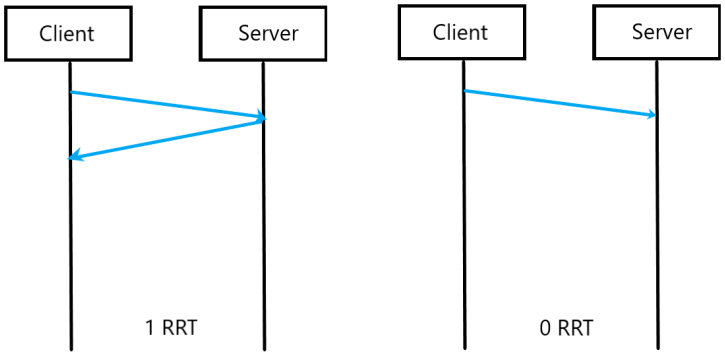

Abbildung 2.7: 1-RRT und 0-RRT Handshakes

In der Abbildung 2.8 sendet der Client am Anfang der Verbindung ein Initial-Paket, das ein TLS ClientHello-Paket enthält. Sofern die enthaltenen Parameter für den Server akzeptabel sind, antwortet er mit einem Initial-Paket, einschließlich TLS ServerHello. Auf diese Nachricht folgt ein Handshake-Paket, das den Rest der TLS-Servernachrichten enthält. Der Handshake endet mit einer Handshake Message vom Client. Anwendungsdaten können nun über die sogenannten 1-RTT-Pakete ausgetauscht werden [21].

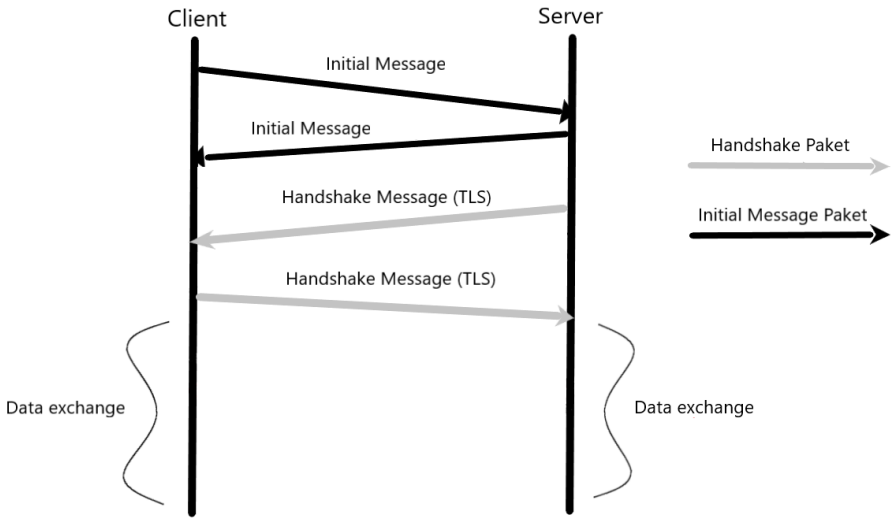

Abbildung 2.8: QUIC 1-RRT Handshake im Detail basierend auf [21].

Im dargestellten Handshake kann ein Angreifer wie bei TCP eine Art SYN-Flooding durchführen. Um solch einer Art von Angriffen entgegenzuwirken, begrenzt QUIC die

Größe der Antworten auf einen Faktor von 3 und validiert nicht verifizierte Clients durch Retry-Pakete.

2.3.2 DNS

Das DNS ist ein verteilter Dienst, dessen Aufgabe darin besteht, eine Namensauflösung durchzuführen, indem Domain-Namen in IP-Adressen und umgekehrt übersetzt werden. Dadurch müssen sich Menschen keine IPv4- oder IPv6-Adressen merken. Weiterhin können Domänennamen einer neuen IP-Adresse zugeordnet werden, wenn sich die IP-Adresse des Hosts ändert [19].

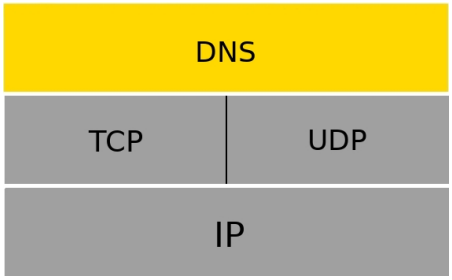

Abbildung 2.9: DNS-Protokoll-Stack: Eigenentwurf

Die Abbildung 2.9 das zeigt DNS-Protokoll-Stack. Der Transport von DNS-Anfragen kann sowohl mit dem verbindungslosen Protokoll UDP als auch mit dem verbindungsorientierten Protokoll TCP erfolgen. Der Server-Port 53 wird sowohl für UDP als auch TCP verwendet, um Anfragen entgegenzunehmen [28]. Generell wird UDP als Transportprotokoll vor TCP zur Übermittlung von DNS-Anfragen bevorzugt. Zum einen bringt TCP weitaus mehr Overhead als UDP und zum anderen muss in TCP ein Verbindungsaufbau durch den 3-Way-Handshake stattfinden, was zu einer höheren Antwortzeit führt.

2.3.2.1 DNS-Message-Format

Die gesamte Kommunikation innerhalb DNS wird in einem einzigen Format übertragen, das als Message bezeichnet wird. Das DNS-Protokoll verwendet zwei Arten von DNS-Nachrichten, nämlich Anfragen und Antworten. Jede Nachricht besteht aus einem Header

und vier Abschnitten, question, answer, authority, und einem weiteren Abschnitt für zusätzliche Informationen. Der Header-Abschnitt steuert den Inhalt dieser vier Abschnitte [28].

Abbildung 2.10: DNS-Message basierend auf [28]

- **Header-Abschnitt:** enthält Felder, die angeben, welche der übrigen Abschnitte vorhanden sind, und ob die Nachricht eine Query oder Answer ist. Auf den Header wird in Kapitel 2.11 genauer erläutert, da ein Flag (TC Flag) im Header von besonderer Bedeutung ist [28].
- **Question-Abschnitt:** enthält Felder, die einen Query an einen Nameserver beschreiben [28].
- **Answer-Abschnitt:** enthält Informationen, die den Question-Abschnitt beantworten [28].
- **Authority-Abschnitt:** enthält Informationen, die auf einen autoritativen Nameserver verweisen [28].
- **Additional-Abschnitt:** enthält weitere Informationen, die sich auf die Query beziehen [28].

2.3.2.2 Header-Abschnitt

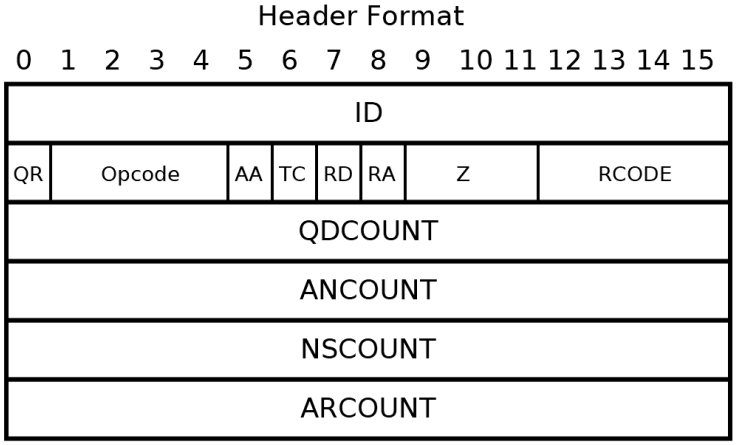

Abbildung 2.11: DNS-Header [28].

Der Header-Abschnitt besteht aus sechs Feldern mit jeweils sechzehn Bits, also insgesamt zwölf Bytes. Die ersten 16 Bits dienen der Transaktion-ID, die dazu dient, die Antwort mit der Abfrage abzugleichen. Die Transaktion-ID wird vom Client in der Query erstellt und vom Server in der Antwort zurückgegeben. Das nächste Feld ist für Flags vorgesehen. Dies ist ein wichtiger Teil des DNS-Headers, da diese Flags die Antwort von der Anfrage sowie die iterative von der rekursiven Query unterscheiden [28].

Die Flags sind folgende:

- **QR:** Query/Response Flag. Wenn 0, ist die Nachricht eine Query. Wenn 1, ist die Nachricht eine Response.
- **Opcode:** Operation code. Ein Vier-Bit-Feld, das die Art der Query in dieser Nachricht angibt. In der Regel bedeutet 0 normale Abfrage, es gibt jedoch auch andere gültige Optionen wie 1 für inverse Query und 2 für Serverstatus.
- **AA:**, Authoritative Answer. Dies gibt an, dass der antwortende Nameserver eine Autorität für den betreffenden Domänennamen ist.
- **TC:** Truncated. Wird gesetzt, wenn das Paket größer ist als die zugelassene Paketgröße.
- **RD:** Recursion desired. Wenn 0, ist die Query eine iterative Query. Wenn 1, ist die Query rekursiv.

- **RA:** Recursion available. Wird bei der Response gesetzt, wenn der Server Rekursion unterstützt.
- **Z:** Reserved for future use. Muss bei allen Abfragen und Antworten auf 0 gesetzt werden.
- **Rcode:** 4-Bit-Feld Response code

Die verbleibenden vier Header-Felder sind die Anzahl der Questions, Response, authority, and additional records. Diese Zahlen variieren je nachdem, ob es sich um eine Query oder eine Antwort und welche Art von Response handelt. Im Allgemeinen wird es jedoch immer mindestens eine Question geben [28].

2.3.2.3 DNS-Komponenten

Um den Prozess hinter der DNS-Auflösung und den Ansatz zur Erkennung von IP Spoofing im DNS zu verstehen, ist es wichtig, die verschiedenen DNS-Komponenten zu kennen, die eine DNS-Anfrage durchlaufen muss.

- **Resolver:** sind Programme, die Informationen von Nameservern als Antwort auf Client-Anfragen extrahieren. Jedoch muss zwischen ein **Stub-Resolver** und **Full-Resolver** differenziert werden. Die Aufgabe des **Full-Resolver** besteht darin, die DNS-Namenshierarchie zu durchlaufen, indem möglicherweise mehrere autoritative Nameserver kontaktiert werden müssen, bis eine Query beantwortet werden kann. Der **Stub-Resolver** hingegen ist eine Komponente, auf die Anwendungsprogramme zugreifen, wenn sie das DNS z. B. für die Auflösung von Domänennamen in IP-Adressen verwenden. Der **Stub-Resolver** dient lediglich als Vermittler zwischen der Anwendung, die eine DNS-Auflösung benötigt, und einem **Full-Resolver** [28] [37] [27].
- **DNS Nameserver:**enthält Informationen über einen Teil des Domänenraums und Pointer auf andere Nameserver, um Informationen aus einem anderen Teil des Domänenraums zu ermitteln. Es gibt verschiedene Kategorien von Nameserver. Der **Root-Nameserver** ist der erste Schritt bei der Auflösung von Hostnamen in IP-Adressen. Es gibt insgesamt 13 **Root-Nameserver**, wobei jeder einzelne **Root-Nameserver** alle Namen und IP-Adressen aller **Top-Level-Domains** (TLDs) enthält. Ein TLD-Nameserver ist der nächste Schritt bei der Suche und verwaltet Informationen für alle Domänennamen, die eine gemeinsame Domänenerweiterung

aufweisen, wie .com oder .net. Der **autoritative Nameserver** bildet die letzte Station in der Nameserver-Abfrage. Wenn der autorisierende Nameserver Zugriff auf den angeforderten Datensatz hat, gibt er die IP-Adresse für den angeforderten Hostnamen an den **Resolver** zurück [28] [37] [27].

Die gängigen Betriebssysteme (Windows, Linux sowie BSD- und UNIX-basierte Systeme) enthalten keinen Full-Resolver. Der typische Prozess zur Namensauflösung in den gängigen Betriebssystemen gestaltet sich wie folgt: [28] [27]

1. Eine Anwendung ruft den Stub Resolver als Funktionsaufruf aus einer Bibliothek heraus.
2. Der Stub-Resolver sendet eine rekursive DNS Query an einen Full-Resolver über das Netz. Falls erforderlich, sendet der Stub-Resolver die Query erneut, bis er entweder eine Antwort erhält oder aufgibt.
3. Der DNS-Server, der als Full-Resolver fungiert, schaut in seinen Cache und führt dann, falls erforderlich, eine Rekursion durch, indem es andere Nameserver fragt, um die DNS-Namenshierarchie durchzulaufen, bis es die Antwort findet.
4. Der Full-Resolver sendet eine Antwort an den Stub-Resolver in Form einer DNS-Nachricht.
5. Der Stub-Resolver gibt den Datensatz an die Anwendung zurück.

3 Entwurf

In diesem Kapitel wird die Methodik zur Erkennung von IP-Spoofing für UDP-basierte Protokolle vorgestellt. Hinzu wird exemplarisch aufgezeigt, wie es in die zwei konkreten UDP-Protokolle, QUIC und DNS, eingebettet wird.

Die entwickelte Methode ist ein reaktiver hostbasierter UDP-Probing-Ansatz, welcher darauf abzielt, mit Paket Probes Antworten zu erlangen, nachdem wir eine Client-Anfrage erhalten haben. Die Paket Probes werden in UDP-Datagramm übertragen, die wir an die Quelle senden. Dadurch möchten wir Antworten aus UDP-basierten Protokollen wie DNS oder QUIC provozieren, damit wir die Echtheit der Quelle validieren können.

Die Methodik kann direkt auf den Hosts implementiert werden, ohne unmittelbar irgendwelche Änderungen auf der Router-Software durchzuführen. Allerdings wird die Herausforderung darin bestehen, speziell gestaltete Pakete an Hosts zu senden und die Antwort zu provozieren.

3.1 IP-Spoofing erkennen mittels Rückfrage

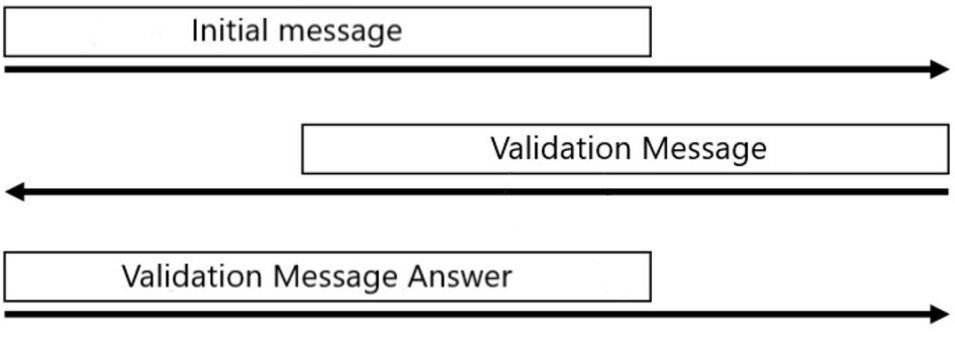

Abbildung 3.1: Allgemeiner Rückfragemechanismus

Die allgemeine Methode wird in der Abbildung 3.1 dargestellt. Im ersten Schritt empfangen wir eine Nachricht von einer beliebigen Quelle, die wir in unserem Fall **Initial Message** nennen. Um die Validierung der IP-Quelladresse durchzuführen, senden wir im zweiten Schritt eine sogenannte **Validation Message**. Die Message wird benutzt, um ein Probing der Gegenseite durchzuführen. Dabei möchten wir eine Rückantwort von der Quelle provozieren. Die **Validation Message** kann je nach Protokoll anders heißen oder aussehen, dennoch bleibt die Aufgabe, dass eine Antwort provoziert werden soll,

Durch das Verfahren möchten wir zumindest die Reaktivität der Gegenstelle testen, um eine Kommunikation zu etablieren. Im besten Falle wäre, wenn wir zusätzlich zur Reaktivität den State der Gegenseite auf Echtheit überprüfen können, sofern die Protokolle es hergeben können. In der Abbildung 3.1 nennen wir die Antwort **Validation Message Answer**, die es daraufhin zu untersuchen gilt.

Wenn die Gegenseite auf die **Validation Message** antwortet, wissen wir, dass es sich um eine echte Quelle handelt, die mit uns zu kommunizieren versucht. Falls die Quelle nicht antworten sollte, könnte es mehrere Gründe haben.

- Das Quell-IP-Adressfeld des Pakets ist gespooft.
- Client ist ausgefallen.
- Client hat sich entschieden, die **Validation Message** zu ignorieren.

Die Einbettung des geschilderten Verfahrens wird für jedes Protokoll in einem eigenen Kapitel ausführlich dargestellt.

3.2 IP-Spoofing Erkennung mit QUIC mittels RETRY

Die QUIC Working Group wusste seit Beginn des QUIC-Projekts, dass QUIC-Server das Ziel von DDOS-Angriffen und IP-Spoofing sein können. Aus diesem Grund spezifizierte das Team im QUIC-Transportprotokoll einen „Retry“-Prozess, welche der Server verwenden kann, wenn Angreifer Attacken antizipieren [21]. So wird im RFC 9000 das Verfahren folgendermaßen geschildert:

> „ A server might wish to validate the client address before starting the cryptographic handshake. QUIC uses a token in the Initial packet to provide address validation prior to completing the handshake. This token is delivered

> to the client during connection establishment with a Retry packet (see Section 8.1.2) or in a previous connection using the NEW TOKEN frame (see Section 8.1.3).“ [21]

Wir werden die Option, welche bereits in dem Kapitel 2.3.1.4 geschildert wurde, benutzen, um den Rückfragemechanismus in QUIC umzusetzen.

3.2.0.1 QUIC Handshake mit Retry

Der Aufbau des Retry-Pakets, welcher unter 2.3.1.4 vorgestellt worden ist, wird im Handshake einer typischen QUIC-Verbindung mit vorheriger Adressvalidierung in Abbildung 3.2 dargestellt.

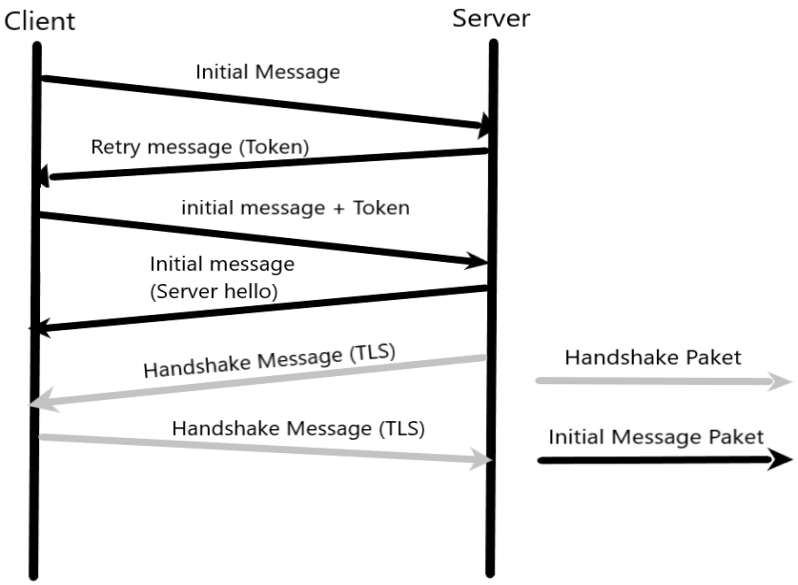

Abbildung 3.2: QUIC Handshake mit Retry basierend auf [21].

Zuerst sendet der Client ein Initial-Paket, das eine TLS 1.3 ClientHello-Nachricht enthält. Um die Quelle zu validieren, antwortet der Server auf das Initial-Paket mit einem Retry-Paket, das ein Token enthält. Sofern der Client mit einer Kopie des Retry-Tokens antworten kann, wird der Handshake fortgesetzt. Der Server antwortet mit einem Initial-Paket, einschließlich des TLS ServerHello. Dieser Nachricht folgt ein Handshake-Paket, das den Rest der TLS-Servernachrichten enthält. Das Handshake-Paket endet mit einer Nachricht vom Client. Ein Angreifer, der eine gefälschte IP-Adresse verwendet, erhält

den Retry Token nicht, infolgedessen kann er keine Rückantwort an den Server senden [21].

3.2.1 Zugeschnittener Handshake für die Adressvalidierung

Aus der Abbildung 3.2 muss für unseren Zweck nicht der vollständige Handshake durchgeführt werden. Die Abbildung 3.3 zeigt den angepassten Handshake.

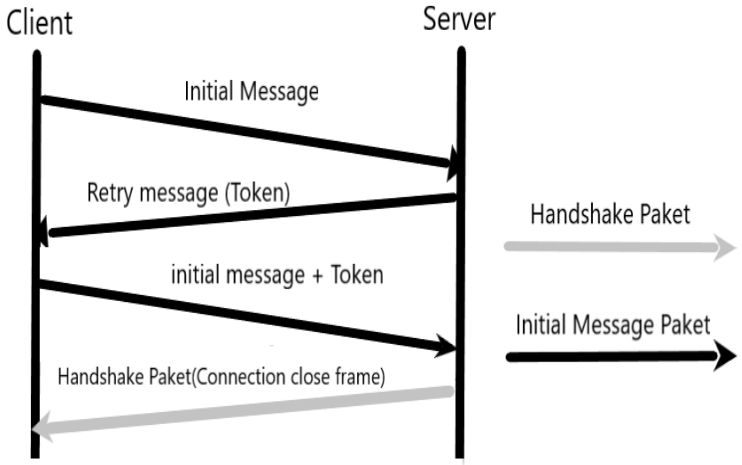

Abbildung 3.3: Angepasster QUIC Handshake mit Retry basierend auf [21].

Für unseren Setup ist lediglich der Retry-Prozess relevant, da die Adressvalidierung vollzogen ist, wenn der Server den kopierten Token von Client enthält und einen Abgleich mit den vom Server erstellten Adressvalidierungs-Token durchführt. Nachdem der Retry-Prozess vollzogen ist, senden wir an den Client ein CONNECTION CLOSE Frame, welches dem Client das Ende der Verbindung signalisiert.

3.3 IP-Spoofing Erkennung in DNS

Das DNS-Protokoll wurde in den letzten Jahren mehrfach ausgenutzt, um DNS-Amplifikation-Attacken durchzuführen. Mittels **IP-Spoofing** wird die Attacke überhaupt erst möglich gemacht und die Antwort auf die IP-Adresse des Opfers gelenkt. Diesbezüglich wurden zwei Ansätze ausgearbeitet, diese werden im Verlauf der vorliegenden Arbeit vorgestellt. Beide Ansätze sehen vor, den vorgestellten Rückfragemechanismus in Kapitel 3.1 auf

ihre eigene Art umsetzen. Der erste Ansatz macht sich den TC Flag aus dem DNS-Header zunutze, um eine Rückantwort zu provozieren [28]. Der zweite Ansatz benutzt die BADCOOKIE-Option aus dem DNS-Cookies-Mechanismus, um ebenfalls eine Rückantwort zu provozieren [1].

3.3.1 Truncation Flag Ansatz

Das Truncation Flag ist ein Binary-Bit-Feld im DNS-Header siehe DNS-Header-Format in Abbildung 2.11.

Wenn die Nachrichtengröße mehr als 512 Byte beträgt, wird das TC Bit (Truncation) im DNS gesetzt, um den Client darüber zu informieren, dass die Nachrichtenlänge die zulässige Größe überschritten hat. Bei EDNS Clients wird das TC Bit gesetzt, wenn eine Antwort größer ist als der vom Client angegebene EDNS-Puffer oder die maximale zulässige Nachrichtengröße von mehr als 4096 Bytes übersteigt. In beiden Fällen kann der Client erneut eine Query über TCP durchführen, das keine Größenbeschränkung hat.

Wir werden genau diese Eigenschaft des TC Flag benutzen, um eine Rückantwort bei dem Client über TCP herbeizuführen. Dabei werden wir das TC Flag in der Response setzen. In den Answer Abschnitt fügen wir noch eine Bogon-Adresse hinzu, um den Client denken zu lassen, dass bereits ein Teil der truncated-Antwort mitgesendet wird.

Der DNS-Response Header würde folgendermaßen aussehen:

<table>
<tr><td colspan="8">ID</td></tr>
<tr><td>QR = 1</td><td>Opcode= 0000</td><td>AA= 1</td><td>TC = 1</td><td>RD = 0</td><td>RA = 1</td><td>Z = 0</td><td>RCODE = 0000</td></tr>
<tr><td colspan="8">QDCOUNT = 1</td></tr>
<tr><td colspan="8">ANCOUNT = 1</td></tr>
<tr><td colspan="8">NSCOUNT = 0</td></tr>
<tr><td colspan="8">ARCOUNT = 0</td></tr>
</table>

Abbildung 3.4: DNS Truncated Response Header

Um einen Truncated Response zu konstruieren wurde BIND 9 als Referenz genommen um die Korrektheit der Response sicherzustellen und keine falschen konstruierten Truncated Pakete zu senden. In der Abbildung 3.4 ist zu erkennen, dass der Header einen Question (QDCOUNT=1) und einen Answer (ANCOUNT=1) Abschnitt enthält. Der Truncated-Bit signalisiert dem Client, dass die Query unter Verwendung TCP erneut zu stellen. Das authoritative Flag signalisiert, dass der Server eine Autorität für die Domäne ist. Operation und Response Code teilen dem Client mit, dass es sich hier um eine Standard Query ohne einen Error Code handelt.

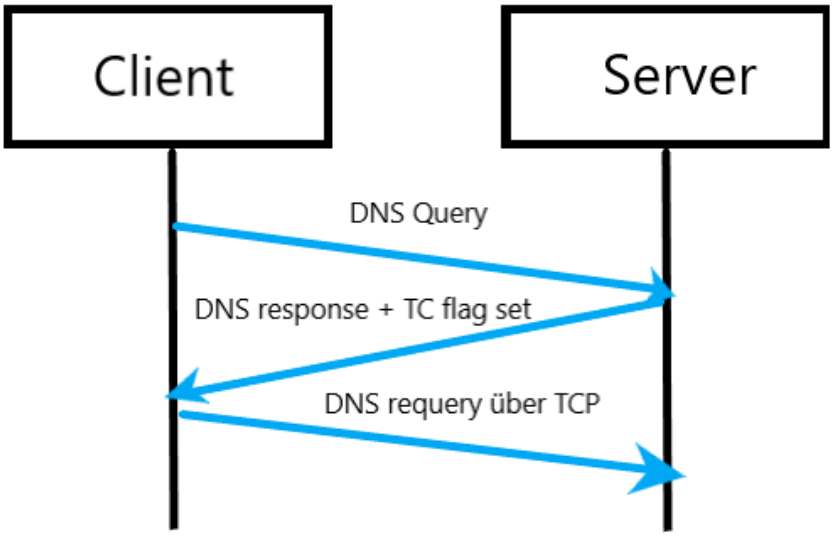

Abbildung 3.5: DNS Verbindung via TC Flag: Entwurf basierend auf [28].

Die Abbildung 3.5 gibt den Verlauf der Kommunikation wieder. Zunächst sendet der Client eine DNS-Query an unseren DNS-Server. Daraufhin erstellen wir eine DNS-Response Message, die den erwähnten Header aus der Abbildung 3.4 enthält. Nun hat der Client zwei Optionen auf unsere Response zu reagieren. Entweder kann der Client den TC Bit ignorieren oder erneut ein Query über TCP durchführen.

Für unseren Versuch ist es wünschenswert, dass die Clients die letztere Option durchführen, damit der Rückfragemechanismus mit diesem Ansatz funktioniert. Die Evaluation in Kapitel 5 wird zeigen, wie erfolgversprechend der Ansatz geworden ist und welche Option die Clients letztendlich selektiert haben.

3.3.2 DNS-Cookies-Ansatz

DNS Cookies wurde mit RFC7873 eingeführt, um den DNS Client als auch den DNS-Server vor Off-path-Attacken zu schützen. Wir werden den Mechanismus benutzen, um IP-Spoofing zu erkennen. DNS Cookies erfordern Unterstützung vom Client als auch vom Server [1].

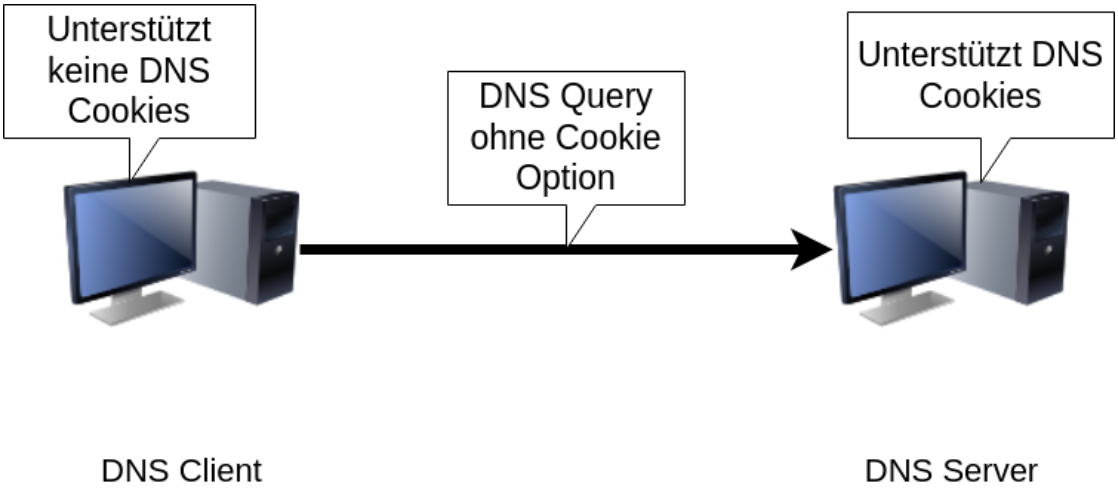

Abbildung 3.6: Client Query ohne Cookie Support. Entwurf basierend auf [1].

Wenn eine Situation wie in Abbildung 3.6 entsteht, in welcher der Cookie Support nur von einer Seite implementiert wird, werden DNS Cookies ignoriert. DNS Cookies werden in DNS-Nachrichten als COOKIE-Option innerhalb des Extended DNS (EDNS) OPT-Resource Record hinzugefügt. Aus diesem Grunde setzen DNS Cookies zwingend für den Client als auch für den Server EDNS-Unterstützung voraus [1] [11].

Sowohl der Client als auch der Server können generierte Cookies in ihren DNS-Nachrichten bereitstellen. Der Client kann dann überprüfen, ob der Server den Client-Cookie in zukünftigen Kommunikationen inkludiert und bereitstellen kann, um sicherzustellen, dass die Pakete nicht von einem Angreifer gefälscht worden sind [1].

DNS-Cookies bieten eine BADCOOKIE-Option, um eine Rückantwort vom Client zu provozieren, wie es der Rückfragemechanismus aus Kapitel 3.1 vorsieht [1].

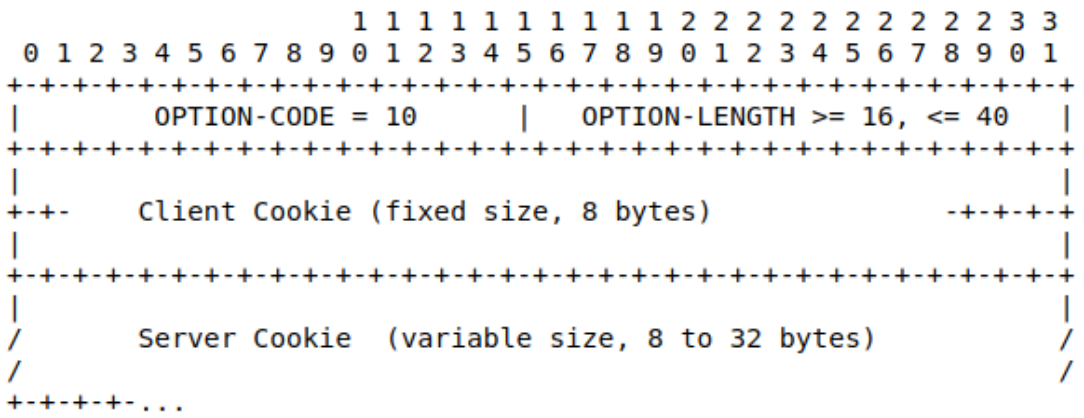

Abbildung 3.7: OPT RR COOKIE OPTION [1].

Client-Cookies haben eine Länge von 8 Bytes, ein Server-Cookie hat eine Größe von 8 bis 32 Byte. Bei einer Query, die von einem Client an einen Server gesendet wird, bei dem der Client den Server nicht kennt, ist der Server-Cookie leer. Ein Server-Cookie wird zur Bestätigung der Identität eines Clients verwendet. Clients sollten daraufhin dieses Cookie in zukünftige Queries hinzufügen, um ihre Identität zu verifizieren. Wenn ein Server eine Query ohne ein Server-Cookie erhält, kann der Server die BADCOOKIE Error Response hinzufügen, um eine Rückantwort vom Client zu provozieren [1].

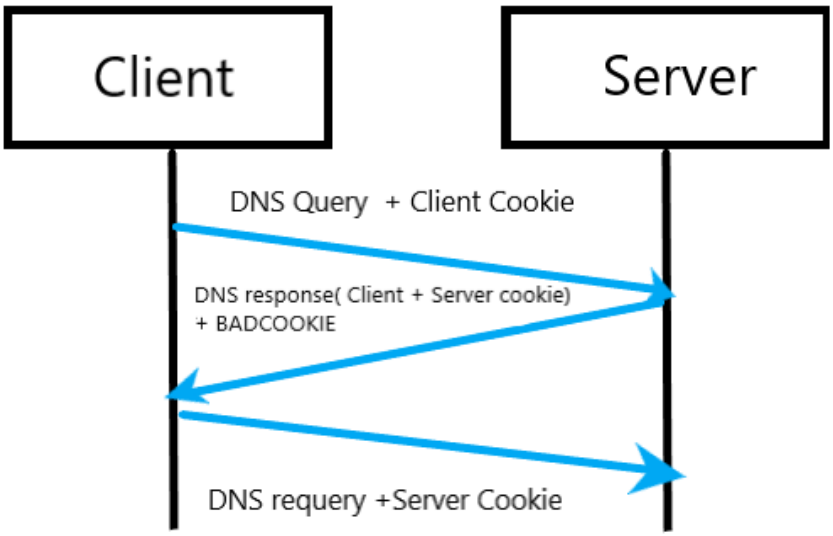

Abbildung 3.8: DNS-Verbindung via DNS Cookies: Entwurf basierend auf [1].

Die Abbildung 3.8 zeigt den Verlauf der Kommunikation mittels DNS-Cookie-Ansatz. Vorausgesetzt DNS Cookies ist auf der Client-Seite **implementiert und aktiviert** dann fügt und sendet der Client im ersten Schritt ein Query mit einer COOKIE-Option. Die COOKIE-Option enthält wiederum nur ein Client-Cookie und kein Server-Cookie, da der Client unseren DNS-Server erstmal nicht kennt. Vorausgesetzt DNS Cookies ist auf der Server-Seite **implementiert und aktiviert,** dann sucht der Server in der Query nach einem Client-Cookie in der COOKIE-Option. Server erstellt darauf hin eine Response mit ein vom Server generierten Servercookie einschließlich der BADCOOKIE Error Response und wird an den Client versendet. Der Client empfängt die Response und sieht ein valides Client-Cookie mit COOKIE OPT-Fehlercode, der anzeigt, dass der Server ein BADCOOKIE inkludiert hat. Als nächsten Schritt sollte der Client sofort mit dem neuen Server-Cookie, das er gerade erhalten hat, erneut ein Query senden.

Allerdings gilt es zu erwähnen, dass im letzten Schritt der Kommunikation dem Client zwei Optionen zur Verfügung stehen:

1. Ignorieren
2. Requery mit Client + Server Cookie

Mit diesem Ansatz hoffen wir, dass die Clients die letztere Option durchführen, damit der Rückfragemechanismus mit diesem Ansatz funktioniert. Ebenso müssen die Clients EDNS und die DNS-Cookie-Option unterstützten. Laut der Analyse aus [12] unterstützen von 93.395 eindeutigen IP-Adressen nur 8.471 (9,1 Prozent) die Client-Cookie-Option. Die Evaluation in Kapitel 5 wird die Ergebnisse aus [12] entweder bestätigen oder widersprechen.

4 Durchführung

Dieses Kapitel stellt die verwendeten Werkzeuge zur Durchführung vor und geht auf die Implementierung des vorgestellten Rückfragemechanismus in Kapitel 3 ein. Die Durchführung der Methodik erfolgt beispielgebend anhand QUIC und DNS.

4.1 Werkzeuge

Nun werden kurz die Werkzeuge und Bibliotheken vorgestellt, die benutzt wurden, um den Entwurf für die zwei Protokolle umzusetzen.

4.1.1 Scapy

Das Scapy-Modul ist eine Python-basierte Bibliothek, die zur Manipulation von Netzwerkpaketen verwendet wird. Im Kontext unserer Arbeit wurde die Bibliothek dazu genutzt gespoofte DNS-Pakete an unseren DNS-Server zu schicken und zu testen [3].

> „Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. It can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks or network discovery (it can replace hping, 85 percent of nmap, arpspoof, arp-sk, arping, tcpdump, tshark, p0f, etc.). It also performs very well at a lot of other specific tasks that most other tools can't handle, like sending invalid frames, injecting your own 802.11 frames, combining technics (VLAN hopping+ARP cache poisoning, VOIP decoding on WEP encrypted channel, ...), etc“ [3]

4.1.2 Dnsjava

Dnsjava ist eine Java-DNS-Bibliothek zur simpleren Durchführung von Queries, Zone Transfers oder Dynamic Updates. Wir haben Dnsjava verwendet, um einen angepassten DNS-Nameserver für unsere Bedürfnisse zu schreiben. Der DNS-Server wurde anschließend benutzt, um den TC-Flag-Ansatz zu testen und zu evaluieren [39].

Die Entwickler der Bibliothek beschreiben Dnsjava wie folgt:

> „Dnsjava is an implementation of DNS in Java. It supports all defined record types (including the DNSSEC types), and unknown types. It can be used for queries, zone transfers, and dynamic updates. It includes a cache which can be used by clients, and an authoritative only server. It supports TSIG authenticated messages, partial DNSSEC verification, and EDNS0. It is fully thread safe. It can be used to replace the native DNS support in Java.“ [39]

4.1.3 RockSaw

RockSaw ist ein Socket-Bibliothek in Java. Ich habe die Bibliothek benutzt, um IP RAW Sockets im Kwik Server zu erstellen [10].

Die Entwickler der Bibliothek haben RockSaw folgendermaßen beschrieben:

> „RockSaw is a simple API for performing network I/O with IPv4 and IPv6 raw sockets in Java. It is the de facto standard API for multi-platform raw socket programming in Java, having been deployed on hundreds of thousands of computing nodes as part of commercial products and custom enterprise applications.“ [10]

4.1.4 Pcap4J

Pcap4J ist eine Java-Bibliothek, die ich verwendet habe, um alle Pakete in einem Interface zu empfangen, unabhängig davon, ob sie an das Interface adressiert sind oder nicht. Dies war vonnöten, um alle möglichen QUIC-Anfragen im Kwik Server empfangen zu können.

Die Entwickler der Bibliothek beschreiben Pcap4J auf folgende Weise:

> „Pcap4J is a Java library for capturing, crafting and sending packets. Pcap4J wraps a native packet capture library (libpcap, WinPcap, or Npcap) via JNA and provides you Java-Oriented APIs.“ [40]

4.1.5 TCPreplay

TCPreplay ist ein Open-Source-Programm zum Bearbeiten und Wiedergeben von zuvor erfasstem Netzwerkverkehr durch PCAP-Dateien mit beliebiger Geschwindigkeit [24]. Ich habe TCPreplay benutzt, um das von mir erfasste Netzwerkverkehr gegen den Server einzuspielen. Auf der offiziellen Homepage wird es folgendermaßen vorgestellt:

> „TCPreplay is a suite of free Open Source utilities for editing and replaying previously captured network traffic. Originally designed to replay malicious traffic patterns to Intrusion Detection/Prevention Systems, it has seen many evolutions including capabilities to replay to web servers.“ [24]

4.1.6 Tcpdump

Tcpdump ist ein verbreiteter Paket-Sniffer für die Kommandozeile. Es wird bevorzugt, wenn beabsichtigt wird, schnell und mit wenigen Handgriffen Pakete aufzuzeichnen. Die Performance von Tcpdump ist am besten für schnelle Scans und die Aufzeichnung von Paketen geeignet. Wireshark hingegen ist stets die erste Wahl für komplexe Scans [18].

4.1.7 TShark

TShark ist ein Tool zum Analysieren von Netzwerkprotokollen. Damit können Pakete aus einem Live-Netzwerk erfasst oder Pakete aus einer zuvor gespeicherten Datei gelesen werden. Das native Capture Dateiformat von TShark ist das pcapng-Format, das zugleich von Wireshark und verschiedenen anderen Tools verwendet wird. Pcap wird unter anderem auch unterstützt [6]. Auf der offiziellen Homepage wird es wie folgt vorgestellt:

> „TShark is a network protocol analyzer. It lets you capture packet data from a live network, or read packets from a previously saved capture file, either printing a decoded form of those packets to the standard output or writing the packets to a file. TShark's native capture file format is pcapng format, which is also the format used by wireshark and various other tools.“ [6]

4.1.8 Wireshark

Wireshark ist eine grafische Anwendung von Tshark. Tshark ist die Anwendung ohne die grafische Benutzeroberfläche [7]. Auf der offiziellen Webseite wird es wie folgt erläutert:

> „Wireshark is the world's foremost and widely-used network protocol analyzer. It lets you see what's happening on your network at a microscopic level and is the de facto (and often de jure) standard across many commercial and non-profit enterprises, government agencies, and educational institutions. Wireshark development thrives thanks to the volunteer contributions of networking experts around the globe and is the continuation of a project started by Gerald Combs in 1998.“ [7]

4.1.9 Nslookup

Nslookup ist ein Befehlszeilentool, mit dem DNS abgefragt werden kann, um die Zuordnung zwischen Domainname und IP-Adresse oder anderen DNS-Einträgen zu erhalten.

> „Displays information that you can use to diagnose Domain Name System (DNS) infrastructure. Before using this tool, you should be familiar with how DNS works. The Nslookup command-line tool is available only if you have installed the TCP/IP protocol.“ [5]

4.1.10 Resolve-DnsName

Resolve-DnsName ähnelt dem Nslookup-Befehlszeilentool, das mit Windows geliefert wird, oder dem Dig-Tool in Linux. Dieses Tool ist Teil des dnsclient PowerShell-Moduls, das mit Windows 10 und weiteren Windows Systemen geliefert wird [26].

> „The Resolve-DnsName cmdlet performs a DNS query for the specified name. This cmdlet is functionally similar to the nslookup tool which allows users to query for names.“ [26]

4.1.11 Dig

Dig (Domain Information Groper) ist ein Befehlszeilentool zur Abfrage von DNS Nameservern. Es können verschiedene DNS-Einträge abgefragt werden, darunter Host-Adressen, Mail-Exchanges und Nameserver. [8] Auf der ISC Homepage wird es folgendermaßen vorgestellt:

> „dig is a flexible tool for interrogating DNS name servers. It performs DNS lookups and displays the answers that are returned from the name server(s) that were queried. Most DNS administrators use dig to troubleshoot DNS problems because of its flexibility, ease of use, and clarity of output. Other lookup tools tend to have less functionality than dig.“ [8]

4.1.12 Kwik Server

Kwik ist eine Implementierung des QUIC-Protokolls in Java. Es unterstützt sowohl Client als auch einen Server mit einer Retry-Option. Weiterhin unterstützt es die offizielle QUIC v1 (RFC 9000) sowie einige ältere IETF-Drafts: Draft-32, Draft-31, Draft-30 und Draft-29 [14]. Der Kwik-Server wurde hier benutzt, um die Retry-Option in QUIC zu testen. Es gibt eine Reihe von experimentellen Implementierungen neben Kwik. Allerdings habe ich Kwik ausgewählt, da ich bereits mit der Java-Syntax vertraut war und es dadurch einfacher fiel, die Code-Modifizierung an dem Server für meine Zwecke durchzuführen.

4.1.13 BIND9

BIND9 wurde als ein DNS-Nameserver für unsere Versuchszwecke benutzt, um den DNS-Cookie-Ansatz zu testen und zu evaluieren [9].

> „BIND 9 has evolved to be a very flexible, full-featured DNS system. Whatever your application is, BIND 9 probably has the required features. As the first, oldest, and most commonly deployed solution, there are more network engineers who are already familiar with BIND 9 than with any other system.“ [9]

4.1.14 Quiche Client

Quiche ist eine Rust Open-Source-Implementierung des QUIC-Transportprotokolls [33]. So stellen die Entwickler Quiche vor:

> „Quiche is an implementation of the QUIC transport protocol and HTTP/3 as specified by the IETF. It provides a low level API for processing QUIC packets and handling connection state. The application is responsible for providing I/O (e.g. sockets handling) as well as an event loop with support for timers.“ [33]

Da der Kwik-Client an Performance mangelte und noch nicht ausgereift gewesen war, wurde stattdessen der deutliche performantere Quiche-Client bevorzugt.

4.2 Umsetzung in QUIC

In diesem Unterkapitel wird der Retry-Mechanismus in QUIC im lokalen Netzwerk getestet und validiert. Denn es muss praktisch in Erfahrung gebracht werden, ob unser Kwik Server die korrekten Retry-Pakete baut und die Antworten im lokalen Netzwerk provozieren kann. Anschließend wird der Kwik Server im Internet deployt, um zu sehen, wie die QUIC Clients im Internet auf die Retry-Anfragen reagieren. So gilt es zu testen, ob eine Sache im lokalen Netzwerk und anschließend ebenso im gesamten Internet funktioniert. Die gesammelten Daten werden in Kapitel 5 ausgewertet.

4.2.1 Testaufbau im lokalen Netzwerk

Um das Konzept für QUIC durchzuführen, habe ich das Experiment in einem lokalen Netzwerk aufgesetzt. Der Aufbau ist nachfolgend aufgeführt.

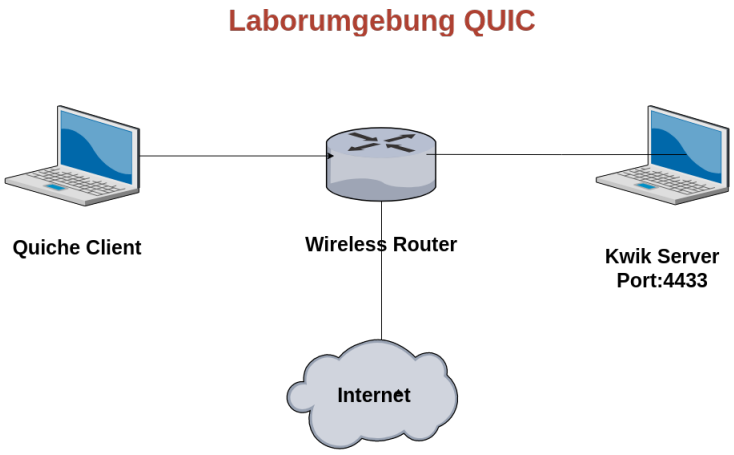

Abbildung 4.1: LAN Netzwerk für QUIC Experiment

Der Quiche QUIC Client wurde auf einem Laptop mit Lubuntu, eine leichtgewichtige Linux-Distribution, aufgesetzt. Dies geschah in folgenden Schritten:

- Quiche benötigt Rust 1.53 oder höher [33]. Die neueste stabile Rust-Version wurde mit rustup installiert. Rustup ist ein Installationsprogramm für die Programmiersprache Rust [13].

- Nach Einrichtung der Rust-Build-Umgebung wurde der Quiche-Quellcode mit Cargo gebaut. Cargo ist der Rust-Paketmanager, welcher ermöglicht, die verschiedenen Abhängigkeiten zwischen Rust-Paketen zu bauen.

Der Kwik Server wurde auf einem anderen Laptop ebenfalls mit Lubuntu, einer leichtgewichtigen Linux- Distribution, aufgesetzt. Hierfür wurden folgende Schritte durchgeführt, um den Kwik Server aufzusetzen:

- Kwik ist eine Implementierung des QUIC-Protokolls in Java. Hierfür musste JDK installiert werden, damit die Ausführung von Kwik auf dem Laptop ermöglicht wird.
- Mittels Gradle wurde der nötige Quellcode in IntelliJ IDEA für den Kwik Server kompiliert [22].

Ich habe den Kwik Server für unseren Fall so modifiziert, dass der Kwik Client automatisch den Retry-Mechanismus startet, falls eine Verbindungsanfrage von einer Quelle kommt. Der Codeschnipsel zur Aktivierung der Retry-Option in Kwik ist in Abbildung 4.2 zu sehen.

```
/**
 * QUIC server.
 */
public class Server implements ServerConnectionRegistry {

    private final boolean requireRetry = true;
```

Abbildung 4.2: Aktivierung der Retry-Option

4.2.1.1 CSVlogger

Zusätzlich wurde ein CSVlogger hinzugefügt, um Daten für die Evaluation zu sammeln und auszuwerten. Er loggt die nötigen Informationen wie Timestamp der Retry-Anfrage als auch die Ankunft der Retry-Antwort, die IP-Quelladresse, den Port sowie weitere wichtige Informationen.

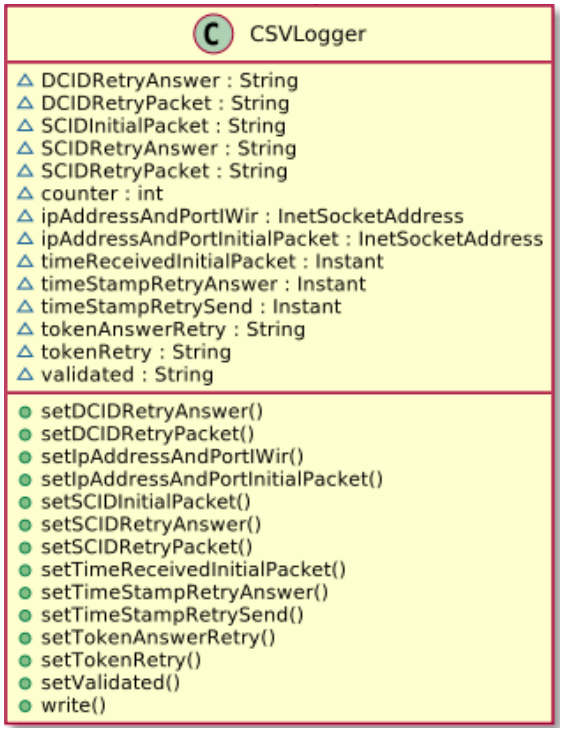

Abbildung 4.3: CSV Logger für Kwik Server

4.2.1.2 Generierung von QUIC-Anfragen

Im ersten Schritt wurden 6000 valide Anfragen mit ein Bash Script generiert. Dabei wurde der Quiche Client jedes Mal neu gestartet. Die 6000 Anfragen werden in Kapitel 5 genutzt, um die Antwortzeiten zu messen. In der Abbildung 4.4 ist der Code Snippet dargestellt, um die Anfragen zu erzeugen.

Abbildung 4.4: Bash Script zur Erzeugung von Quiche Anfragen

```
for i in $(seq 1 6000); do
    cargo run --manifest-path=tools/apps/Cargo.toml --release --bin
    quiche-client -- https://192.168.188.40:4433/resource --no-verify
    sleep 1
done
```

Jede QUIC-Anfrage wurde in Form eines Initial-Pakets, welches in Kapitel 2.3.1.3 vorgestellt worden ist durch den Quiche Client erzeugt und an den Kwik Server auf Port

4433 versendet. Das Aufbau des Initial-Pakets ist der nachfolgenden Abbildung zu entnehmen.

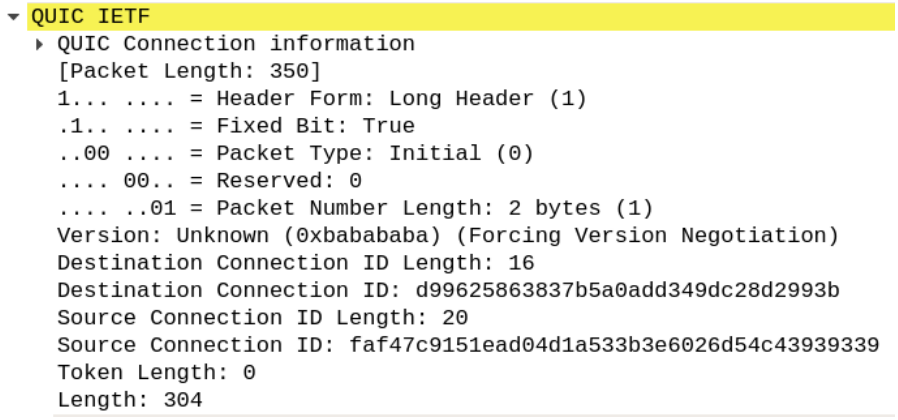

Abbildung 4.5: Der Aufbau ein Initial Pakets in Wireshark

Alle erzeugten Quiche-Anfragen wurden mit Tshark an mein WLAN Interface aufgezeichnet. Dadurch wurden alle QUIC Quiche-Anfragen an meinen Kwik Server auf Port 4433 aufgezeichnet. (siehe Abbildung 4.6)

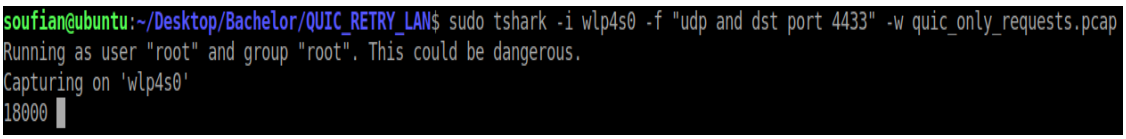

Abbildung 4.6: Paketaufzeichnung der Quiche-Anfragen mit Tshark

Time	Source	Destination	Protocol	Length	Info
0.000	192.168.188.31	192.168.188.40	QUIC	1242	Initial,
4.867	192.168.188.31	192.168.188.40	QUIC	1242	Initial,
9.150	192.168.188.31	192.168.188.40	QUIC	1242	Initial,
13.4…	192.168.188.31	192.168.188.40	QUIC	1242	Initial,
15.0…	192.168.188.31	192.168.188.40	QUIC	1242	Initial,

Abbildung 4.7: Ausschnitt QUIC-Anfragen in Wireshark

In der Abbildung 4.7 ist ein Ausschnitt der aufgezeichneten QUIC-Anfragen in Wireshark, die mit dem Bash Script erzeugt worden sind, zu sehen.

4.2.1.3 Retry-Kommunikation durch Kwik Server

Der Kwik Server antwortet auf ein Initial-Paket mit einem Retry-Paket, um eine Antwort vom Quiche-Client zu erzwingen. Die komplette Retry-Kommunikation wurde durch eine Tshark-Instanz einschließlich Retry-Anfragen und Retry-Antworten auf Port 4433 aufgezeichnet. (siehe Abbildung 4.8)

```
soufian@ubuntu:~/Desktop/Bachelor/QUIC_RETRY_LAN$ sudo tshark -i wlp4s0 -f "udp and port 4433" -w quic_with_retry.pcap
Running as user "root" and group "root". This could be dangerous.
Capturing on 'wlp4s0'
47978
```

Abbildung 4.8: Paketaufzeichnung der kompletten Retry-Kommunikation mit Tshark

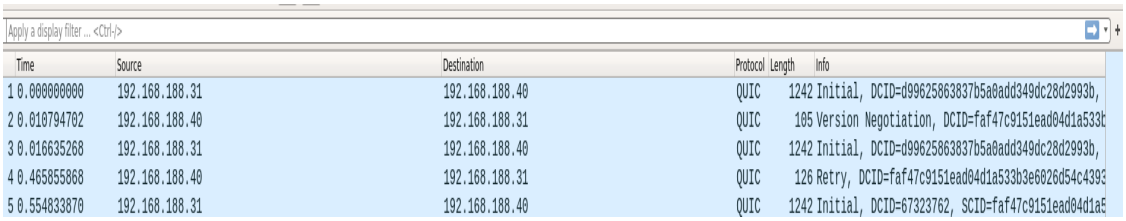

Abbildung 4.9: Ausschnitt einer QUIC Kommunikation durch Retry in Wireshark

In der Abbildung 4.9 ist ein Ausschnitt einer aufgezeichneten QUIC-Kommunikation mittels Retry in Wireshark.

1. Der QUIC Client mit der IP 192.168.188.31 sendet ein inital-Paket an den Kwik Server mit der IP 192.168.188.40.
2. Der Server antwortet mit einem Version Negotiation Paket. Dieses enthält eine Liste der Versionen, die der Server akzeptiert.
3. Anschließend sendet der Client ein Initial-Paket mit einer vom Server unterstützte QUIC version.
4. Der Server sendet ein Retry-Paket zur Adressvalidierung
5. Der Client antwortet mit einem Initial-Paket bestehend aus dem Server-Token.

Bei den erzeugten Initial-Paketen vom Quiche Client ist aufgefallen, dass der Kwik Server dazu gezwungen wurde, ein Version Negotiation mit dem Quiche Client durchzuführen.

Dadurch ist der Rückfragemechanismus bereits vollzogen, bevor der Retry-Prozess gestartet worden ist, da vom Client eine Antwort provoziert werden konnte. Aus diesem Grunde wird in Kapitel 4.2.3 überprüft, ob wir nicht bereits mit einem Version-Negotiation-Paket Antworten von Clients erzwingen können.

4.2.1.4 Validierung

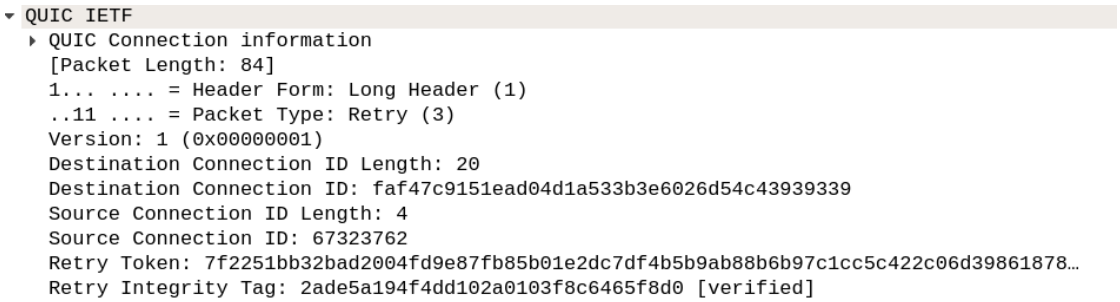

Abbildung 4.10: Retry-Paket mit generiertem Token in Wireshark

In der Abbildung 4.10 ist das Retry-Paket zu sehen, das vom Kwik Server erstellt wird. Das Retry-Paket enthält den Token, welches der Quiche Client bei der Rückantwort zurückgeben muss, damit der Kwik Server die Echtheit des Clients verifizieren kann.

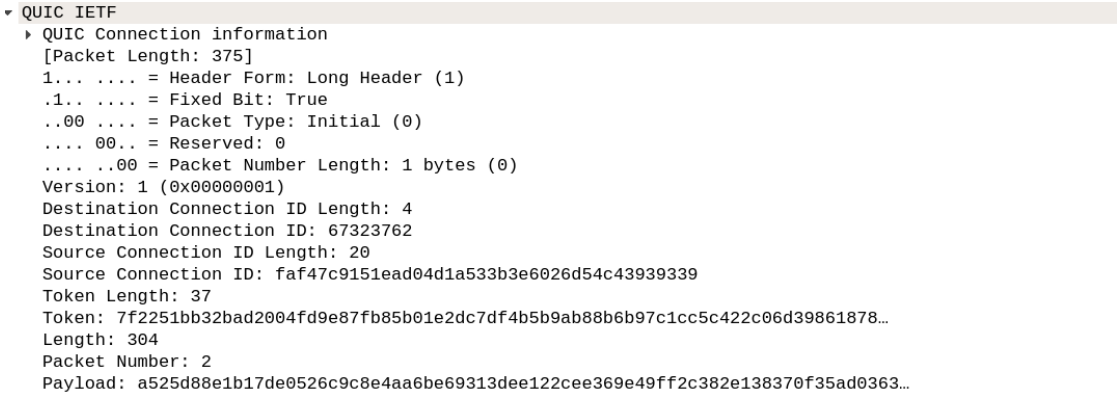

Abbildung 4.11: Initial-Paket vom Quiche Client als Antwort auf das Retry-Paket

In der Abbildung 4.11 ist die Retry Antwort aufgeführt, die vom Quiche Client erstellt wird. Sie enthält das Token, welches der Kwik Server in der Retry-Anfrage gesendet hat.

Der Quiche Client sendet die Retry-Antwort mit dem richtigen Token und der Kwik Server kann verifizieren, dass die Gegenseite echt ist.

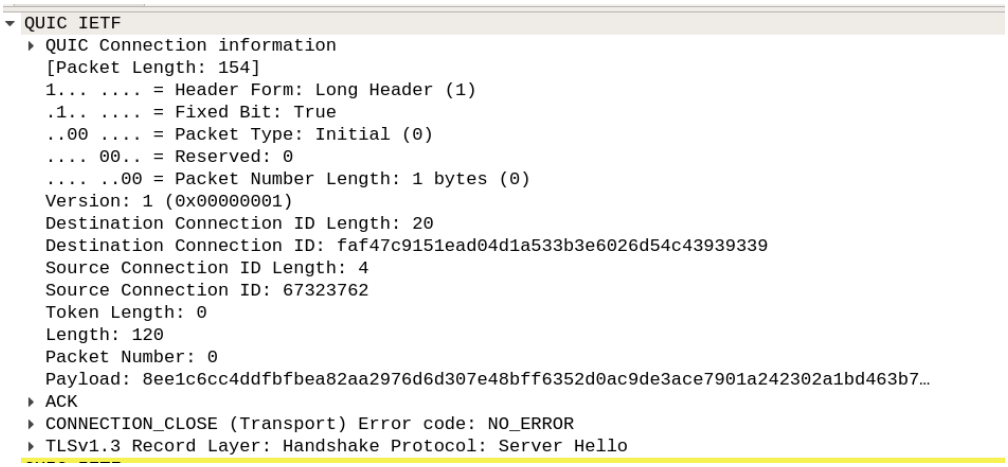

Abbildung 4.12: Connection close Frame um den Retry-Prozess abzuschließen

Weiterhin wurde jede abgeschlossene Validierung mit einem **Connection close** Frame beendet, da nur der Retry-Prozess für die Addressvalidierung entscheidend ist. Der Server sendet wie in Abbildung 4.12 zu sehen ist ein **Connection close** Frame, um seinen Peer zu benachrichtigen, dass die Verbindung geschlossen wurde.

4.2.1.5 IP-Spoofing von Initial-Paketen

Vier Werkzeugen, die in 4.1 vorgestellt wurden, habe ich für das IP-Spoofing benutzt:

- Kwik
- TCPreplay
- Tshark
- Quiche

Zu Beginn wurden die aufgezeichneten QUIC-Anfragen aus 4.6 mit Tshark gefiltert. Dies sind die ersten Initial-Pakete nach der Version Negotiation und vor ein Retry in QUIC. (siehe Abbildung 4.13).

```
soufian@ubuntu:~/Desktop/Bachelor/QUIC_RETRY_LAN$ sudo tshark -nr quic_only_requests.pcap -w quic_initial_requests.pcap
 -Y "udp.dstport==4433 and quic and quic.long.packet_type==0 and quic.packet_number==1 and quic.scil>0 and quic.dcil>0
and tls.handshake.type==1"
Running as user "root" and group "root". This could be dangerous.
```

Abbildung 4.13: Filterung aller Pakete außer die Initial-Pakete nach Version Negotiation

Als dritten Schritt zeigt die Abbildung 4.14 wie die Initial-Pakete mit TCPreplay von einem externen Acer Host im LAN gegen den Kwik Server gesendet worden sind. Durch das Einspielen der Pakete mit TCPreplay, besteht hinter den Paketen kein aktiver Client. Die Pakete werden zu einem Zeitpunkt verschickt, an dem kein Client existiert und sind somit gespoofte Pakete gespoofte Pakete.

```
soufian@soufian-Aspire-ES1-572:~/Schreibtisch$ sudo tcpreplay -t -i wlp2s0 quic_initial_requests.pcap
Actual: 5997 packets (7448274 bytes) sent in 0.527581 seconds
Rated: 14117782.8 Bps, 112.94 Mbps, 11366.97 pps
Flows: 5407 flows, 10248.66 fps, 5997 flow packets, 0 non-flow
Statistics for network device: wlp2s0
        Successful packets:        5997
        Failed packets:            0
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0
```

Abbildung 4.14: replay Initial-Pakete mit TCPreplay

Zu jeder gespooften Anfrage hat der Kwik Server eine Retry-Anfrage wie in Abbildung 4.10 gesendet. Wir konnten durch die Retry-Anfragen keine Rückantworten provozieren, da die Quellen gespooft waren.

4.2.2 QUIC Deployment im Internet

Der **Kwik Server** wurde erstmal auf einer öffentlichen IP-Adresse in einer VM deployt. Um Kwik laufen zu lassen, wurden auf der VM folgende Konfigurationen vorgenommen:

- JDK 11 installiert, um die Ausführung von Kwik auf der VM zu ermöglichen.
- Mittels Gradle wurde der Quellcode vom Kwik Server als ein JAR package file format gebaut. Die JAR File wurde daraufhin mittels Secure copy protocol auf die VM kopiert und anschließend ausgeführt.

Der erste Deployment lief knapp 8 Wochen lang, jedoch es gab zu wenig QUIC-Anfragen. Aus diesem Grund wurde ein zweites Deployment auf einen anderen Server mit einem /24 Präfix aufgesetzt, um mehr QUIC-Anfragen zu empfangen. Auf einem Interface empfangen wir UDP Traffic für das /24 Präfix, die an etwa 256 Adressen gerichtet sind. Der UDP Traffic, der an das Präfix adressiert ist, wird über Routing-Regeln an das Interface weitergeleitet, an dem wir horchen.

Hierfür musste der Kwik Server umgebaut werden, da der Datagramm Socket nur an eine Adresse und ein Port gebunden war. Folgende Änderungen wurden durchgeführt:

- Mit der Bibliothek Pcap4J wurde ein RAW Socket hinzugefügt, der den UDP Traffic von einem gegebenen Interface empfängt. Daraufhin wurde der empfangene UDP Traffic nach QUIC-Anfragen gefiltert.
- Beim Versenden der Retry-Pakete wurde der UDP Header inklusive IP-Header mittels Pcap4J geschrieben. Der QUIC Payload wurde als RAW bytes in den UDP Paket hinzugefügt. Das Ganze wurde anschließend über ein IP RAW Socket mittels RockSaw Socket Library versendet.

Auf den neuen Server wurden folgende Konfigurationen eingestellt, damit das modifizierte Kwik läuft:

- Hier wurde ebenfalls JDK 11 installiert.
- JAR File neu gebaut, um Pcap4J als Abhängigkeit zu inkludieren.
- RockSaw Socket Library wurde direkt auf dem Server installiert.

Das zweite Deployment lief 2 Wochen neben dem ersten Deployment, um Anfragen zu sammeln.

4.2.3 Testung von Standard QUIC Clients

In diesem Unterkapitel wird die Frage erörtert, ob QUIC Clients, die viel in Gebrauch sind, bereits mit einem Version Negotiation ein Handshake hinbekommen können? Dies sind vor allem Browser, die in Linux, Windows und macOS benutzt werden. Hierfür wurden Firefox und Chrome in Linux und Windows getestet. Safari wurde hingegen in macOS getestet.

Damit einzelne Browser mit uns über QUIC kommunizieren können, müssen wir dem Browser bekanntgeben, dass wir eine QUIC-Kommunikation unterstützen können. Wir können die Unterstützung von QUIC durch die Verwendung von Alt-Svc response Header in HTTP bekanntgeben [32]. Wenn ein Browser eine Anfrage über HTTP an eine Quelle stellt, mit der es noch nie zuvor gesprochen hat, weiß es nicht, ob die Quelle QUIC unterstützt, sodass es die erste Anfrage über TCP sendet. Die Antwort über den HTTP-header Alt-Svc, signalisiert dem Browser dass wir QUIC unterstützen.

Aus diesem Grund wurde ein Python Script geschrieben, dass stets das Alt-Svc response Header auf TCP Anfragen zurück schickt. Das Script ist in Abbildung 4.15 dargestellt.

```
import http.server
import socketserver
from http.server import HTTPServer, BaseHTTPRequestHandler
import http.server, ssl

server_address = (IP, 443)

class MyHandler(http.server.SimpleHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("alt-svc", 'h3=":443"; ma=2592000')
        self.end_headers()
        print(self.wfile)
        self.wfile.write(bytes("<html><head><title>Title goes here.</title></head>/html>","utf-8"))
        self.wfile.write(bytes("<body><p>This is a test.</p>", "utf-8"))
        self.wfile.write(bytes("</body></html>", "utf-8"))

httpd = http.server.HTTPServer(server_address, MyHandler)

httpd.socket = ssl.wrap_socket(httpd.socket,
                               server_side=True,
                               certfile='localhost.pem',
                               ssl_version=ssl.PROTOCOL_TLS)
httpd.serve_forever()
```

Abbildung 4.15: Python Script, das stets das Alt-Svc Header zurück schickt

Alle Browser-Anfragen wurden gegen den Kwik Server mit der öffentlichen IP getestet. Das Deployment für die öffentliche IP wurde bereits in Kapitel 4.2.2 geschildert. Der Python Server wurde ebenfalls auf dem VM gestartet um Browser-HTTP-Anfragen abzufangen und mit dem alt-svc Response Header zu beantworten.

Erstmal wurde die Firefox Version 94 sowohl in Windows und Linux getestet. Das QUIC Protokoll wird seit der Firefox-Version 88 standardmäßig für alle Nutzer unterstützt.

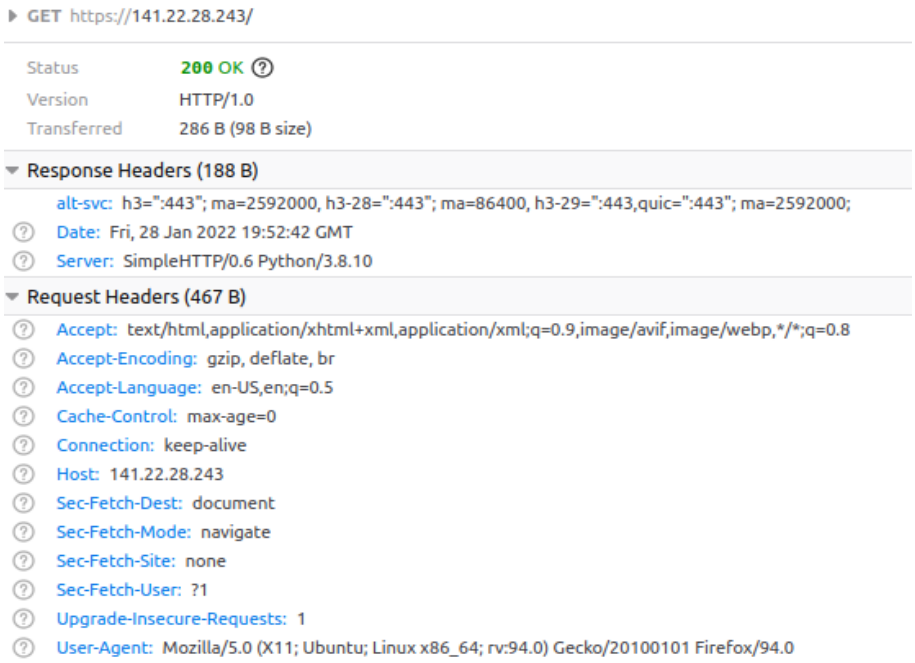

Abbildung 4.16: Firefox HTTP Request in Linux

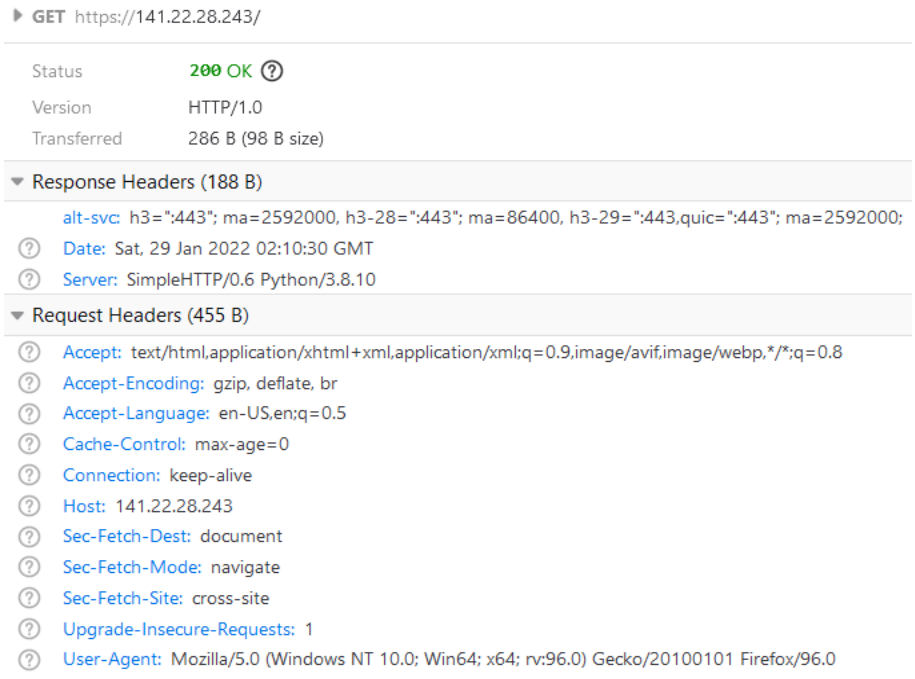

Abbildung 4.17: Firefox HTTP Request in Windows

In den Abbildungen 4.16 und 4.17 sind zwei Firefox HTTP requests sowohl für WIndows als auch für Linux zu sehen. Unser kleiner Python HTTP Server antwortete auf beide Requests mit dem alt-svc Response Header mit dem Hinweis, dass wir QUIC unterstützen. Damit wurde Firefox signalisiert, eine QUIC-Anfrage auf Port 443 zu starten.

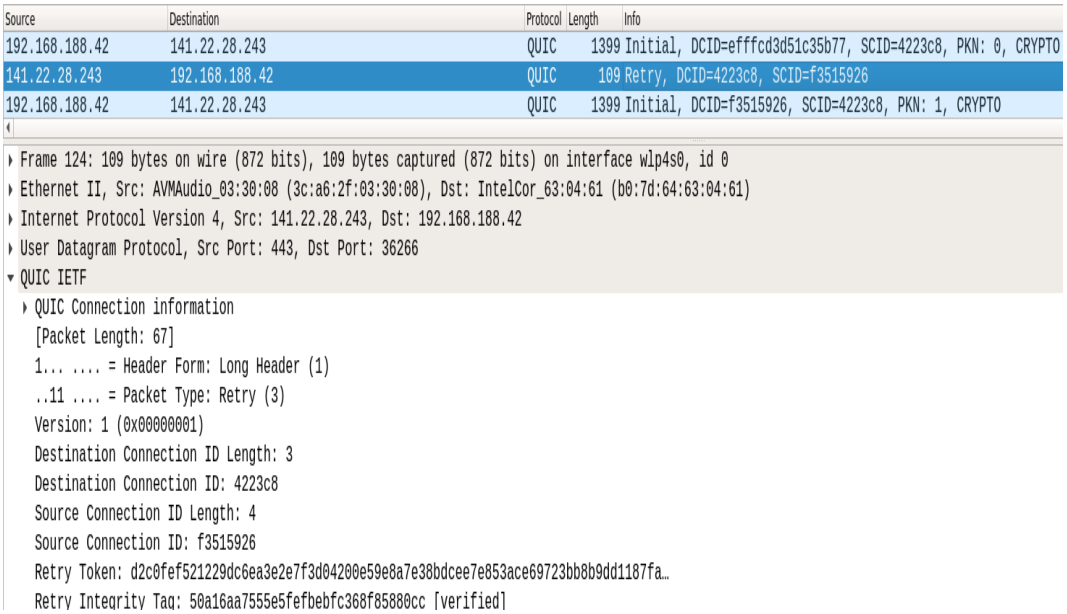

Abbildung 4.18: Retry-Kommunikation durch Firefox sowohl in Windows als auch Linux

In der obigen Abbildung ist erkennbar, dass Firefox kein Version Negotiation mit unserem Kwik Server gestartet hat, stattdessen wurde eine Retry-Kommunikation aufgebaut, um den Firefox Client zu validieren.

Als Nächstes wurde Google Chrome in der Version 97 auf Linux und Windows getestet. In den Abbildungen 4.19 und 4.20 sind zwei Chrome HTTP requests sowohl für Windows als auch für Linux aufgeführt. Unser kleiner Python HTTP Server antwortete auf beide Requests ebenfalls dem alt-svc Response Header mit dem Hinweis, dass wir QUIC unterstützen. Damit wurde Chrome signalisiert, eine QUIC-Anfrage auf Port 443 zu starten.

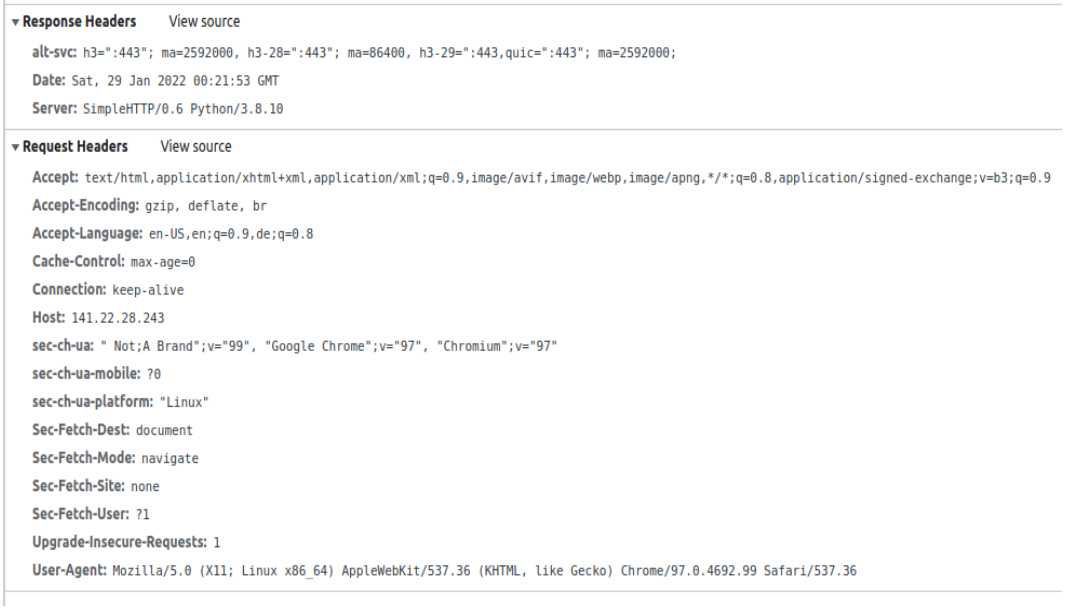

Abbildung 4.19: Chrome HTTP Request in Linux

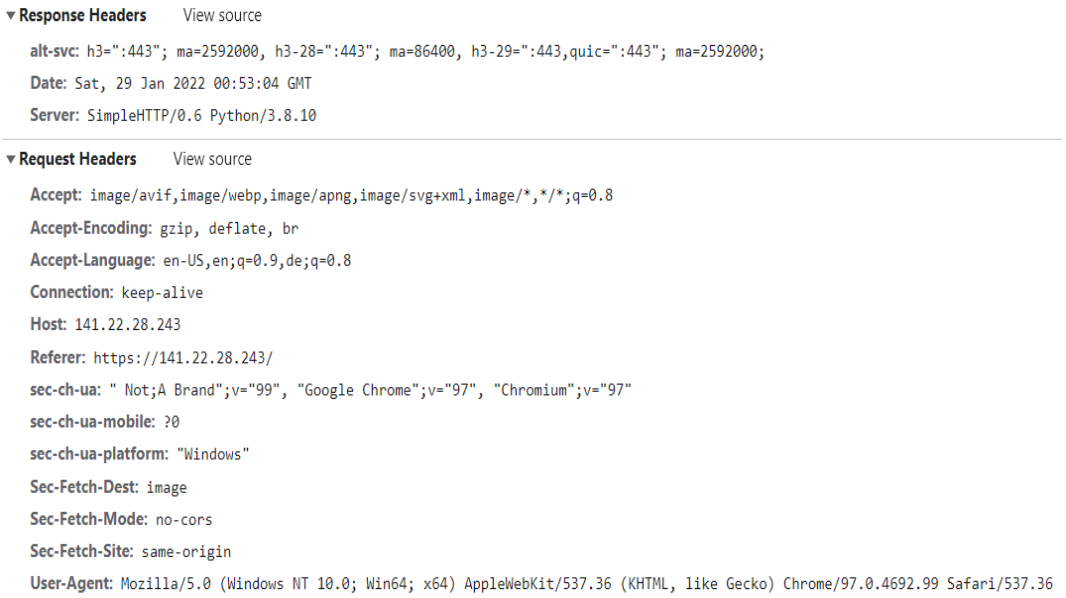

Abbildung 4.20: Chrome HTTP Request in Windows

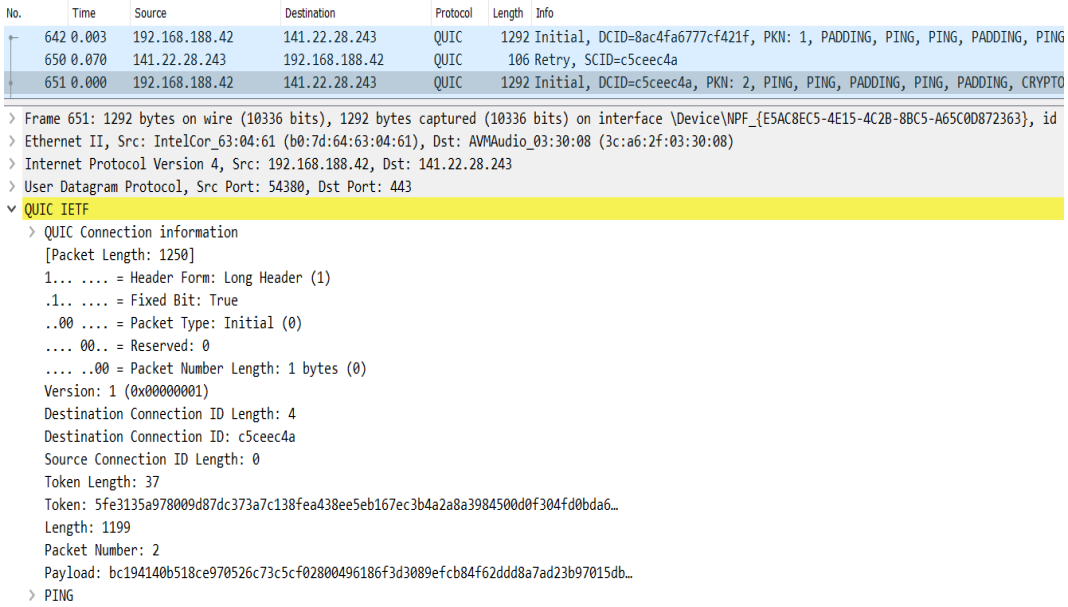

Abbildung 4.21: Retry-Kommunikation durch Chrome sowohl für Windows als auch Linux

Der Abbildung 4.21 ist zu entnehmen, dass Chrome kein Version Negotiation mit unserem Kwik Server gestartet hat, stattdessen wurde eine Retry-Kommunikation aufgebaut, um den Chrome Client zu validieren.

Daraufhin wurde Safari in der Version 15 auf MacOS getestet. In der Abbildung 4.22 sieht man das Safari HTTP request in MacOS. Unser kleiner Python HTTP Server antwortete auf den Request mti dem alt-svc Respnse Header mit dem Hinweis, dass wir QUIC untersützen. Damit wurde Safari signalisiert ein QUIC Anfrage auf Port 443 zu starten.

In der Abbildung 4.23 ist zu sehen, dass Safari kein Version Negotiation mit unserem Kwik Server gestartet hat, vielmehr wurde eine Retry-Kommunikation aufgebaut, um den Safari Client zu validieren.

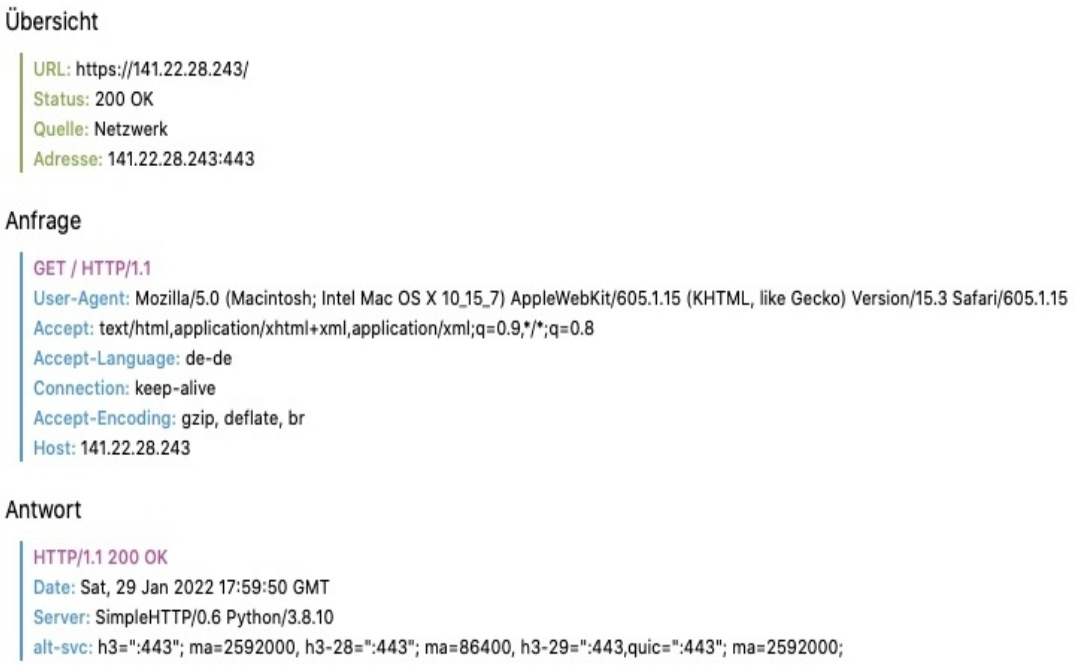

Abbildung 4.22: Safari HTTP Request in MacOS

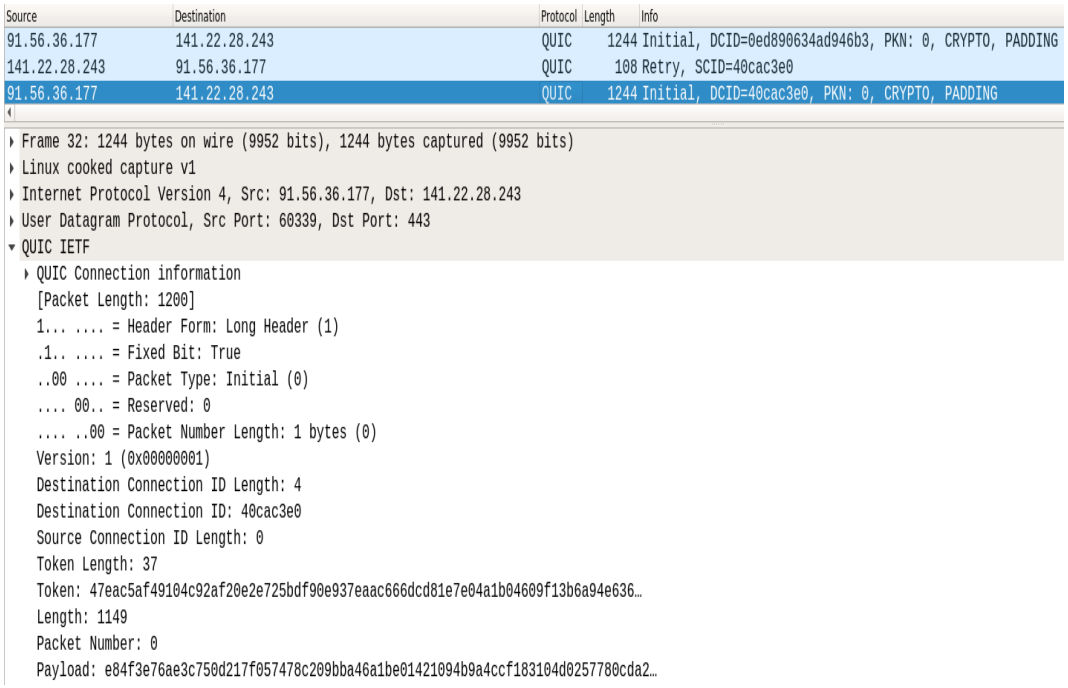

Abbildung 4.23: Safari Retry Kommunikation in MacOS

4.2.3.1 Zusammenfassung

Tabelle 4.1: Browserverhalten gegenüber Retry und Version Negotiation

Browser	**Retry-Kommunikation**	**Version Negotiation**
Firefox in Linux/Windows	Ja	Nein
Chrome in Linux/Windows	Ja	Nein
Safari in MacOS	Ja	Nein

Die Tabelle 4.1 fasst nochmal die Ergebnisse zusammen. Alle getesteten Browser haben ausschließlich die Retry-Kommunikation mitgemacht. Eine Version Negotiation hat hingegen keiner der getesteten Browser vollzogen.

4.3 Umsetzung in DNS

4.3.1 Umsetzung Truncation-Flag-Ansatz

4.3.1.1 DNS Server

Wir haben die **Dnsjava** Bibliothek verwendet, um einen angepassten leichtgewichtigen DNS-Nameserver für unsere Bedürfnisse zu schreiben und den TC-Flag-Ansatz möglichst einfach umzusetzen.

Damit wir DNS-Anfragen über UDP und gleichzeitig durch Truncated Flag provozierte Rückantworten über TCP akzeptieren können, wurde hierfür ein Multiplexer verwendet. Der Multiplexer ist ein Java NIO Selector, der eine oder mehrere Java NIO Channels verarbeiten kann und feststellt, welche Channels zum Lesen oder Schreiben bereit sind [3]. Pro Transportprotokoll wurde ein Kanal erstellt. In der Abbildung 4.24 wurde für TCP und UDP jeweils ein Kanal erstellt.

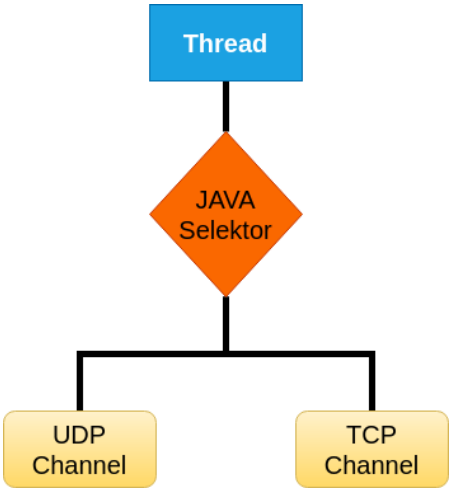

Abbildung 4.24: Java Selektor für TCP und UDP

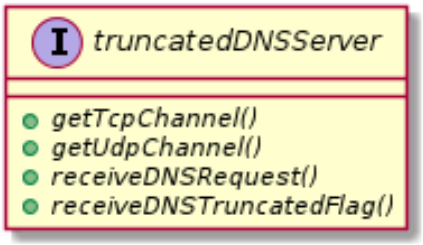

Abbildung 4.25: Interface vom DNS Server für TC-FLag-Ansatz

- **getTcpChannel** gibt den Tcp Channel zurück.
- **geUdpChannel** gibt den Udp Channel zurück.
- **receiveDNSRequest** nimmt ein DNS request über UDP entgegen und versendet daraufhin eine Response mit TC Flag.
- **receiveDNSTruncatedFlag** nimmt die Rückantwort des Clients über TCP an.

```
Message response = new Message();
Header header = new Header();
header.setID(request.getHeader().getID());
header.setFlag(Flags.TC);
header.setFlag(Flags.QR);
header.setFlag(Flags.RA);
header.setFlag(Flags.AA);
response.setHeader(header);
Record question = request.getQuestion();
response.addRecord(question, Section.QUESTION);
for (int j = 0; j < 1; ++j) {
    response.addRecord(Record.fromString(Name.root, 1, 1, 86400L,
    "127.0.0.1", Name.root), Section.ANSWER);
}
```

Abbildung 4.26: Codeschnipsel zur Erstellung der DNS-Response mit TC Flag

Das Codeschnipsel in Abbildung 4.26 erstellt eine Response genauso wie sie in Abbildung 3.4 definiert ist. Der TC Flag wird mit weiteren Flags im Header gesetzt und ein Teil der Antwort besteht aus einem Resource Record mit einer Bogon Adresse 127.0.0.1.

4.3.1.2 Testaufbau für TC im lokalen Netzwerk

Um das Konzept für DNS durchzuführen wurde das Experiment in einem lokalen Netzwerk aufgesetzt. Der Aufbau der Laborumgebung ist in der Abbildung 4.27 dargestellt.

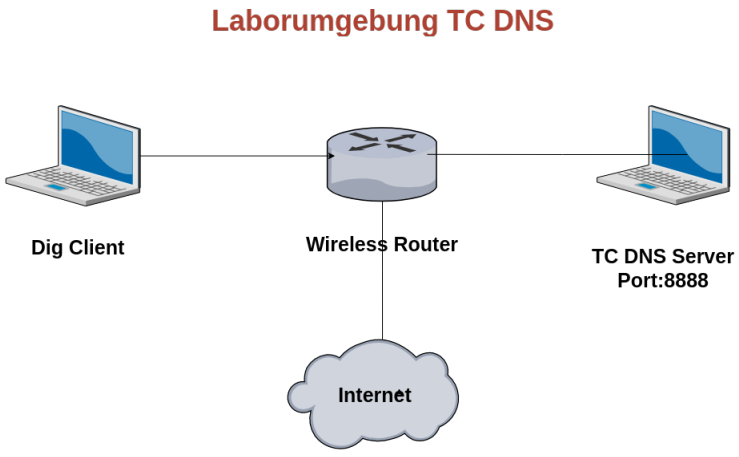

Abbildung 4.27: LAN-Netzwerk für TC-DNS-Experiment

Der TC DNS Server wurde auf einem Laptop mit Lubuntu, eine leichtgewichtige Linux-Distribution, aufgesetzt und die DNS-Anfragen wurden mit dem Dig Client erzeugt. Der DNS Server für den TC-Ansatz wurde durch folgende Schritte aufgesetzt:

- Der DNS Server wurde in Java geschrieben. Hierfür wurde JDK 11 installiert, um die Ausführung des DNS Servers möglich zu machen.
- Mittels Maven wurde die Bibliothek Dnsjava als Abhängigkeit hinzugefügt, um ein Teil des Quellcodes vom DNS Server zu kompilieren.

4.3.1.3 Generierung von DNS-Anfragen für TC

6000 valide Anfragen wurden mit dem Dig Client in ein Bash Script generiert. Diese 6000 Anfragen werden in Kapitel 5 genutzt, um die Antwortzeiten zu messen. In der Abbildung 4.28 ist der Codeschnipsel zur Erzeugung der Anfragen dargestellt.

Abbildung 4.28: Bash Script zur Erzeugung von DNS-Anfragen mittels Dig

```
for i in $(seq 1 6000); do
    dig +qr +edns @192.168.188.40 -p 8888 localhost
    sleep 1
done
```

```
▾ Domain Name System (query)
    Transaction ID: 0x7b94
  ▾ Flags: 0x0120 Standard query
      0... .... .... .... = Response: Message is a query
      .000 0... .... .... = Opcode: Standard query (0)
      .... ..0. .... .... = Truncated: Message is not truncated
      .... ...1 .... .... = Recursion desired: Do query recursively
      .... .... .0.. .... = Z: reserved (0)
      .... .... ..1. .... = AD bit: Set
      .... .... ...0 .... = Non-authenticated data: Unacceptable
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 1
  ▾ Queries
    ▸ localhost: type A, class IN
```

Abbildung 4.29: Der Aufbau einer DNS-Anfrage in Wireshark

Der Aufbau einer DNS-Anfrage ist in Abbildung 4.29 zu sehen. Jede DNS-Anfrage wies das DNS Nachrichtenformat auf, welches in Kapitel 2.3.2.1 vorgestellt worden ist. Jede Anfrage wurde vom Dig Client erzeugt und an den DNS-Server auf Port 8888 versendet.

Der gesamte Netzwerkverkehr der generierten DNS-Anfragen wurde an Port 8888 mit TShark an mein WLAN-Interface **wlp4s0** aufgezeichnet (siehe Abbildung 4.30). Ein Ausschnitt der validen Anfragen ist ebenfalls durch Wireshark in Abbildung 4.31 zu sehen. Anfragen wurde von der IP-Quelle 192.168.188.31 zum DNS-Server mit der Ziel-IP 192.168.188.42 gesendet. Auf jeder dieser Anfragen antwortet der Server mit DNS truncated Paketen.

```
soufian@ubuntu:~/Desktop/Bachelor/DNS/TC Flag Ansatz/TC_Ansatz_im_LAN$ sudo tshark -i wlp4s0 -f "port 8888" -w dns_TC_valid_requests.pcap
Running as user "root" and group "root". This could be dangerous.
Capturing on 'wlp4s0'
49396 ^C
```

Abbildung 4.30: Paketaufzeichnung valider DNS-Anfragen mit Tshark

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.188.31	192.168.188.42	DNS	92	Standard query 0x7b94 A loca
2	0.310110002	192.168.188.42	192.168.188.31	DNS	84	Standard query response 0x7b
3	0.407866005	192.168.188.31	192.168.188.42	TCP	74	34591 → 8888 [SYN] Seq=0 Win
5	0.410113085	192.168.188.31	192.168.188.42	TCP	66	34591 → 8888 [ACK] Seq=1 Ack

Abbildung 4.31: Ausschnitt valider DNS-Anfragen in Wireshark

4.3.1.4 Validierung durch TC

Jede DNS-Anfrage wurde mit einer DNS-Response beantwortet, bei welcher der Truncation Flag auf True gesetzt worden ist (siehe dazu Abbildung 4.33).

In Abbildung 4.32 ist eine beispielhafte TC-Kommunikation aufgeführt, wo der Client nach eine TC-Anfrage über TCP zurückkommt.

Source	Destination	Protocol	Info
192.168.188.31	192.168.188.42	DNS	Standard query 0x7b94 A localhost OPT
192.168.188.42	192.168.188.31	DNS	Standard query response 0x7b94 A localhost A 127.0.0.1
192.168.188.31	192.168.188.42	TCP	34591 → 8888 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=3021634149 TSecr=0 WS=128
192.168.188.42	192.168.188.31	TCP	8888 → 34591 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=3677092216 TSe
192.168.188.31	192.168.188.42	TCP	34591 → 8888 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=3021634152 TSecr=3677092216
192.168.188.31	192.168.188.42	TCP	34591 → 8888 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=52 TSval=3021634152 TSecr=3677092216
192.168.188.42	192.168.188.31	TCP	8888 → 34591 [ACK] Seq=1 Ack=53 Win=65152 Len=0 TSval=3677092219 TSecr=3021634152
192.168.188.42	192.168.188.31	TCP	8888 → 34591 [RST, ACK] Seq=1 Ack=53 Win=65152 Len=0 TSval=3677092221 TSecr=3021634152

Abbildung 4.32: Ausschnitt einer DNS-Kommunikation mittels TC in Wireshark

1. Der Dig Client mit der IP 192.168.188.31 sendet eine DNS-Anfrage an den DNS-Server mit der IP 192.168.188.42.
2. Der DNS-Server antwortet mit einer DNS-Response. TC Flag ist ebenfalls gesetzt.
3. Anschließend kommt der Dig Client über TCP zurück und versucht ein Handshake mit unserem DNS-Server aufzubauen.
4. Der DNS-Server sendet ein TCP RST-Flag, damit die Verbindung sofort beendet werden soll.

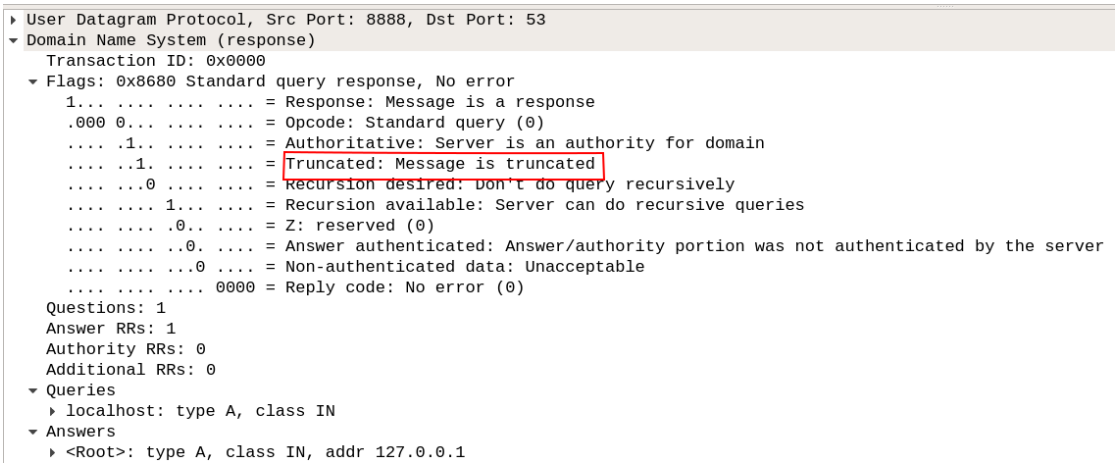

Abbildung 4.33: DNS-Response mit truncated Flag in Wireshark

4.3.1.5 IP-Spoofing von DNS-Paketen

Weiterhin wurden 6000 DNS-Anfragen mit zufälligen IP-Quelladressen an den DNS-Server mittels Scapy gespooft und versendet. Der dazugehörige Codeschnipsel ist in Abbildung 4.34 befindlich. Die Aufzeichnung und ein Ausschnitt der gespooften Anfragen an den DNS-Server mit der Ziel-IP 192.168.188.40 sind in den Abbildungen 4.35 und 4.36 zu entnehmen.

Abbildung 4.34: Python Script zur Erzeugung von IP gespooften DNS-Anfragen

```
from scapy.all import *
count = 0;
while count!=6000:
    answer = send(IP(src=RandIP(),dst="192.168.188.40")/
    UDP(dport=8888,chksum=0)/DNS(rd=1,qd=DNSQR(qname="localhost")))
count=count+1;
```

```
soufian@ubuntu:~/Desktop/Bachelor/TC_Ansatz_im_LAN$ sudo tshark -i wlp4s0 -f "udp and port 8888" -w dns_TC_spoofed_requests.pcap
Running as user "root" and group "root". This could be dangerous.
Capturing on 'wlp4s0'
13996 ^C
```

Abbildung 4.35: Paketaufzeichnung IP spoofed DNS-Anfragen mit Tshark

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	133.243.80.41	192.168.188.40	DNS	69	Standard query 0x0000 A local
2	0.000399703	192.168.188.40	133.243.80.41	DNS	84	Standard query response 0x000
3	0.036008961	91.21.186.131	192.168.188.40	DNS	69	Standard query 0x0000 A local
4	0.036386734	192.168.188.40	91.21.186.131	DNS	84	Standard query response 0x000
5	0.079973919	125.197.160.164	192.168.188.40	DNS	69	Standard query 0x0000 A local
6	0.080390175	192.168.188.40	125.197.160.164	DNS	84	Standard query response 0x000

Abbildung 4.36: Ausschnitt gespoofter DNS-Anfragen in Wireshark

Zu jeder gespooften Anfrage hat der DNS-Server eine TC-Anfrage wie in Abbildung 4.33 gesendet.Anhand der TC-Anfragen konnten keine Rückantworten provoziert werden, da die Quellen gespooft waren.

4.3.1.6 TC Deployment im Internet

Der implementierte DNS-Server wurde ebenfalls auf eine VM getestet, um live Daten zu sammeln und zu erkunden, wie erfolgversprechend der TC-Ansatz im Internet wirklich ist. Die VM hängt an einer einzelnen öffentlichen IP und wird dementsprechend im Internet gescannt. Auf der VM wurden folgende Konfigurationen vorgenommen, um den DNS-Server laufen zu lassen:

- JDK 11 installiert, um die Ausführung vom DNS auf der VM zu ermöglichen.
- Mittels Maven wurden der Quellcode vom DNS-Server und die Bibliotheken als ein JAR package file format gebaut. Der JAR File wurde dann auf dem VM ausgeführt.

Der DNS-Server hat zwei Wochen lang auf den UDP Port 53 gehorcht, um Anfragen entgegenzunehmen. Zusätzlich wurde TCPDump benutzt, um den DNS-Verkehr auf dem Port aufzuzeichnen. (siehe hierzu die Abbildungen 4.37 und 4.38). Die Daten werden in Kapitel 5 ausgewertet.

```
benafia@caliban:~$ sudo tcpdump -vvv -i enp7s0 -w allesjetzt2.pcap host 141.22.213.55 and port 53
[sudo] password for benafia:
tcpdump: listening on enp7s0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C2370 packets captured
2370 packets received by filter
0 packets dropped by kernel
```

Abbildung 4.37: Paketaufzeichnung der DNS-Anfragen im Internet mit TShark

```
benafia@caliban:~$ sudo java -jar DNS-Server-1.0-SNAPSHOT-jar-with-dependencies.jar
[sudo] password for benafia:
----------------------------
Usage: java DNS_Server <port>
Default port: 53
----------------------------

DNS Server is running at port 53...
```

Abbildung 4.38: Der implementierte DNS-Server an einer öffentlichen IP auf Port 53

4.3.1.7 Testung von Standard Clients gegenüber TC-Ansatz

Darüber hinaus wurde im Experiment folgende Frage erörtert: Wie verhalten sich die DNS Clients, die viel in Gebrauch sind, gegenüber dem TC-Flag-Ansatz ? Dies sind vor allem die Clients in den Betriebssystemen wie Linux, Windows und macOS. Ebenso ist es wichtig zu wissen, wie gängige Browser wie Firefox oder Chrome auf den Ansatz reagieren. Als Erstes wurden die gängigen Browser Chrome und Firefox in Windows und Ubuntu getestet. Hierfür wurden die Anfragen auf unseren DNS Server umgeleitet. (siehe Abbildung 4.39 und 4.40)

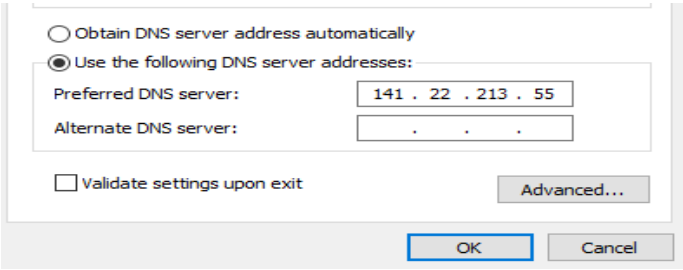

Abbildung 4.39: DNS-Server Umleitung in Windows

```
# This file is managed by man:systemd-resolved(8). Do not edit.
#
# This is a dynamic resolv.conf file for connecting local clients to the
# internal DNS stub resolver of systemd-resolved. This file lists all
# configured search domains.
#
# Run "systemd-resolve --status" to see details about the uplink DNS servers
# currently in use.
#
# Third party programs must not access this file directly, but only through the
# symlink at /etc/resolv.conf. To manage man:resolv.conf(5) in a different way,
# replace this symlink by a static file or a different symlink.
#
# See man:systemd-resolved.service(8) for details about the supported modes of
# operation for /etc/resolv.conf.

#nameserver 127.0.0.53
nameserver 141.22.213.55
options edns0
search fritz.box
```

Abbildung 4.40: DNS-Server Umleitung in Ubuntu

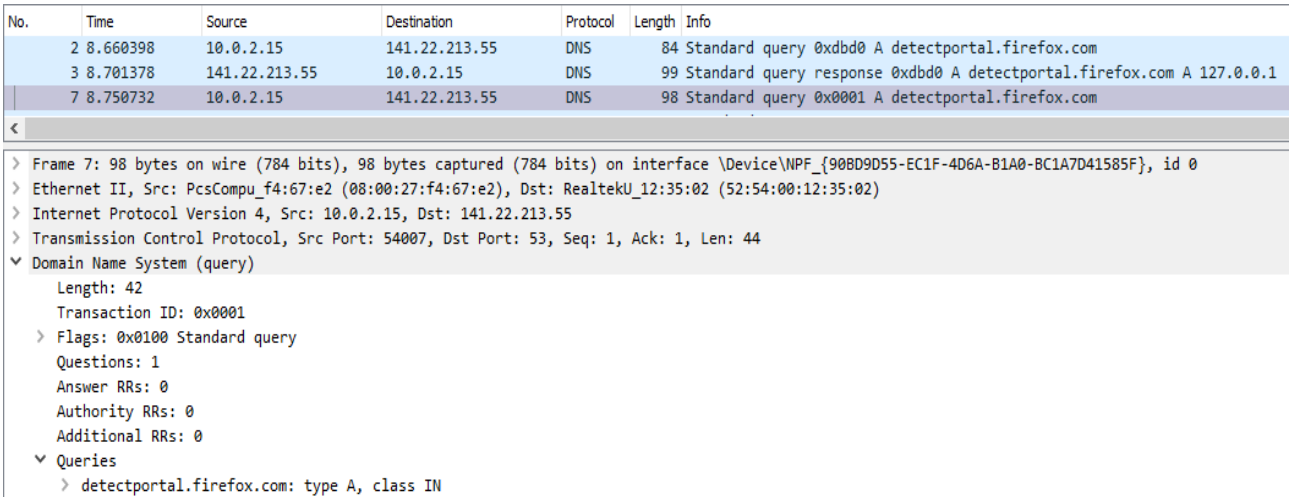

Abbildung 4.41: Firefox Antwort auf TC Flag

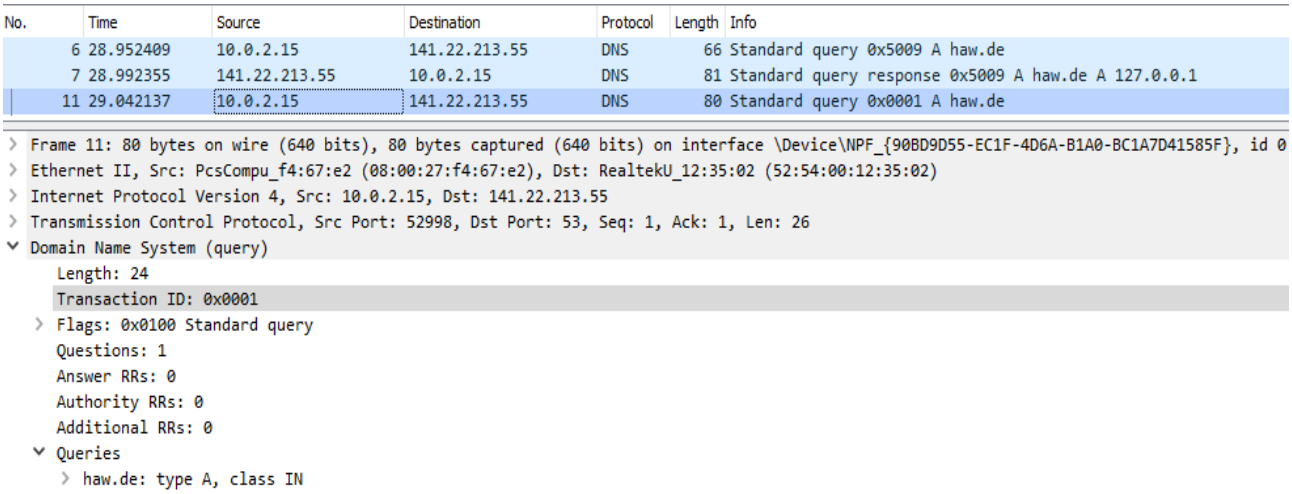

Abbildung 4.42: Chrome Antwort auf TC Flag

In Abbildung 4.41 ist zu sehen wie der DNS Client von Firefox über TCP zurück antwortet. Genauso wie Firefox antwortet Chrome auf den Truncated Flag über TCP zurück. (siehe Abbildung 4.42).

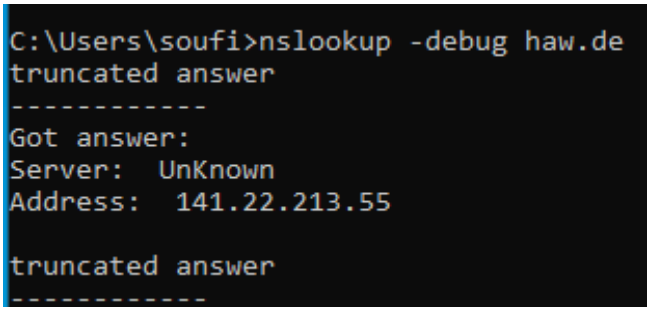

Abbildung 4.43: nslookup Antwort auf TC Flag in Windows

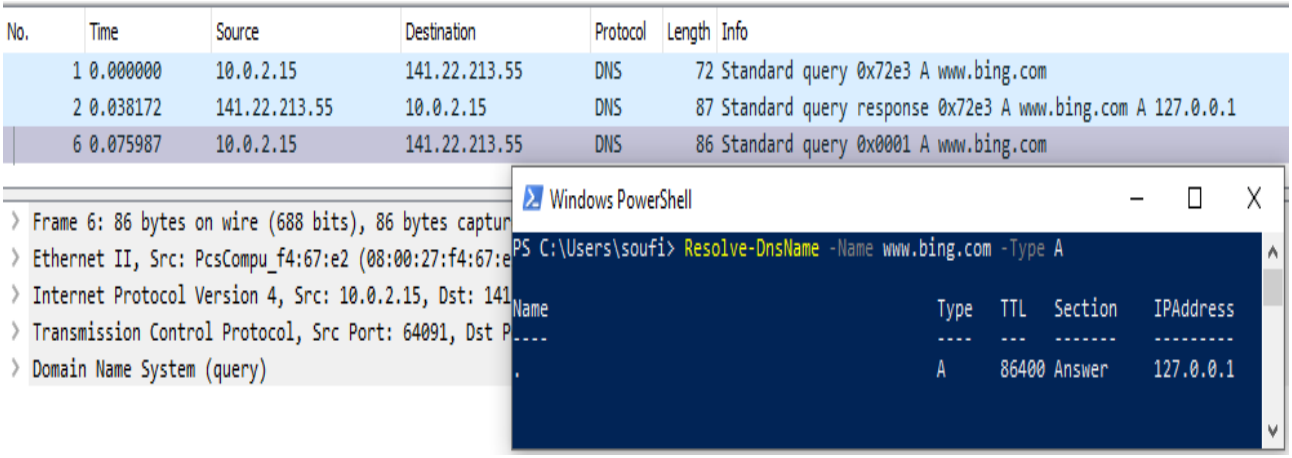

Abbildung 4.44: PowerShell Resolver Antwort auf TC Flag in Windows

Populäre DNS Clients wie nslookup und der PowerShell DNS Resolver antworten auf den Truncated Flag ebenfalls über TCP zurück.

In macOS wurde Safari getestet und reagiert entsprechend auf den TC Flag mit einer Antwort über TCP (siehe 4.45). Hierfür wurden ebenfalls die Anfragen auf unseren DNS-Server umgeleitet. Die Konfiguration ist in Abbildung 4.46 zu sehen.

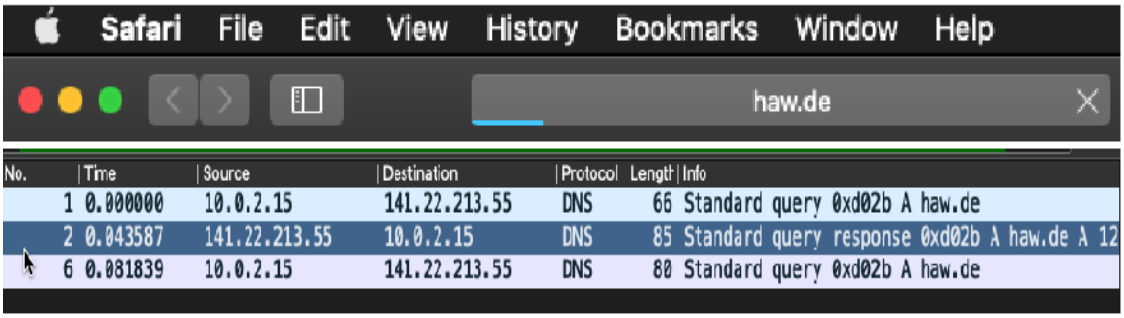

Abbildung 4.45: Safari Antwort auf TC Flag in Windows

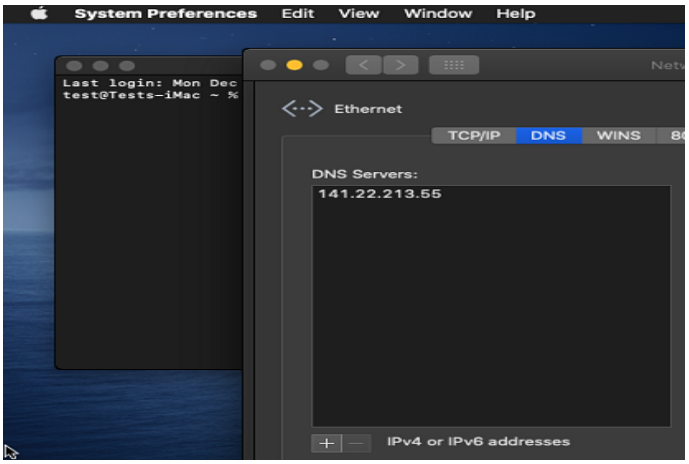

Abbildung 4.46: DNS-Server Umleitung in MAC

Tabelle 4.2: Browser-Verhalten gegenüber TC-Ansatz

Browser	**TC Kommunikation**
Firefox in Linux/Windows	Ja
Chrome in Linux/Windows	Ja
Safari in MacOS	Ja
nslookup	Ja
Resolve-DnsNam	Ja

Die Tabelle 4.2 fasst nochmal die Ergebnisse zusammen. Alle getesteten Clients haben die TC-Kommunikation mitgemacht.

4.3.2 Umsetzung DNS-Cookies-Ansatz

In diesem Unterkapitel wird der DNS-Cookie-Mechanismus im lokalen Netzwerk getestet und validiert. Anschließend wird der Bind9 Server im Internet deployt, um zu ermitteln, wie die DNS Clients sich gegenüber dem DNS-Cookies-Ansatz verhalten. Die gesammelten Daten werden in Kapitel 5 ausgewertet.

4.3.2.1 Testaufbau für DNS Cookies im lokalen Netzwerk

Die Laborumgebung für den DNS-Cookie-Ansatz unterscheidet sich nicht viel von jenem des TC Flag-Ansatzes. Der Aufbau der Laborumgebung entspricht der Abbildung 4.27 mit dem Unterschied, dass der BIND9 Server mit dem TC DNS-Server ersetzt werden muss. Der BIND9 Server wurde auf einen Laptop mit Lubuntu eine leichtgewichtige Linux-Distribution aufgesetzt und die DNS-Anfragen wurden mit dem Dig Client erzeugt.

BIND9 unterstützt DNS Cookies sowohl auf der Client-Seite als auch auf der Server-Seite. Hierfür wurde die Option explizit in der Konfiguration aktiviert.

```
options {
        directory "/var/cache/bind";
        listen-on port 53{141.22.213.55;};
        allow-query{any;};
        recursion no;
        require-server-cookie yes;
        // If there is a firewall between you and nameservers you want
        // to talk to, you may need to fix the firewall to allow multiple
        // ports to talk.  See http://www.kb.cert.org/vuls/id/800113

        // If your ISP provided one or more IP addresses for stable
        // nameservers, you probably want to use them as forwarders.
        // Uncomment the following block, and insert the addresses replacing
        // the all-0's placeholder.

        //========================================================================
        // If BIND logs error messages about the root key being expired,
        // you will need to update your keys.  See https://www.isc.org/bind-keys
        //========================================================================
        //dnssec-validation auto;
        //listen-on-v6 { any; };
};
```

Abbildung 4.47: Bind9-Server-Konfiguration

Die Option **require-server-cookie** zwingt den Server das RCode BADCOOKIE einschließlich eines korrekten Server-Cookies zurückzugeben, wenn ein Client eine UDP-Anfrage mit einem leeren oder ein inkorrekten Server-Cookie sendet. Es muss jedoch

ein Client-Cookie vorhanden sein, damit der Server in diesem Fall ein BADCOOKIE zurückgeben kann.

4.3.2.2 Generierung von DNS-Anfragen für DNS Cookies

6000 valide Anfragen wurden mit Dig mit der Cookie Option generiert.Diese werden in Kapitel 5 genutzt, um die Antwortzeiten zu messen.In der Abbildung 4.48 der Codeschnipsel zur Erzeugung der Anfragen dargestellt.

Abbildung 4.48: Bash Script zur Erzeugung von DNS-Anfragen mittels Dig

```
for i in $(seq 1 6000); do
    dig +qr +edns +cookie @192.168.188.40 -p 8888 localhost
    sleep 1
done
```

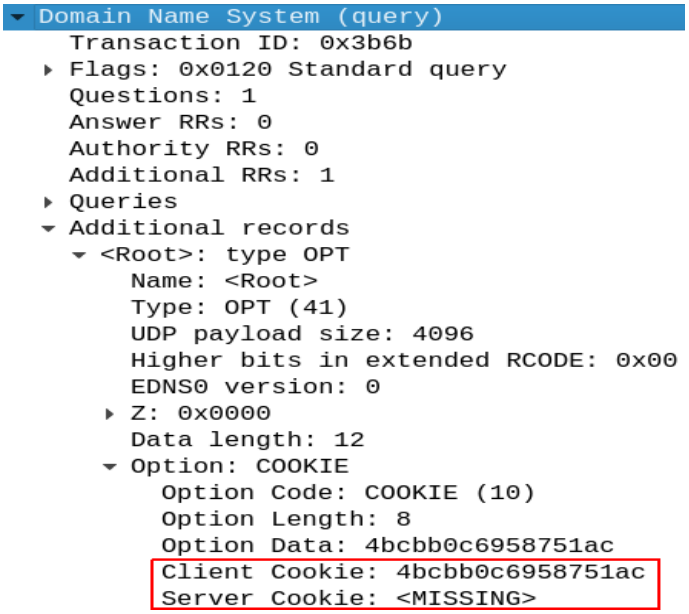

Abbildung 4.49: Der Aufbau einer DNS-Cookie-Anfrage in Wireshark

Der Aufbau einer DNS-Cookie-Anfrage ist in Abbildung 4.49 aufgeführt. Jede DNS-Cookie-Anfrage wies den DNS-Additional-Abschnitt auf, welcher in Kapitel 2.3.2.1 vorgestellt worden ist. Jede Anfrage wurde vom Dig Client erzeugt und an den Bind9 Server auf Port 8888 versendet. Im Additional-Abschnitt ist zu sehen, dass der Cookie als eine

OPT-Option eingetragen ist. Hierbei ist es wichtig, ein Client Cookie in der Anfrage hinzuzufügen, da der Mechanismus anderenfalls, wie bereits in 3.3.2 erwähnt, nicht funktionieren wird.

4.3.2.3 Validierung durch DNS Cookies

Jede DNS-Cookie-Anfrage wurde mit einer DNS Response beantwortet, wo der BADCOOKIE code hinzugefügt worden ist. In der Abbildung 4.50 ist ein Ausschnitt einer aufgezeichneten DNS-Kommunikation mittels DNS Cookies in Wireshark aufgeführt.

Source	Destination	Protocol	Info
192.168.188.31	192.168.188.42	DNS	Standard query 0x3b6b A localhost OPT
192.168.188.42	192.168.188.31	DNS	Standard query response 0x3b6b RRset exists A localhost OPT
192.168.188.31	192.168.188.42	DNS	Standard query 0xc182 A localhost OPT

Abbildung 4.50: Ausschnitt einer DNS-Kommunikation mittels DNS Cookies in Wireshark

1. Der Dig Client mit der IP 192.168.188.31 sendet eine DNS-Cookie-Anfrage an den Bind 9 Server mit der IP 192.168.188.42.
2. Der Bind 9 Server fügt seinen generierten Server Cookie in der DNS Response und versendet es an den Dig Client. In der Response ist der BADCOOKIE code gesetzt.
3. Der dig Client sieht den BADCOOKIE code und sendet erneut ein Query, wo der Client Cookie und der zuvor generierte Server Cookie enthalten sind.

In der Abbildung 4.51 ist der Aufbau der DNS-Cookie-Anfrage, die vom Dig Client erstellt wird, aus einer anderen Sicht aufgeführt.

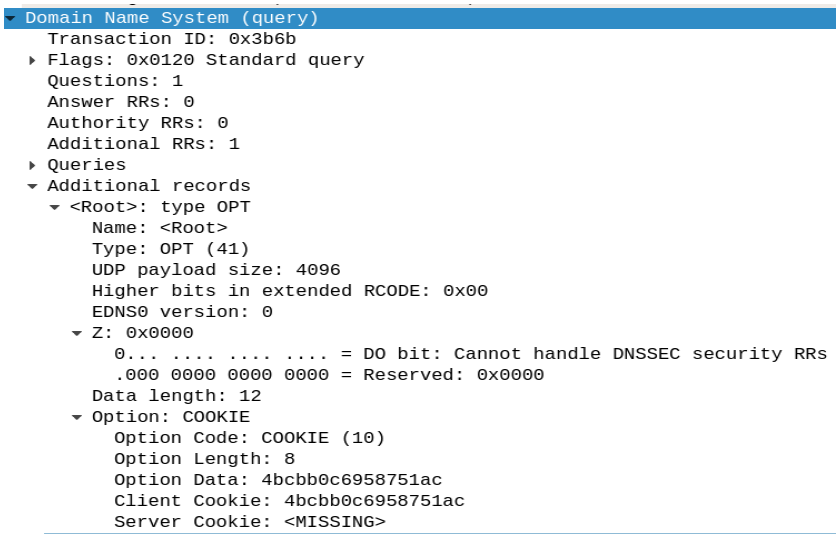

Abbildung 4.51: DNS Cookie Anfrage mit einem generierten Client Cookie

Die von BIND9 erzeugte Response, um eine Rückantwort zu provozieren, ist in der Abbildung 4.52 im Detail zu sehen.Anhand des extended RCODE ist erkennbar, dass die Antwort ein BADCOOKIE error code enthält. Extended RCODE bildet die oberen 8 Bits des erweiterten 12-Bit-RCODE im Additional-Abschnitt zusammen mit den in RFC1035 definierten 4 Bits. Die IANA hat für den BADCOOKIE den RCODE 23 zugewiesen und entspricht der Bit-Reihenfolge 000000010111 in der Antwort [20].

```
Transaction ID: 0x3b6b
▾ Flags: 0x8107 Standard query response, RRset exists
    1... .... .... .... = Response: Message is a response
    .000 0... .... .... = Opcode: Standard query (0)
    .... .0.. .... .... = Authoritative: Server is not an authority for domain
    .... ..0. .... .... = Truncated: Message is not truncated
    .... ...1 .... .... = Recursion desired: Do query recursively
    .... .... 0... .... = Recursion available: Server can't do recursive queries
    .... .... .0.. .... = Z: reserved (0)
    .... .... ..0. .... = Answer authenticated: Answer/authority portion was not authenticated by the server
    .... .... ...0 .... = Non-authenticated data: Unacceptable
    .... .... .... 0111 = Reply code: RRset exists (7)
  Questions: 1
  Answer RRs: 0
  Authority RRs: 0
  Additional RRs: 1
▸ Queries
▾ Additional records
  ▾ <Root>: type OPT
      Name: <Root>
      Type: OPT (41)
      UDP payload size: 4096
      Higher bits in extended RCODE: 0x01
      EDNS0 version: 0
    ▸ Z: 0x0000
      Data length: 28
    ▸ Option: COOKIE
```

Abbildung 4.52: BADCOOKIE Response in Wireshark

```
▾ Domain Name System (query)
    Transaction ID: 0xc182
  ▸ Flags: 0x0120 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 1
  ▸ Queries
  ▾ Additional records
    ▾ <Root>: type OPT
        Name: <Root>
        Type: OPT (41)
        UDP payload size: 4096
        Higher bits in extended RCODE: 0x00
        EDNS0 version: 0
      ▾ Z: 0x0000
          0... .... .... .... = DO bit: Cannot handle DNSSEC security RRs
          .000 0000 0000 0000 = Reserved: 0x0000
        Data length: 28
      ▾ Option: COOKIE
          Option Code: COOKIE (10)
          Option Length: 24
          Option Data: 4bcbb0c6958751ac0b9c9a4961cace3fada2962feebfab77
          Client Cookie: 4bcbb0c6958751ac
          Server Cookie: 0b9c9a4961cace3fada2962feebfab77
```

Abbildung 4.53: DNS-Anfrage nach Empfang des BADCOOKIE Code in Wireshark

Abschließend zeigt die Abbildung 4.53, wie der Dig Client den Client Cookie und den zuvor generierten Server Cookie in der Anfrage hinzufügt, nachdem es die BADCOOKIE-Option empfangen hat.

4.3.2.4 IP-Spoofing von DNS Paketen

In Abbildung 4.54 wurden die Anfragen an Destination Port 8888 mit Tshark aufgezeichnet um gespoofte Pakete mit TCPreplay im nächsten Schritt zu abzuspielen.

```
soufian@ubuntu:~/Desktop/Bachelor/DNS/DNS Cookie Ansatz/DNS_Cookies_Ansatz_LAN$ sudo tshark -i wlp4s0 -f "udp and dst port 8888" -w dnscookies_only_requests.pcap
Running as user "root" and group "root". This could be dangerous.
Capturing on 'wlp4s0'
12000 ^C
```

Abbildung 4.54: Paketaufzeichnung der DNS-Anfragen mit Tshark

Als nächsten Schritt zeigt die Abbildung 4.55, wie die DNS-Anfragen mit TCPreplay an den BIND9 Server gesendet worden sind. Zum Zeitpunkt der Einspielung durch TCPreplay bestand hinter den Anfragen kein DNS Client.

```
soufian@soufian-Aspire-ES1-572:~/Schreibtisch$ sudo tcpreplay -t -i wlp2s0 dnscookies_only_requests.pcap
[sudo] Passwort für soufian:
Actual: 12000 packets (1200000 bytes) sent in 0.462955 seconds
Rated: 2592044.5 Bps, 20.73 Mbps, 25920.44 pps
Flows: 9815 flows, 21200.76 fps, 12000 flow packets, 0 non-flow
Statistics for network device: wlp2s0
        Successful packets:        12000
        Failed packets:            0
        Truncated packets:         0
        Retried packets (ENOBUFS): 0
        Retried packets (EAGAIN):  0
```

Abbildung 4.55: Replay spoofed DNS-Cookies-Anfragen mit TCPreplay

Zu jeder gespooften Anfrage hat der BIND 9 Server ein BADCOOKIE Response wie in Abbildung 4.52 gesendet. Durch die BADCOOKIE Responses konnten keine Rückantworten provoziert werden, da die Quellen gespooft waren.

4.3.2.5 DNS Cookies Deployment im Internet

Der BIND9 wurde ebenfalls auf eine VM getestet, um live Daten zu sammeln und zu erkunden, wie erfolgversprechend der DNS Cookies im Internet wirklich ist. Die Auswertung der Daten werden im Kapitel 5 vorgestellt. Der BIND9 Server hängt an der öffentlichen IP 141.22.213.55 auf Port 53.

```
tcp        0      0 141.22.213.55:53        0.0.0.0:*               LISTEN      2941962/named
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN      199979/systemd-reso
udp        0      0 141.22.213.55:53        0.0.0.0:*                           2941962/named
udp        0      0 141.22.213.55:53        0.0.0.0:*                           2941962/named
udp        0      0 127.0.0.53:53           0.0.0.0:*                           199979/systemd-reso
```

Abbildung 4.56: BIND 9 Server horcht in der VM auf Port 53

Der DNS-Server hat knapp zwei Wochen lang auf den UDP Port 53 gehorcht, um Anfragen entgegenzunehmen. Mittels Tcpdump wurden der DNS-Verkehr aufgezeichnet. (siehe Abbildung 4.57).

```
benafia@caliban:~$ sudo tcpdump -vvv -i enp7s0 -w bind9Data.pcap host 141.22.213.55 and port 53
[sudo] password for benafia:
tcpdump: listening on enp7s0, link-type EN10MB (Ethernet), capture size 262144 bytes
^C2012 packets captured
2012 packets received by filter
0 packets dropped by kernel
```

Abbildung 4.57: Paketaufzeichnung von DNS-Verkehr mit Tcpdump

4.3.2.6 Testung von Standard Clients gegenüber DNS Cookie Ansatz

Wie bei dem TC-Flag-Ansatz wurden bei dem DNS-Cookies-Ansatz verschiedene Clients auf verschiedenen Betriebssystemen getestet. Bei Windows und Ubuntu wurden die Anfragen auf dem BIND9 Server auf die gleiche Weise wie bei dem TC-Flag-Ansatz umgeleitet. (siehe Abbildungen 4.39 und 4.40).

Beide Browser Chrome und Firefox haben in Ubuntu und Windows als Default ein no EDNS-Anfrage wie in Abbildung 4.58 durchgeführt, da der Additional-Abschnitt leer ist. Dementsprechend kann keine Adressvalidierung anhand eines DNS-Cookies-Ansatzes durchgeführt werden, da dies EDNS Clients voraussetzt.

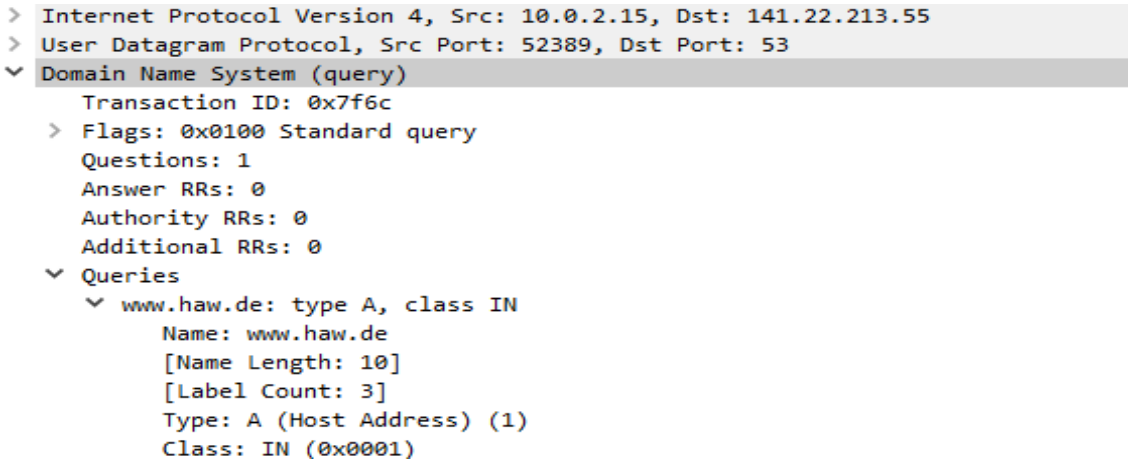

Abbildung 4.58: No EDNS Anfrage in Wireshark

Nslookup und der PowerShell DNS Resolver Resolve-DnsName unterstützen keine Anfragen mit DNS-Cookie-Option.

In macOS wurde der DNS-Cookie-Ansatz unter Safari ebenfalls getestet. Alle Anfragen wurden auf den BIND9 Server umgeleitet. Die gleiche Konfiguration kann der Abbildung 4.46 entnommen werden. Die Anfragen waren hier ebenfals no EDNS-Anfragen, sodass DNS Cookies nicht vollzogen werden konnte.

Tabelle 4.3: Browser-Verhalten gegenüber DNS-Cookies-Ansatz

Browser	**DNS Cookies Kommunikation**
Firefox in Linux/Windows	Nein
Chrome in Linux/Windows	Nein
Safari in MacOS	Nein
nslookup	Nein
Resolve-DnsNam	Nein

Die Tabelle 4.3 fasst nochmal die Ergebnisse zusammen. Sämtliche getesteten Clients haben die DNS-Cookies-Kommunikation nicht mitgemacht.

5 Evaluation

In diesem Abschnitt werden die Auswertungen für QUIC und DNS präsentiert. Die Ergebnisse von der Laborumgebung (Lokales Netzwerk) als auch Internet werden separat vorgestellt.

Weiterhin wurden Boxplots zur Darstellung der Ergebnisse benutzt. In Abbildung 5.1 sieht man ein Beispiel Boxplot, dass die einzelnen Elemente beschreibt, um die weiteren kommenden Boxplots besser interpretieren zu können.

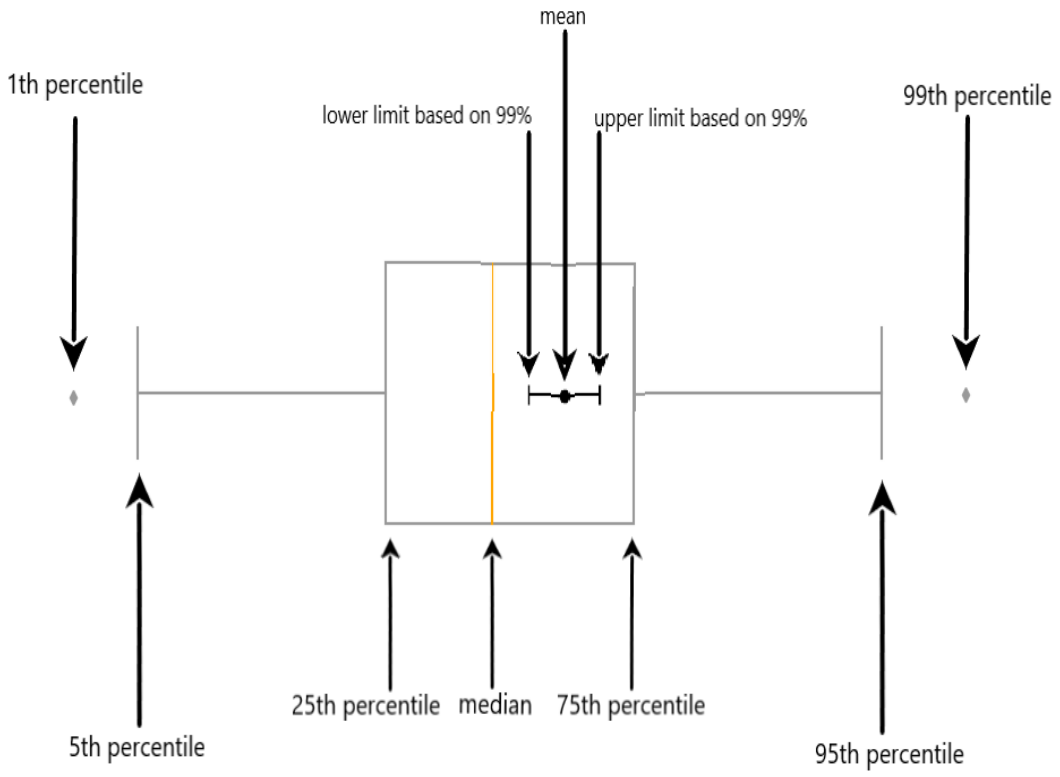

Abbildung 5.1: Beispiel-Boxplot dient zur Interpretation der Ergebnisse

- **Median**: der mittlere Wert des Datensatzes.
- **mean**: der Mittelwert des Datensatzes
- **lower/upper limit**: Konfidenzintervall für den mean. Der Mean liegt mit 99 Prozent Wahrscheinlichkeit in dem Bereich.
- **Unteres Quartil/25th percentile**: 25 Prozent der Werte liegen auf oder unter diesem Wert
- **Oberes Quartil/75th percentile**: 75 Prozent der Werte liegen auf oder unter diesem Wert.
- **Oberer Whisker/95th percentile**: 95 Prozent der Werte liegen auf oder unter diesem Wert.
- **Unterer Whisker/5th percentile**: 5 Prozent der Werte liegen auf oder unter diesem Wert.
- **Outliers**: Alles was über der 95th percentile oder unter der 5th percentile liegt, gilt als Ausreißer.

5.1 QUIC im lokalen Netzwerk

Im Folgenden wird die Anzahl der validen Anfragen und der gespooften Anfragen verglichen, beginnend mit der Umgebung im lokalen Netzwerk.

Tabelle 5.1: Valide QUIC-Quiche-Anfragen

Anzahl	**Retry-Anfragen**	**Retry-Antworten**	**Valide Tokens**	**keine Retry-Antworten**
6000	6000	6000	6000	0

In der Tabelle 5.1 ist das Ergebnis der validen generierten Quiche-Anfragen an den Kwik Server im lokalen Netzwerk aufgeführt. Es wurden vom Quiche Client 6.000 Anfragen erstellt, um ein statistisch signifikantes Ergebnis zu erhalten. Für jede dieser Anfragen gab es von Kwik eine Retry-Anfrage mit eindeutigen Token und für jede Retry-Anfrage gab es eine Retry-Antwort mit einem validen Token. Ein valider Token gilt, wenn der gesendete Token in der Retry-Anfrage mit den Token von der Retry-Antwort übereinstimmt.

Die Erwartungshaltung war, dass zu 6.000 Quiche-Anfragen 6.000 Retry-Antworten mit jeweils einem validen Token zurück antworten, da beide Seiten (Client und Server) voneinander wissen müssen, dass sie den gleichen Token haben. Diese Erwartungshaltung wurde anhand der Tabelle 5.1 bestätigt.

Tabelle 5.2: Gespoofte QUIC-Anfragen

Anzahl	**Retry-Anfragen**	**Retry-Antworten**	**Valide Tokens**	**keine Retry-Antworten**
6000	6000	0	0	6000

In der Tabelle 5.2 ist das Ergebnis der gespooften Anfragen im lokalen Netzwerk zu sehen. Es wurden 6.000 gespoofte Anfragen erstellt, um ein statistisch signifikantes Ergebnis zu erlangen. Für jede dieser Anfragen gab es von Kwik eine Retry-Anfrage. Für jede dieser Anfragen gab es keine Retry-Antwort und folglich 0 valide Tokens.

Die Erwartungshaltung war, dass zu 6.000 gespooften Anfragen jeweils eine Retry-Anfrage vom Kwik Server erstellt wird und keine einzige Retry-Antwort erhalten wird, da ein nicht existierender Client nicht in der Lage ist, auf eine Retry-Anfrage zu antworten. Diese Erwartungshaltung wurde anhand der Tabelle 5.2 bestätigt.

In einem Szenario, wo wir auf gespoofte Anfragen ein Retry senden, werden wir nie eine Antwort bekommen. In diesen Fällen müssen wir ein Timeout haben, bei dem wir uns sicher sind, dass wir damit nicht zugleich Clients invalidieren, von denen wir noch eine Antwort bekommen könnten. Aus diesem Grunde wurde im lokalen Netzwerk die Dauer der Retry-Antwort gemessen, also, wie lange es dauert, bis ein Client eine Retry-Antwort zurückschickt, nachdem der Server die Retry-Anfrage gesendet hat.

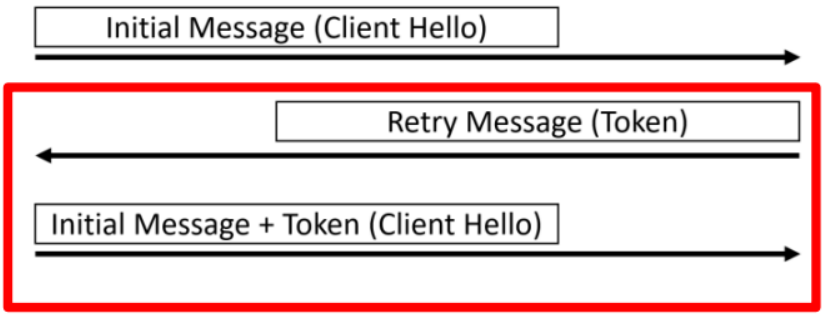

Abbildung 5.2: Verlauf einer Retry-Anfrage mit einer Retry-Antwort

Der rot umrahmte Teil in der Abbildung 5.2 verdeutlicht noch einmal, welcher Teil der Kommunikation gemessen wird. Die Messung umfasst die Dauer vom Senden einer Retry-Message (Retry-Anfrage) durch den Server bis hin zur Antwort des Clients auf die Retry-Anfrage.

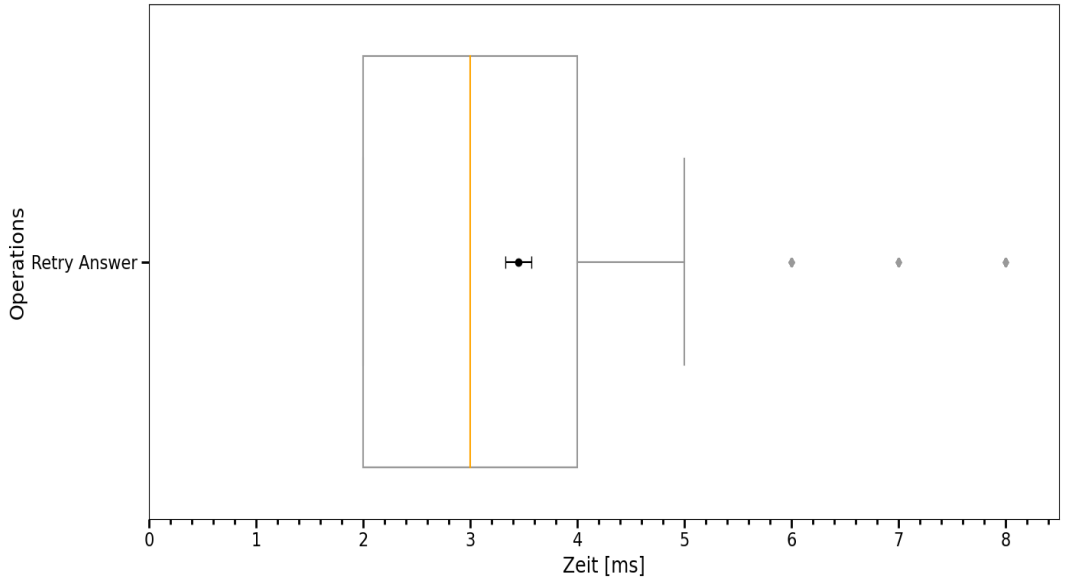

Abbildung 5.3: Messung der Dauer von Retry-Antworten von 6000 Verbindungen

Das Ergebnis ist in der Abbildung 5.3 zu sehen. Die Beschreibung der einzelnen Elemente des Boxplots ist in Abbildung 5.1 befindlich. Die durchschnittliche Dauer einer Retry-Antwort ist am Mean-Wert zu erkennen und beträgt 3,45 ms. Bei einer Datensatzgröße von 6.000 Verbindungen können wir sagen, dass das Konfidenzintervall für den mean bei einer Konfidenz-Größe von 99 Prozent zwischen 3.298 ms und 3.613 ms liegt.

Die orange Linie in der Mitte stellt den Median und gleichzeitig den Mittelpunkt der Daten dar. Der untere Whisker ist nicht sichtbar, da unter dem unteren Quartil der Box dem 1th percentile gleicht.

Aus den Messungen heraus können wir den Timeout-Wert im lokalen Netzwerk besser abschätzen, da die Messung zur Dauer der Retry-Antwort eine wichtige Größe für das Timeout ist.

5.2 QUIC im Internet

Alle gesammelten Daten im Internet wurden mit Tshark bzw. Tcpdump in PCAP Files aufgezeichnet und in Wireshark ausgewertet.

Tabelle 5.3: Valide QUIC-Anfragen im Internet

Anzahl	**Retry-Anfragen**	**Retry-Antworten**	**Valide Tokens**	**keine Retry-Antworten**
114	114	74	74	40

Die Tabelle 5.3 offenbart, dass der Kwik Server 114 QUIC-Anfragen basierend auf dem UDP-Protokoll empfangen konnte. Die QUIC-Anfragen kamen von 55 eindeutigen Quellen. 50 der eindeutigen Adressen sind drei Subnetze von Censys, Inc. zuzuordnen. Es gab weniger Adressen als Anfragen, da manche Quelladressen wiederholt Anfragen gestellt haben. An 114 QUIC-Anfragen konnten wir 114 Retry-Anfragen senden. Bei 74 von 102 Retry-Anfragen konnten wir eine Retry-Antwort provozieren.

Durch https://ipinfo.io/ wurden die IP-Adressen nach AS-Nummern aufgelöst. 5.4 zeigt die Verteilung der Retry-Antworten auf 6 AS-Nummern.

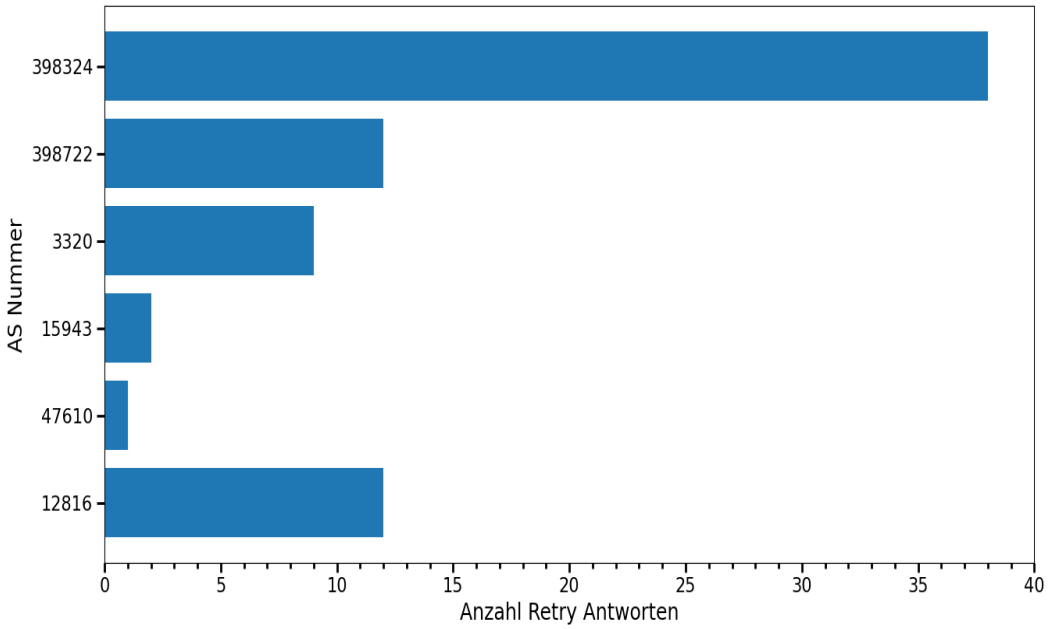

Abbildung 5.4: Retry Antworten nach AS Nummern

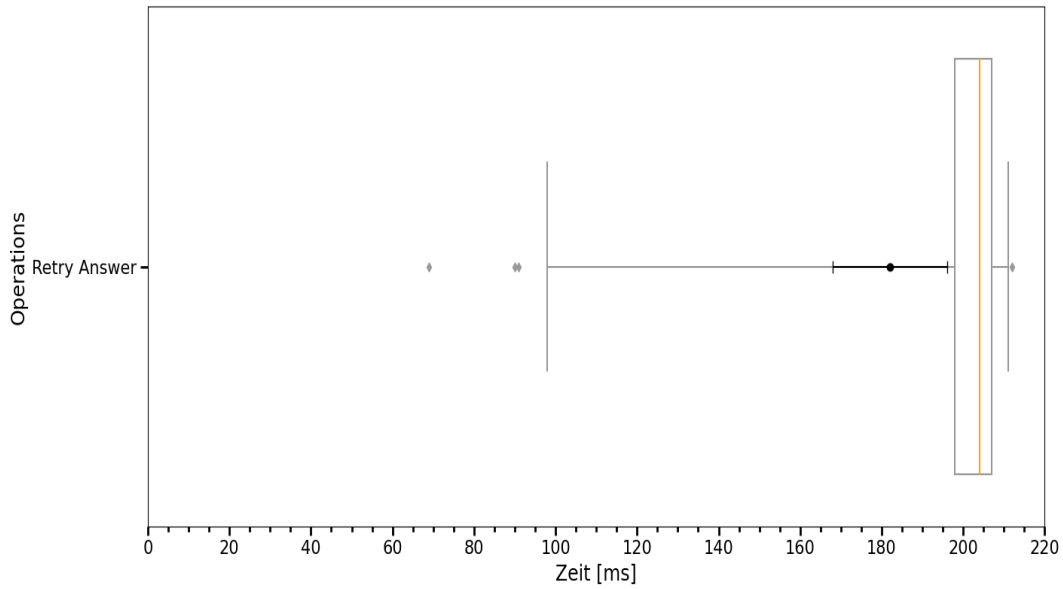

Abbildung 5.5: Messung der Retry-Antwortdauer im Internet bei 62 Verbindungen

Das Ergebnis ist in der Abbildung 5.5 zu sehen. Die Beschreibung der einzelnen Elemente des Boxplots ist in Abbildung 5.1 befindlich. Die durchschnittliche Dauer einer Retry-Antwort ist am Mean-Wert erkennbar und beträgt 182 ms. Bei einer Datensatzgröße von 62 Verbindungen können wir sagen, dass das Konfidenzintervall für den Mean bei einer Konfidenz-Größe von 99 Prozent zwischen 168 ms und 196 ms liegt. Die orange Linie in der Mitte stellt den Median und gleichzeitig den Mittelpunkt der Daten dar.

Aus den Messungen heraus können wir den Timeout-Wert im Internet besser abschätzen, da die Messung zur Dauer der Retry-Antwort eine wichtige Größe für das Timeout darstellt.

Der Retry-Ansatz konnte durchaus überzeugen, da wir bei mehr als 50 Prozent der Anfragen durch eine Retry-Anfrage eine Antwort provozieren konnten, um eine Validierung durchzuführen.

5.3 DNS im lokalen Netzwerk

Beide DNS-Ansätze wurden in im lokalen Netzwerk mit validen Anfragen als auch gespooften Anfragen getestet.

Tabelle 5.4: Valide DNS-Anfragen für TC-Ansatz

DNS-Anfragen	TC-Anfragen	TC-Antworten	keine TC-Antworten
6000	6000	6000	0

In der Tabelle 5.4 wurden 6.000 valide DNS-Anfragen für den TC-Ansatz im lokalen Netzwerk generiert. Für jede TC-Anfrage haben wir eine TC-Antwort über TCP zurückbekommen. Die Erwartungshaltung war, dass wir mit jeder TC-Anfrage eine Rückantwort provozieren und eine Kommunikation mit der Gegenseite etablieren können. Das Ergebnis aus 5.4 hat die Erwartungshaltung bestätigt.

Tabelle 5.5: Valide DNS-Anfragen für DNS-Cookies-Ansatz

DNS Client-Cookie Anfragen	Badcookie Anfragen	Badcookie Antworten	keine Badcookie Antworten
6000	6000	6000	0

In der Tabelle 5.5 wurden 6.000 valide DNS-Anfragen für den DNS-Cookie-Ansatz im lokalen Netzwerk generiert. Jede DNS-Anfrage wurde an den BIND9 Server mit einem Client Cookie versendet. Unser konfigurierte BIND9 Server antwortete auf jede Anfrage mit einem BADCOOKIE und für jede BADCOOKIE-Anfrage konnten wir den Client erfolgreich zu einer Rückantwort provozieren.

Im lokalen Netzwerk konnten wir unter kontrollierten Bedingungen feststellen, dass beide Ansätze erfolgreich eine Rückantwort provozieren und somit valide Anfragen erfolgreich erkennen. Als Nächstes wurden gespoofte Anfragen mit den beiden Ansätzen getestet.

Tabelle 5.6: Gespoofte DNS-Anfragen für TC-Ansatz

DNS-Anfragen	TC-Anfragen	TC-Antworten	Keine TC-Antworten
6000	6000	0	6000

In der Tabelle 5.6 wurden 6.000 DNS-Anfragen mit gefälschten IPs generiert, um den TC-Flag-Ansatz zu testen. Jede gefälschte DNS-Anfrage wurde mit einer TC-Anfrage beantwortet und 0 TC-Antworten wurden erhalten.

Tabelle 5.7: Gespoofte DNS-Anfragen für DNS-Cookies-Ansatz

DNS Client-Cookie Anfragen	**Badcookie Anfragen**	**Badcookie Antworten**	**keine Badcookie Antworten**
6000	6000	0	6000

In der Tabelle 5.7 wurden 6.000 DNS-Client-Cookie-Anfragen mit gefälschten IPs generiert, um den DNS-Cookie-Ansatz zu testen. Jede gefälschte DNS-Anfrage wurde mit einer BADCOOKIE-Anfrage beantwortet und 0 BADCOOKIE-Antworten wurden erhalten.

Die Erwartungshaltung war, dass bei beiden Ansätzen keine Antwort vom DNS Client erlangt wird, da ein nicht existierender Client nicht in der Lage ist, auf eine TC- oder BADCOOKIE-Anfrage zu antworten. Diese Erwartungshaltung wurde anhand der Tabellen 5.6 und 5.7 bestätigt.

Als Nächstes wurde das Antwortverhalten wie bei QUIC zwischen Client und Server im lokalen Netzwerk untersucht. Aus diesem Grund wurde die Dauer der BADCOOKIE- und Truncated-Antwort gemessen, also, wie lange es dauert, bis ein Client auf eine BADCOOKIE- oder eine Truncated-Anfrage eine Antwort zurückschickt.

Die BADCOOKIE- und Truncated-Antwort ist in der Abbildung 5.6 anhand der Boxplots im Vergleich zu sehen. Die Beschreibung der einzelnen Elemente des Boxplots ist in Abbildung 5.1 befindlich. Bei einer BADCOOKIE-Antwort besteht die durchschnittliche Dauer 2,90 ms und bei einer Truncated-Antwort 2.29 ms. Das Konfidenzintervall für das BADCOOKIE Answer Delay liegt zwischen 2.85 ms und 2.95 ms. Bei dem Truncated Answer Delay liegt es zwischen 2.26 ms und 2.32 ms.

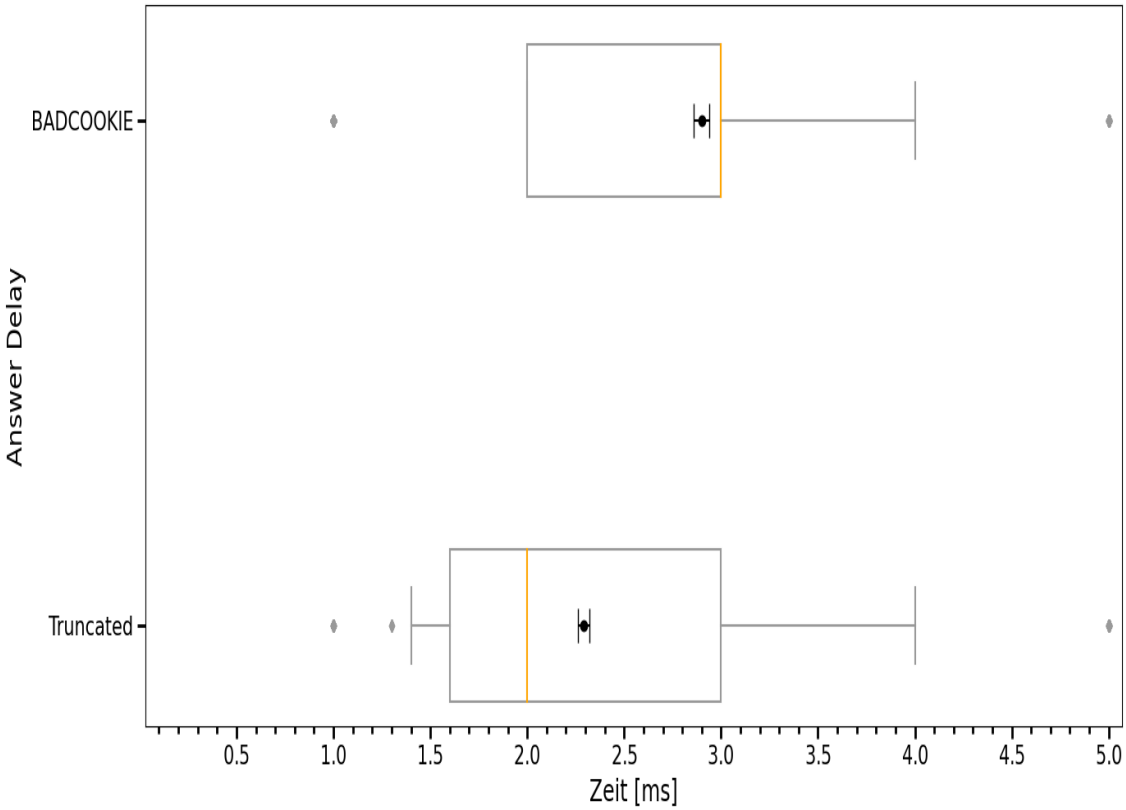

Abbildung 5.6: BADCOOKIE und Truncated Antwortdauer im Vergleich anhand 6000 Messungen

Der Median überlappt bei BADCOOKIE Answer Delay dem oberen Quartil und liegt bei 3ms. Dies geschieht, da ein großer Anteil der Messwerte gleich hohe Werte im Datensatz aufweist. Der Median liegt beim Truncated-Ansatz bei 2ms. Beide Operationen haben zudem Ausreißer und werden mit Punkten markiert. Der Timeout-Wert kann auch bei den beiden Operationen gleich sein, da die durchschnittliche Dauer der Antworten sich um Zehntel Millisekunden unterscheidet.

5.4 DNS im Internet

Alle gesammelten Daten im Internet wurden mit Tshark bzw. Tcpdump in PCAP Files aufgezeichnet und in Wireshark ausgewertet.

Tabelle 5.8: Valide DNS-Anfragen für TC-Ansatz

DNS-Anfragen	TC-Anfragen	TC-Antworten	keine TC-Antworten
560	532	7	525

Die Tabelle 5.8 zeigt, dass unser implementierter DNS-Server 560 DNS-Anfragen, basierend auf dem UDP-Protokoll, empfangen hatte. Die DNS-Anfragen kamen von 128 eindeutigen Quellen. Es gab weniger Adressen als Anfragen, da manche Quelladressen wiederholt Anfragen gestellt haben. 532 von 560 DNS-Anfragen konnten mit einer TC-Anfrage beantwortet werden. Die restlichen 28 DNS-Anfragen waren Malformed-Pakete auf die keine TC-Anfragen geschickt werden konnte. Lediglich bei 7 von 532 TC-Anfragen konnte eine TC-Antwort provoziert werden.

Die 7 TC-Antworten kamen von 2 eindeutigen IP-Adressen. Eine IP-Adresse hat wiederholt DNS-Anfragen gestellt. Durch https://ipinfo.io/ wurden die IP-Adressen nach AS Nummern aufgelöst. 5.7 zeigt die Verteilung der TC-Antworten auf die beiden Quellen.

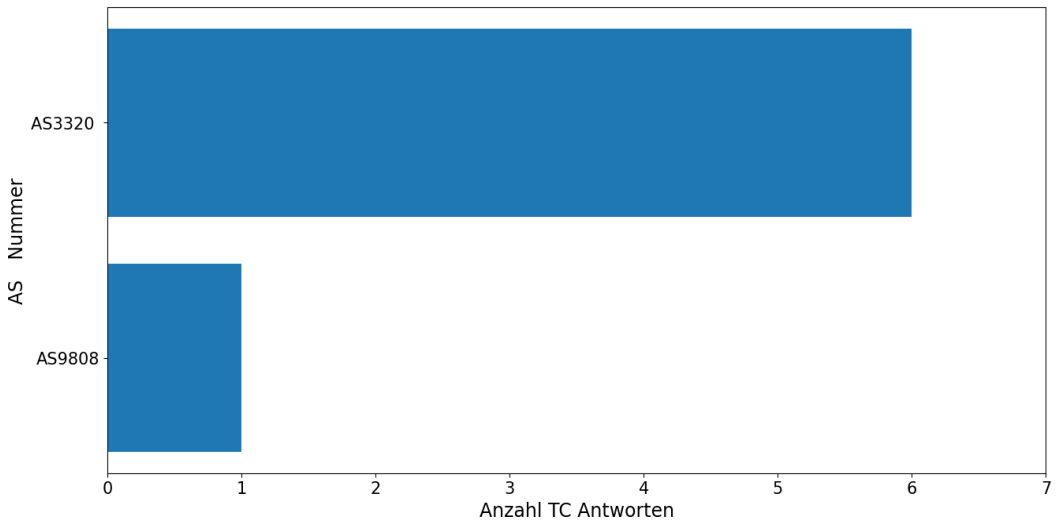

Abbildung 5.7: TC-Antworten nach eindeutigen Adressen

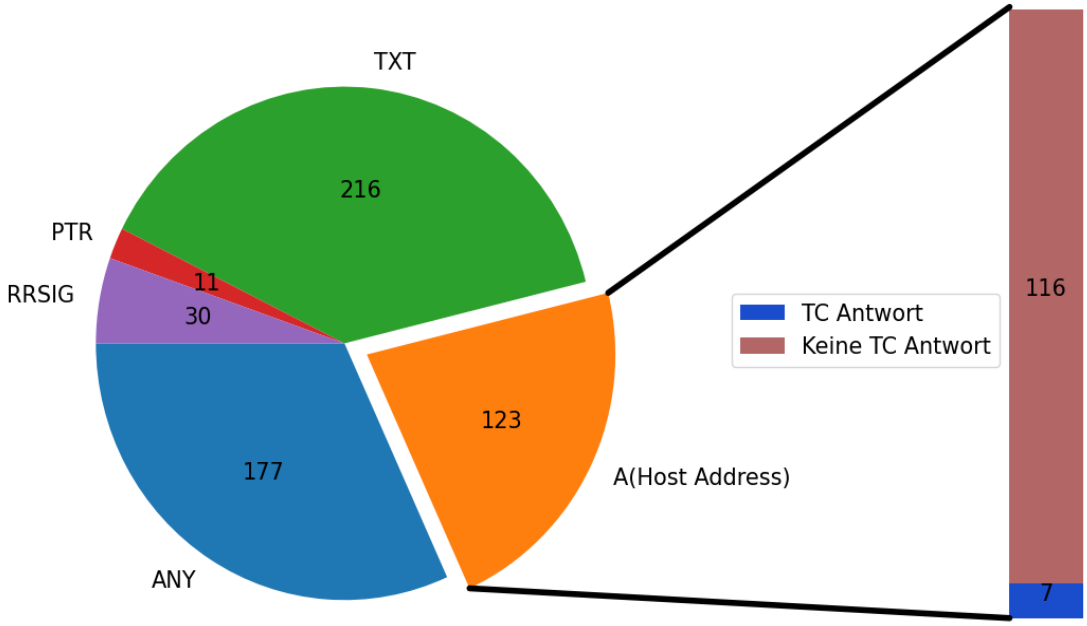

Abbildung 5.8: DNS-Anfragen nach Query Type klassifiziert

In der Abbildung 5.8 wurden die DNS-Anfragen nach dem Query Type klassifiziert. Es wurde festgestellt, dass alle TC-Antworten, die erhalten wurden, von Quelladressen kamen, die ein DNS-Query vom Typ **A (Host Address)** gesendet haben. Alle anderen Query Types haben dagegen keine TC-Antwort zurückgegeben. Die Mehrheit der DNS-Anfragen waren vom TYP TXT. 183 der TXT-Anfragen waren Anfragen, um die Version eines BIND DNS-Servers zu identifizieren.

Tabelle 5.9: Gesammelte DNS-Anfragen für DNS Cookies Ansatz im Internet

DNS Anfragen	**Client-Cookie-Anfragen**	**Badcookie-Anfragen**	**Badcookie-Antworten**
473	0	0	0

Die Tabelle 5.9 zeigt die gesammelten DNS-Anfragen für den DNS-Cookie-Ansatz und dessen Auswertung. Der BIND9 Server hat 473 DNS Anfragen, basierend auf dem UDP-

Protokoll empfangen. Die DNS-Anfragen kamen von 138 eindeutigen Quellen. Einige dieser Quelladressen haben wiederholt Anfragen gestellt. Keiner dieser Anfragen hatte die Cookie-Option unterstützt, dementsprechend konnten keine BADCOOKIE-Anfragen gesendet oder BADCOOKIE-Antworten erhalten werden.

Tabelle 5.10: EDNS Support anhand der gesammelten Daten im Internet

DNS-Anfragen	**EDNS**	**NO EDNS**
473	3	470

In der Tabelle 5.10 wurde der EDNS Support dargestellt. EDNS, das eine Voraussetzung für DNS-Cookies ist, wurde nur in 3 Anfragen unterstützt, wobei eine Quelleadresse zwei von den drei EDNS-Anfragen an uns gerichtet hat.

Beide Ansätze konnten durchgehend nicht im Internet überzeugen, da in beiden Ansätzen eine relative geringe Anzahl bis keine Antwort provoziert werden konnte, um die Quelle zu validieren. Im Falle des TC-Ansatzes haben weniger als 2 Prozent eine TC-Antwort zurückgegeben. Ein Grund, warum Clients nicht auf den TC-Ansatz zurück antworten, besteht möglicherweise darin, dass z. B. die Mehrheit der TXT-Anfragen die BIND Server Version herausfinden wollten und diesbezüglich eine Überschreitung der Nachrichtengröße von mehr als 512 Byte zum Setzen des TC-Bits (Truncation) sehr unwahrscheinlich ist. Ein weiterer Grund ist, dass die RRSIG- sowie ANY-Anfragen normalerweise gespoofte Anfragen sind und demnach nicht zurück antworten [30]. Bei DNS Cookies konnten keine Antworten provoziert werden, da keiner der DNS Clients die Client-Cookie-Option, geschweige denn EDNS, unterstützt hat.

Der TC-Ansatz hat zumindest den Vorteil, dass es kein EDNS Support von beiden Seiten bedarf. Bei DNS Cookies hat keine Quelle Client Cookie in den Anfragen unterstützt, um überhaupt eine Validierung zu starten. Diese mangelnde Unterstützung von Client Cookies lässt sich zugleich in den Beobachtungen von [12] bestätigen

6 Fazit und Ausblick

Das Ziel dieser Arbeit war es, den Rückfragemechanismus für UDP-basierte Protokolle zu entwickeln. Es wurden zwei Protokolle vorgestellt, anhand deren der Rückfragemechanismus eingebettet und getestet wurde. Es ist zu erkennen, dass die entwickelten Ansätze in QUIC und DNS im lokalen Netzwerk durchaus Rückantworten provozieren konnten, allerdings war die Erfolgsquote für DNS im Internet sehr mager.

IP-Spoofing wird weiterhin von Cyberkriminellen als Waffe eingesetzt, um Denial-of-service oder Amplifikation-Attacken auf UDP-basierte Protokolle durchzuführen. Folglich ist es weiterhin notwendig, den Rückfragemechanismus in anderen UDP-basierten Protokollen auszuprobieren.

Vorausschauend sollte der Rückfragemechanismus auf weitere Netzwerkprotokolle entwickelt werden. Es gibt eine Reihe an UDP-Protokollen, die von CISA als potenzielle Angriffsvektoren identifiziert werden [2]. Dazu gehören typische Amplifier wie NTP, der als Nächstes untersucht werden könnte.

Literaturverzeichnis

[1] 3rd, D. E. ; Andrews, M.: Domain Name System (DNS) Cookies / IETF. May 2016 (7873). – RFC

[2] Agency, Cybersecurity Infrastructure S.: *UDP-Based Amplification Attacks*. https://www.cisa.gov/uscert/ncas/alerts/TA14-017A. – Accessed: 2022-02-06

[3] Biondi, Philippe: *Scapy*. https://scapy.net/. 2008. – Accessed: 2022-02-06

[4] Caida: *Spoofer Project*. https://www.caida.org/projects/spoofer/. – Accessed: 2022-02-04

[5] Cherenson, Andrew: *nslookup*. https://linux.die.net/man/1/nslookup. 2010. – Accessed: 2022-02-06

[6] Combs, Gerald: *tshark*. https://www.wireshark.org/docs/man-pages/tshark.html. 1998-2021. – Accessed: 2022-02-06

[7] Combs, Gerald: *Wireshark*. https://www.wireshark.org/. 1998-2021. – Accessed: 2022-02-06

[8] Consortium, Internet S.: *dig - DNS lookup utility*. https://linux.die.net/man/1/dig. 2013. – Accessed: 2022-02-06

[9] consortium, Internet S.: *Bind9*. https://www.isc.org/bind/. 2021. – Accessed: 2022-02-06

[10] Corporation, Savarese Software R.: *RockSaw Raw Socket Library for Java*. https://www.savarese.com/software/rocksaw/. – Accessed: 2022-02-06

[11] Damas, J. ; Graff, M. ; Vixie, P.: Extension Mechanisms for DNS (EDNS(0)) / IETF. April 2013 (6891). – RFC

[12] Davis, Jacob ; Deccio, Casey: *A Peek into the DNS Cookie Jar*. S. 302–316, 03 2021. – ISBN 978-3-030-72581-5

[13] Diggory Blake, the Mozilla C. ; contributors, Rustup: *Rustup: the Rust toolchain installer*. https://github.com/rust-lang/rustup. – Accessed: 2022-02-06

[14] Doornbosch, Peter: *Kwik*. https://github.com/ptrd/kwik. 2021. – Accessed: 2022-02-04

[15] Eddy, W.: TCP SYN Flooding Attacks and Common Mitigations / IETF. August 2007 (4987). – RFC

[16] Ehrenkranz, Toby ; Li, Jun: On the State of IP Spoofing Defense. In: *ACM Trans. Internet Technol.* 9 (2009), may, Nr. 2, S. 444–458. – URL https://doi.org/10.1145/1516539.1516541. – ISSN 1533-5399

[17] Ferguson, P. ; Senie, D.: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing / IETF. May 2000 (2827). – RFC

[18] Group, The T.: *tcpdump*. https://www.tcpdump.org/. 2010. – Accessed: 2022-02-06

[19] Herrmann, Dominik: Beobachtungsmöglichkeiten im Domain Name System: Angriffe auf die Privatsphäre und Techniken zum Selbstdatenschutz. In: Holldobler, Steffen (Hrsg.) ; al. et (Hrsg.): *Ausgezeichnete Informatikdissertationen 2014*. Bonn : Gesellschaft für Informatik, 2015, S. 91–100

[20] IANA: *DNS RCodes*. https://www.iana.org/assignments/dns-parameters/dns-parameters.xhtml#dns-parameters-6. – Accessed: 2022-02-06

[21] Iyengar, J. ; Thomson, M.: QUIC: A UDP-Based Multiplexed and Secure Transport / IETF. May 2021 (9000). – RFC

[22] jetbrains: *IntelliJ IDEA*. https://www.jetbrains.com/idea/. – Accessed: 2022-02-06

[23] Kappes, Martin: *Netzwerk- und Datensicherheit : eine praktische Einführung*. Wiesbaden: Springer Vieweg, 2013. – 148–149 S. – URL

https://link.springer.com/book/10.1007/978-3-8348-8612-5#siteedition-academic-link. – ISBN 978-3-8348-8612-5

[24] Klassen, Fred: *tcpreplay*. https://tcpreplay.appneta.com. 2000-2018. – Accessed: 2022-02-06

[25] Majkowski, Marek: *Memcrashed - Major amplification attacks from UDP port 11211*. https://blog.cloudflare.com/memcrashed-major-amplification-attacks-from-port-11211/. 2018. – Accessed: 2021-02-05

[26] Microsoft: *Resolve-DnsName*. https://docs.microsoft.com/en-us/powershell/module/dnsclient/resolve-dnsname?view=windowsserver2022-ps. 2019. – Accessed: 2022-02-06

[27] Mockapetris, P.V.: Domain names - concepts and facilities / IETF. November 1987 (1034). – RFC

[28] Mockapetris, P.V.: Domain names - implementation and specification / IETF. November 1987 (1035). – RFC

[29] Nawrocki, Marcin ; Hiesgen, Raphael ; Schmidt, Thomas C. ; Wählisch, Matthias: QUICsand: Quantifying QUIC Reconnaissance Scans and DoS Flooding Events. In: *Proc. of ACM Internet Measurement Conference (IMC)*. New York : ACM, 2021, S. 283–291. – URL https://doi.org/10.1145/3487552.3487840

[30] Nawrocki, Marcin ; Jonker, Mattijs ; Schmidt, Thomas C. ; Wählisch, Matthias: The Far Side of DNS Amplification: Tracing the DDoS Attack Ecosystem from the Internet Core. In: *Proc. of ACM Internet Measurement Conference (IMC)*. New York : ACM, 2021, S. 419–434. – URL https://doi.org/10.1145/3487552.3487835

[31] NEWMAN, LILY H.: *GitHub Survived the Biggest DDoS Attack Ever Recorded*. https://www.wired.com/story/github-ddos-memcached/. 2018. – Accessed: 2022-02-05

[32] Nottingham, M. ; McManus, P. ; Reschke, J.: HTTP Alternative Services / IETF. April 2016 (7838). – RFC

[33] Pardue, Lucas: *Quiche*. https://github.com/cloudflare/quiche. 2021. – Accessed: 2022-02-04

[34] Postel, J.: User Datagram Protocol / IETF. August 1980 (768). – RFC

[35] Shreedhar, Tanya ; Panda, Rohit ; Podanev, Sergey ; Bajpai, Vaibhav: Evaluating QUIC Performance over Web, Cloud Storage and Video Workloads. In: *IEEE Transactions on Network and Service Management* PP (2021), 12, S. 1–16

[36] Thomson, M. ; Turner, S.: Using TLS to Secure QUIC / IETF. May 2021 (9001). – RFC

[37] Van Winkle, L.: *Hands-On Network Programming with C*. Packt Publishing, 2019. – URL https://www.oreilly.com/library/view/hands-on-network-programming/9781789349863/. – ISBN 9781789349863

[38] Wang, Haining ; Jin, Cheng ; Shin, Kang G.: Defense Against Spoofed IP Traffic Using Hop-Count Filtering. In: *IEEE/ACM Transactions on Networking* 15 (2007), Nr. 1, S. 40–53

[39] Wellington, Brian: *dnsjava*. https://kentros.github.io/dnsjava/. 2004. – Accessed: 2022-02-06

[40] Yamada, Kaito: *Pcap4J*. https://www.pcap4j.org/. – Accessed: 2022-02-06

A Anhang

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original