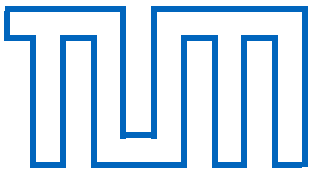


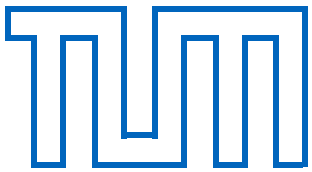
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Measuring Google QUIC Connection Establishment Times

Bernhard Jaeger



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Measuring Google QUIC Connection Establishment Times

Google QUIC-Verbindungsaufbauzeiten messen

Author:	Bernhard Jaeger
Supervisor:	Prof. Dr. Jörg Ott
Advisor:	Dr. Vaibhav Bajpai
Submission Date:	15.08.2018

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.08.2018 Bernhard Jaeger

Abstract

TCP has been the most prevalent connection establishment protocol in the Internet for a long time. Since TCP does not support encryption inherently it later got extended with SSL/TLS. This introduced an additional overhead in connection establishment time but ensured backwards compatibility. This overhead was of less concern in the past since bandwidth was the limiting factor of the Internet. With increasing bandwidth of the Internet the focus shifted back to reducing round trips as connection speed is becoming a limiting factor which can not be improved because the speed of light is constant. Over the last years attempts where made to improve TCP such as TCP Fast Open [5]. They did not see widespread adoption. In 2012 Google started developing their own Connection Protocol called Quick UDP Internet Connections (QUIC). Due to having control over both client and server infrastructures (Chrome, Android, Youtube.com ,Google.com, ...) they were able to deploy their protocol at a large scale and in 2017 as reported by Google an approximately 7% of the Internet traffic was using the QUIC protocol [23]. The QUIC protocol is currently going through a standardisation process by the IETF [12]. This thesis aims to evaluate the connection latency of the Google QUIC protocol as it is deployed on production servers. To achieve this we wrote two measurement test to evaluate the connection establishment times of QUIC and TCP/TLS. Using these programs we collected data to compare both protocols from two vantage points in Munich Germany. We also measured how many websites already adopted the new TLS 1.3 and how the protocol compares to TLS 1.2 and QUIC. Furthermore we measured the impact IPv6 had on the connection establishment times of these protocols. We find that QUIC connects faster than TLS 1.2 and TLS 1.3 in nearly all cases but the degree depended on the measurement environment. TLS 1.3 connected faster than TLS 1.2 in almost all cases and IPv6 had little to no impact on the connection times.

Contents

1 Introduction

1.1 Motivation

Google QUIC is an Internet protocol developed by Google with the aim to replace the commonly used TCP + TLS protocol. As the name suggests one of it's goals is to enable faster connection establishments. Google presented this protocol at SIGCOMM 2017 [23] stating that they deployed QUIC on their servers and clients at an Internet scale and that an estimated of 7% of the Internet traffic is using QUIC. One goal of this thesis is to analyse the connection latency of QUIC as it is deployed in the Internet and compare it to the TCP + TLS protocol. The Internet Engineering Task Force also started working on standardizing the QUIC protocol [12]. In this thesis we did not study IETF QUIC and focused on Google QUIC as it is used in production scenarios.
Another interesting development in the security protocol area is TLS version 1.3 which is currently an Internet-Draft [27]. One goal of this thesis is to determine the penetration of TLS 1.3 in the Internet and to evaluate it's latency compared to it's predecessor TLS 1.2 and Google QUIC.
IPv6 has become an Internet Standard in 2017 [9] after being a Draft Standard since 1998 [31] and it has seen increasing adoption over the last years according to Google statistics [11]. Another goal of this thesis is to find out if IPv6 has any influence on the connection establishment times of Google QUIC or TCP/TLS.

1.2 Research Questions

Several questions were researched in this work to measure and compare the connection latency of Google QUIC as well as the new TLS 1.3 protocol. The thesis engaged in following research Questions:

RQ 1: How many websites adopted TLS 1.3 already?

TLS 1.3 is currently an Internet-Draft [27]. Multiple libraries and websites already started to implement/deploy it. To compare QUIC to TLS 1.3 we need a list of websites that already support it.

RQ 2: How much faster/slower does TCP/TLS 1.3 connect compared to TCP/TLS 1.2?

TLS 1.3 aims to improve TLS 1.2 in terms of security and connection speed. We want to determine how much faster TLS 1.3 connects.

RQ 3: How much faster/slower does QUIC connect compared to TCP/TLS 1.2?

QUIC aims to replace the TCP/TLS protocol. We are interested to see how much latency reduction QUIC gains compared to TLS 1.2.

RQ 4: How much faster/slower does QUIC connect compared to TCP/TLS 1.3?

TCP/TLS 1.3 will likely become the new standard for encryption replacing TCP/TLS 1.2. If QUIC wants to replace TCP it needs to outperform TCP/TLS 1.3 as well.

1.3 Research Contributions

The analysis and research in this thesis contribute to the performance evaluation of the Internet protocols Google QUIC, TCP/TLS 1.2 and TCP/TLS 1.3. Furthermore we analysed the adoption of TLS 1.3 and the influence of IPv6 on the protocols.

Our contributions are:

1. We developed one program called `quic_perf` to measure the connection establishment times of Google QUIC. The DNS lookup code of this program was merged in the LSQUIC test client. We also created another program called `tls_perf` to measure TCP/TLS 1.2 and TCP/TLS 1.3. They are available at [19] and [18]

2. We measured the adoption of TLS 1.3 In Alexa Top 1000 at least 10,7% of websites adopted the protocol. In Alexa Top 100.000 at least 15,9% of websites adopted the protocol. For more details see 3.2.

3. We compared the latency of TLS 1.3 vs TLS 1.2. TLS 1.3 gains a latency reduction of ~10 ms over TLS 1.2 ~80% of the time in our measurements. IPv6 had similar results compared to IPv4 , see 4.2.

4. We compared the latency of QUIC vs TLS 1.2. QUIC had a latency reduction of ~30 ms over TLS 1.2 ~60% of the time and an advantage of ~10 ms ~40% of the time in one of the test environments. On the second test environment QUIC had an advantage of ~5 ms ~40% of the time and ~20 ms ~60% of the time. Also on the second test environment QUIC gains a ~20 ms advantage ~90 % of the time with IPv6. For more details see chapter 3.3 and 4.3.

5. We compared the latency of QUIC vs TLS 1.3. QUIC connects ~10 ms faster than TLS 1.3 ~90% of the time in one test environment and ~2 ms ~99% of the time on the second test environment. IPv6 had similar results compared to IPv4. For more details see 4.4.

As with other work this thesis also has limitations which are discussed in Chapter 5 together with the conclusion. Several lessons we learned while writing the thesis are explained in chapter 5 and further research opportunities are also outlined in chapter 5. Chapter 6 describes how to reproduce the results shown in this work.
Our methodology is described in chapter 3 an evaluation of our result is done in chapter 4.

2 Related Work

Despite the additional overhead that increases latency encrypted connections see increased adoption over the last years [26]. Adoption numbers vary between different Top Lists and measurement tools. One measurement in [10] using the Mozialla Observatory Tool in 02.2017 showed that 40% of Alexa Top 1 Million supported HTTPS. More popular website had a higher adoption rate with 87% of Alexa Top 100 having adopted HTTPS in this measurement. In contrary the Google Top 100 measured with Googlebot only showed an adoption rate of 54%. These and other inconsistencies of Top Lists like Alexa were investigated in [32]. They give several recommendations for using Top Lists such as that the list structure must match the study purpose, the stability of the list should be considered and the list use should be well documented.

Google QUIC is one of the new technologies that tries to mitigate the latency overhead of encrypted connections. As Google QUIC is relatively new and frequently changing not a lot of research has been done regarding it. One of these researches by Yong Cui *et al.* [7] describes design approaches of QUIC. QUIC is a protocol defined on top of UDP that can be implemented in user space which enables faster deployment and update cycles. It is a 1 Round Trip Time (RTT) protocol that allows for 0 RRT connections if the connection partners already know each other. QUIC therefore saves two RTT compared to TCP/TLS 1.2 and one RTT when compared to TCP/TLS 1.3. QUIC also supports multiplexing of simultaneous HTTP streams and modern loss-recovery mechanisms like F-RTO and Early Retransmit.

The first experimental analysis of QUIC was done by [4] in 2015. Their test environment consists of a webserver that only links pictures and Google Chrome on the client site. They show that in their experimental setting QUIC (version 21) outperforms HTTP/1.1 over TLS 1.2 in terms of page load times. QUIC also achieves a higher goodput for small bottleneck buffer sizes. When the network was over-buffered the protocols equally shared the link capacity.

An analysis of the adoption of QUIC in the Wild was done by Jan Rüth *et al.* in [30]. They observed that between 2.6% and 9.1% of Internet traffic used QUIC depending on the vantage point. They also provide weekly statistics about QUIC support in different networks at [28].

Examination of the QUIC protocol can be challenging as studies might be outdated before release due to the rapid development. In [21] they tried to address this challenge.

They measured QUIC using Google Chrome from both Desktop and mobile environments connecting to an Amazon EC2 webserver with Apache. Their findings include that QUIC is unfair when sharing bandwidth with TCP since it increases the congestion window more aggressively. In their tests QUIC outperformed TCP + HTTPS in almost every scenario. The performance gains are decreased however on older phones from 2013 due to application-layer packet processing and encryption. QUIC spends a lot of time in the "Application Limited" state in these mobile environments (58%) compared to desktop (7% of the time) due to QUIC running in userspace process.
The paper [25] from 2016 looked at page loading times comparing HTTP, SPDY and QUIC. As other works they used Google Chrome in their test environment on the client side. On the server side they installed four different web pages on Google websites. Between the two they installed a sharper server to vary network conditions. They concluded that none of the protocols is clearly better than the other two. Who performs best is dependent on network conditions. On good network conditions all protocols had similar page load times. QUIC performed poor in scenarios with high bandwidth and big files to download due to the packet pacing mechanism in QUIC not reaching the maximum network capacity. QUIC performed well in scenarios with high RTT and lots of small objects to download.
Page Load Time was also investigated in [6] comparing HTTP/2 over TCP/TLS vs QUIC. They used a tool called perfy that configures and launches Chromium as client and set up servers using GO-Quic and Golang libraries. They also had a remote testbed from home connecting to production servers using LAN as well as 4G. They saw in their tests that QUIC outperformed TCP/TLS in unstable environments such as wireless mobile networks but for stable environments the performance advantages of QUIC were not so obvious. They showed that QUIC connections are much less sensitive to delay and loss than HTTP/2 connections.

The IETF working group states multiple goals for IETF QUIC [12]. They want to minimize latency costs for applications starting with HTTP/2. QUIC should allow multiplexing without the head-of-line blocking problem. They want to also achieve that QUIC can be deployed by only making changes to path endpoints. Extensions for multipath and forward error correction should be enabled. Contrary to Google QUIC which uses it's own crypto IETF QUIC has the goal to provide always secure transport by using the TLS 1.3 standard. One challenge they pointed out was that about 3 - 5 % of networks block UDP [22]. They proposed that QUIC applications should therefore be implemented with a fallback option to TCP + TLS 1.3. Lower versions of TLS should not be allowed to prevent downgrade attacks.

[2] investigated the impact of IPv6 on connection establishment time of TCP on a global scale. They found that most websites had little difference in TCP connection times between IPv4 and IPv6. The once that showed a difference correlated with the

Content Delivery Network (CDN) not supporting IPv6.

3 Methodology

3.1 Design and Implementation

3.1.1 tls_perf

To compare the latency of QUIC with TCP we need to look at TCP with TLS since QUIC is an encrypted protocol. Therefore we wrote a program that evaluates TLS 1.2 and the new TLS 1.3. Originally we planned to write an extension for the program `happy` [1] used here [3] which already evaluates TCP connections. The library we chose to use for the TLS connections was libcurl since it already supported TLS version 1.3. However libcurl could not be integrated in the existing code base easily because it is a black box like library that gets initially configured by the user and does the connection internally. Therefore we wrote a new program called `tls_perf` [19].

The program connects to a given webserver and outputs data about the connection in the following format:

DNS lookup time;Current time;Hostname;Path;IP address;Port;TCP+TLS handshake time in milliseconds;HTTP Response Code;Protocol version;TCP handshake time

The ;Path; part is only here to align the output with the output of the `quic_perf` program introduced later in this chapter. Libcurl does not yet support URL parsing therefore the entire URL is saved in the ;Hostname; part and the ;Path; part is empty.

The libcurl library works that you configure your connection using the `curl_easy_setopt(...);` function with different parameters. Than you start your connection using `curl_easy_perform(...);`.

You can see this in this snippet from tls_perf.

```
curl_global_init(CURL_GLOBAL_DEFAULT);
CURL *curl;
curl = curl_easy_init();
CURLcode res;
if(bool4 == 1)
       curl_easy_setopt(curl, CURLOPT_IPRESOLVE, CURL_IPRESOLVE_V4);
else if (bool6 == 1)
```

```
        curl_easy_setopt(curl, CURLOPT_IPRESOLVE, CURL_IPRESOLVE_V6);
else
        curl_easy_setopt(curl, CURLOPT_IPRESOLVE, CURL_IPRESOLVE_WHATEVER);
curl_easy_setopt(curl, CURLOPT_URL, url);
curl_easy_setopt(curl, CURLOPT_DEFAULT_PROTOCOL, "https");
curl_easy_setopt(curl, CURLOPT_PORT, port);
curl_easy_setopt(curl, CURLOPT_TIMEOUT, 30L);
if(bool3 == 1)
        curl_easy_setopt(curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_3
| CURL_SSLVERSION_MAX_TLSv1_3);
else
        curl_easy_setopt(curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1_2
| CURL_SSLVERSION_MAX_TLSv1_2);
curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
res = curl_easy_perform(curl);
```

First we initialise the library globally. After that we initialise the individual curl handle curl. A program can have multiple such handles for multiple connections but we only use one here. After that we configure the handle/connection with the `curl_easy_setopt()` function setting url, port, timeout and so on. The option `CURLOPT_WRITEFUNCTION` expects a function that will handle the server response. In our case the function `write_data` simply returns discarding the server response as it is not relevant for us. Finally `curl_easy_perform(curl)` executes the connection as configured. The library logs meta data about the connection which you can access after the connection finished using the `curl_easy_getinfo(...)` function. For example we used `curl_easy_getinfo(curl, CURLINFO_NAMELOOKUP_TIME, &connect_dns);` to retrieve the DNS lookup time. Libcurl logs these values amongst other things [8]:

```
curl_easy_perform()
    |
    |--NAMELOOKUP
    |--|--CONNECT
    |--|--|--APPCONNECT
    |--|--|--|--PRETRANSFER
    |--|--|--|--|--STARTTRANSFER
    |--|--|--|--|--|--TOTAL
    |--|--|--|--|--|--REDIRECT
```

`NAMELOOKUP` is the time until DNS lookup is completed, `CONNECT` is the time until TCP handshake is completed and `APPCONNECT` is the time until TLS handshake is completed. `tls_perf` uses these 3 values as output but subtracts `NAMELOOKUP` from `CONNECT` and

`APPCONNECT` to get the correct TCP and TCP + TLS handshake time.

At the end of the program you have to call the functions:

```
curl_easy_cleanup(curl);
curl_global_cleanup();
```

They clean up and terminate your handles and the library.

To compile `tls_perf` the a libcurl version and a cryptolibrary that both support TLS 1.3 has to be installed on the system. In our tests we used libcurl 7.61.0 and OpenSSL 1.1.1 because they provided the required functionality. For more details see chapter 5 and 6.

There also exists a Windows port for `tls_perf` in a separate directory on the git. However it is not tested very well and was only used for some initial tests. All measurements in this thesis were done using the Linux version. The main difference in the Windows version is that the function `getopt()` which is part of unistd.h is not available in windows. We therefore included a windows port of `getopt()` made by Ludvik Jerabek [20] in the project.

`tls_perf` can also be used to determine whether a website supports TLS 1.3. When given the options -3 (for TLS 1.3) and -x the program will try to connect to the given website and print it's URL in stdout if it was possible to connect to it using TLS 1.3. Since some websites support TLS 1.3 when accessed with www.example.com but not when accessed with example.com the program will additionally try to connect to "www." + URL if the first connection attempt failed. An explanation for this could be that the Contend Delivery Network (CDN) was accessed when connecting to www.example.com which could already support TLS 1.3 but the real server accessed with example.com does not.

Because we used `tls_perf` to test a large number of websites the program has a connection time-out of 30 seconds so tests would run in a timely fashion.

3.1.2 quic_perf

One goal of this thesis was to create a program that measures Google QUIC performance. To accomplish that we needed a QUIC library. LiteSpeed QUIC Client Library (LSQUIC) was the library that we chose because it was written in C, had Linux and Windows support, supported the most recent Google QUIC versions and had active developers. Other QUIC libraries also exist see Future Work 5.

We first tried building LSQUIC on Windows. Doing so we ran into multiple problems such as the cmake script not being able to determine the right compiler name, deprecated functions in the windows port [14] and a runtime error [15]. After reporting these bugs to the developers they fixed all of them respectively within days. After

that we also build LSQUIC on an Ubuntu 16.04 system without running into major problems.

The LSQUIC library comes with a test client that can establish a QUIC connection to a given IP and prints the server response. Since this client already provided some functionality that we needed we decided to fork LSQUIC and write an extension for the test client instead of creating a new client.

The LSQUIC library provides some callbacks. The user can define the functions that handle them himself by defining an `struct lsquic_stream_if` as is demonstrated in the test client:

```
const struct lsquic_stream_if http_client_if = {
    .on_new_conn = http_client_on_new_conn,
    .on_conn_closed = http_client_on_conn_closed,
    .on_new_stream = http_client_on_new_stream,
    .on_read = http_client_on_read,
    .on_write = http_client_on_write,
    .on_close = http_client_on_close,
    .on_hsk_done = http_client_on_hsk_done,
};
```

The left part defines the callback while the right part of the equation is the function handling it. This struct is than passed to the library alongside other options when initializing it:

```
struct http_client_ctx client_ctx;
struct sport_head sports;
struct prog prog;
...
prog_init(&prog, LSENG_HTTP, &sports, &http_client_if, &client_ctx);
```

When doing a test connection with the test client using an example they provided we could not get a response from the server. This problem was due to the server (www.google.com) having different IPs in different regions. The LSQUIC client did not support DNS resolution at this point. You will not be able to establish a connection if you try to connect to an IP-address from another region. After finding out the correct IP-address using a browser add on (DNSLytics) we were able to successfully connect the client to www.google.com using QUIC.

To prevent this problem in the future we decided to implement DNS resolution using the function `getaddrinfo()` in the test client. This feature would benefit most users of the LSQUIC client. Therefore we made a pull request which got merged into LSQUIC after some feedback and revisions. [16]

After this we implemented time measurements and renamed the program to `quic_perf` to accurately describe it's function. Since `quic_perf` is an extension of the original `http_client` the option -t has to be used to output measurements. Otherwise the original behaviour of printing the server response to stdout is preserved. The handshake measurement starts when the connection is created:

```
if(time_option == 1)
       timespec_get(&ts_start, TIME_UTC);
if (NULL == lsquic_engine_connect(...))
       return -1;
```

The handshake measurement than ends when the library calls the `on_hsk_done` callback:

```
static void http_client_on_hsk_done (lsquic_conn_t *conn, int ok)
{
    if(time_option == 1)
    {
        timespec_get(&ts_end, TIME_UTC);
        timespec_diff(&ts_start,&ts_end, &ts_result);
        number_filled += snprintf(output + number_filled, 500 - number_filled,
"%.3lf;", (ts_result.tv_nsec/(double) 1000000));
    }
    LSQ_INFO("handshake %s", ok ? "completed successfully" : "failed");
}
```

The difference between the start and end time is than stored in the output buffer in millisecond format. The output is only printed to stdout right before the program successfully terminates to avoid printing incomplete information. An example of this would be when the DNS lookup succeeded but the QUIC handshake failed. The output format of `quic_perf` is similar to the one of `tls_perf`:

DNS lookup time;Current time;Hostname;Path;IP address;Port;QUIC handshake time in milliseconds;HTTP Response Code;Protocol version;

Unlike in libcurl were DNS lookup is done by the library we did and measured the DNS lookup in the client using `getaddrinfo()` for lookup and `timespec_get()` function for measurement due to it's high accuracy and portability.
QUIC is a protocol that is based on UDP. Some firewalls block UDP messages. To help avoid security risk connected to opening UDP ports we implemented an option (-c PORT) to set the port used on your local machine. That way only one port needs to be opened in the firewall to establish a successful QUIC connection with `quic_perf`: This

is done by setting the property `sin_port` of the `sockaddr_in struct`:

```
struct sockaddr_in sin;
sin.sin_port = htons(local_port);
```

The function `htons()` is necessary to convert the port to network byte order.

3.2 Target Selection

Since TLS 1.3 is a rather new protocol we first needed a list of websites that already supported the protocol. We used `tls_perf` in -x mode as described in 3.1.1 to determine the websites in Alexa Top 1000 and Alexa Top 100.000 that supported TLS 1.3 (we could not test the full Alexa Top 1 Million due to time constraints). A website is counted as supporting TLS 1.3 if ether their example.com or their www.example.com URL can be accessed using only TLS 1.3 from our vantage point in Munich Germany. For limitations of this test see Chapter5. We did our measurement of Alexa Top 1000 on 27.06.2018. At this day at least 107 websites do already support TLS 1.3 meaning 10,7%. The measurement of TLS 1.3 support in Alexa Top 100.000 was done in the time-frame of 19.07.2018 to 25.07.2018. Out of those websites at least 15918 or 15,918% support TLS 1.3. These two lists (used in different measurements respectively) were the websites we used to compare TLS 1.3 against TLS 1.2.

For the QUIC vs TLS 1.2/1.3 comparison we used the website list provided at [29] which was measured on 29.06.2018 see [30] for more details. This list contains all websites out of the Alexa Top 1 Million that support QUIC and have a valid certificate, 267 total. All website lists we used are available at [36].

3.3 Measurement Setup

To have a stable test environment where we can run test for a longer period of time we set up two virtual machines(VM) at the Technical University of Munich with the help of the administrators of the chair for connected mobility. The first VM emilia run on Ubuntu 16.04.2 LTS having a x86_64 QEMU Virtual CPU processor with 4 cores, 8 GiB of RAM and 21 GB disk space. Emilia had a download speed of roughly 300 Mbit/s when measured. This VM has libcurl 7.61.0-Dev and OpenSSL v1.1.1-PreRelease7 installed. The second VM vmott17 runs on Ubuntu 16.04.5 LTS having a x86_64 Intel Xeon processor with 2 cores 2 GiB of RAM and 85GB of RAM and 85 GB of disk space. Vmott17 has libcurl 7.61.0 and OpenSSL v1.1.1-PreRelease8 with a Download speed of roughly 900 Mbit/s when measured. Both VM's have a cable Internet-connection (to avoid additional latencies that a wireless connection might introduce) with the emilia

VM only having IPv4 while the vmott17 VM has an IPv6 connection as well. We set up both programs as described in 6. A focus was to built all the programs and libraries involved with maximum optimisations since not doing so would significantly influence the result, see 5.

3.4 Dataset

Using the VM's described in the previous section we collected a number of datasets. All data we collected is available at [13]. The data is stored in .csv files (comma separated values) with
DnsLookupTime;TimeOfMeasurement;Url;Path;p;Port;ConnectionEstablishmentTime; HttpResponse;Protocol;TcpHandshakeTimeOptional as header that describes the format of the data. All files are named using this pattern:
VM_IPProtocol_TargetSet_Notes.csv
The target set QUIC refers to the 267 websites supporting QUIC, 1kTls13 are the 107 TLS 1.3 websites and 100kTLS13 are the 15918 TLS 1.3 websites, see 3.2. The set 3websites contains the websites www.google.com, www.youtube.com and www.litespeedtech.com.

The table 3.1 gives further information about the datasets not included in the filename.

Table 3.1: Description of Datasets

filename	number of samples	measured between
emilia_v4_1kTls13.csv	16630	07. and 10.07.2018
emilia_v4_3websites_brokendns.csv	355	15. and 17.06.2018
emilia_v4_100kTls13.csv	157292	2. and 05.08.2018
emilia_v4_QUIC_one_measurement.csv	626	01.07.2018
emilia_v4_QUIC_quic_perf_first.csv	41895	16. and 19.07.2018
emilia_v4_QUIC_tls_perf_first.csv	63776	01. and 07.07.2018
vmott17_v4_100kTls13.csv	62919	04. and 05.08.2018
vmott17_v4_QUIC.csv	9930	05. and 06.08.2018
vmott17_v6_100kTls13.csv	108893	01. and 03.08.2018
vmott17_v6_QUIC.csv	16451	30.07.2018 and 01.08.2018

The measurement *emilia_v4_3websites_brokendns.csv* still used the underperforming libcurl DNS resolution see 4.1.
In the measurement *emilia_v4_QUIC_quic_perf_first.csv* the `quic_perf` test was executed before the `tls_perf` test to see the difference this makes on DNS resolution.
Vice versa in *emilia_v4_QUIC_tls_perf_first.csv*

4 Performance Evaluation

4.1 DNS Lookup Times

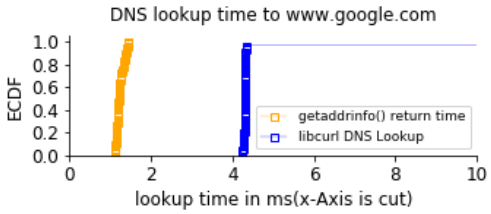


Figure 4.1: DNS lookup times to www.google.com

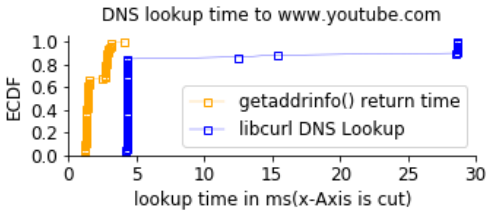


Figure 4.2: DNS lookup times to www.youtube.com

One thing we noticed when initially doing tests was that the DNS lookup times from `tls_perf` that used libcurl were consistently higher than the times from `quic_perf` that manually looked up the DNS using getaddrinfo()(~4ms compared to ~1ms). This was consistent regardless of the order in which the programs where called and therefore was probably not caused by caching.

These two CDFs which were drawn using the *emilia_v4_3websites_brokendns.csv* dataset show Dns lookup times to Google 4.1 and Youtube 4.2 using libcurl in `tls_perf` or getadressinfo() in `quic_perf`.

Since we could not find a solution to this problem we got in contact with the developers of libcurl over the curl mailing list. It turned out that libcurl does have 3 different ways of DNS resolution: threaded, synchronous and c-ares. The problem was that the default threaded resolver was too slow in the first few time-outs. Switching to the synchronous resolver solved our problem and we got the same DNS lookup times in both programs. Daniel Stenberg also made fix for the threaded resolver of libcurl see [33].

To see if there was still any difference after we compiled libcurl using the threaded resolver we made two separate measurements one with `tls_perf` being called first and with `quic_perf` being executed afterwards and vice versa. The one that was called first had a significant increase in lookup time both times as seen in plots 4.3 and 4.4. So we concluded that both programs now had a similar performance for DNS lookups. We

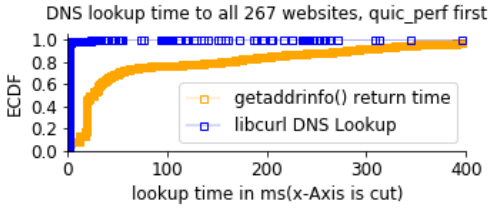


Figure 4.3: DNS Lookup times with the tls_perf being called first

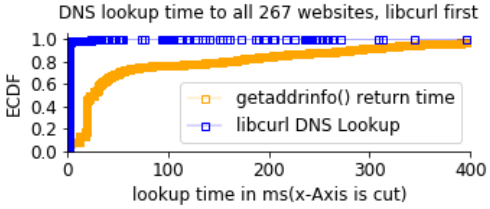


Figure 4.4: DNS Lookup times with the quic_perf program being called first

think the difference shows the performance increase gained from DNS caching for our machine. Figure 4.3 plots the DNS data from *emilia_v4_QUIC_tls_perf_first.csv* and 4.4 plots the data from the set *emilia_v4_QUIC_quic_perf_first.csv*

4.2 TLS 1.2 vs Tls 1.3

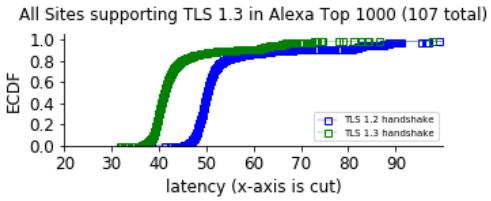


Figure 4.5: Latency of 107 sites in Alexa Top 1000 that support TLS 1.3

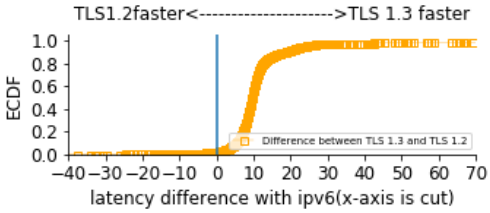


Figure 4.6: Difference between TCP/TLS1.3 and TCP/TLS1.2 in Alexa Top 1000

Figure 4.5 shows the performance of both TLS 1.2 and TLS 1.3 from the measurement set *emilia_v4_1kTls13.csv*. The QUIC data is not plotted here as there were too little websites in the sample that supported it.

To visualize the difference between the two protocols we paired measurements from the set that happened in the same minute against the same URL and plotted the difference in connection establishment time in Figure 4.6.

TLS 1.3 was consistently outperforming TLS 1.2 gaining an advantage of ~10 ms ~90% of the time. One thing to keep in mind here is that all these websites have presumably very good localisation as they are in the Alexa Top 1000 websites.

Figures 4.7 and 4.8 plotted from the set *emilia_v4_100kTls13.csv* show that there is little to no difference between Alexa Top 1000 and Alexa Top 100.000 websites. Again

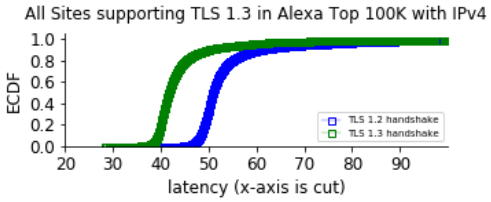


Figure 4.7: Latency of 15981 sites in Alexa Top 100.000 that support TLS 1.3 measured with emilia

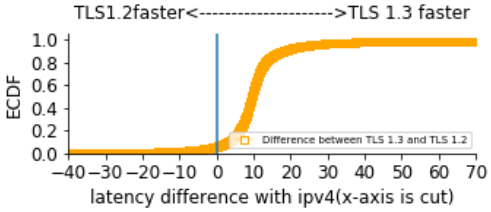


Figure 4.8: Difference between TCP/TLS1.3 and TCP/TLS1.2 in Alexa Top 100.000 measured with emilia

the performance advantage of TLS 1.3 is ~10 ms ~90% of the time.

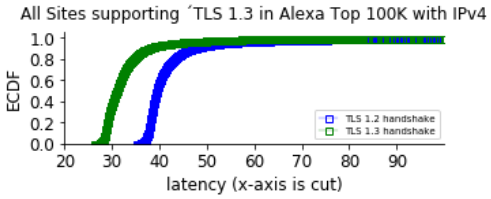


Figure 4.9: Latency of 15981 sites in Alexa Top 100.000 that support TLS 1.3 measured with vmott17

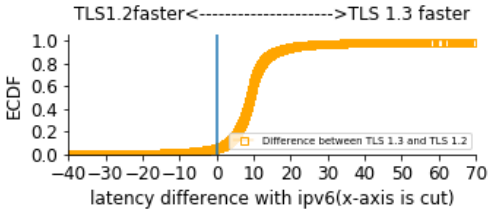


Figure 4.10: Difference between TCP/TLS1.3 and TCP/TLS1.2 in Alexa Top 100.000 measured with vmott17

As shown in figure 4.10 the same relative difference between the two protocols remained when we plotted the dataset *vmott17_v4_100kTls13* which was measured on the VM vmott17. However the absolute performance of both protocols was ~10 ms faster than when measured on the other VM see figure 4.9.

Finding out where this difference comes was not investigated and left for future work. Possible reasons could be the faster Internet connection (unlikely because of the high throughput see [34]), the slightly newer libraries or the different hardware components of the VM. For more details see 3.3.

Lastly we also looked at the impact of IPv6.

The figures 4.11 and 4.12 which were plotted from the set *vmott17_v6_100kTls13* show that IPv6 has no impact on the performance of TLS 1.3 or TLS 1.2.

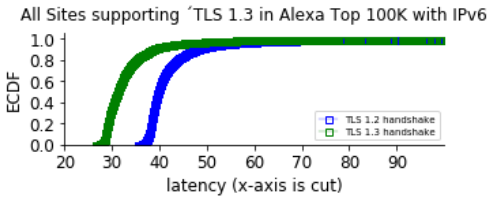


Figure 4.11: Latency of sites in Alexa Top 100.000 that support TLS 1.3 and IPv6 measured with vmott17

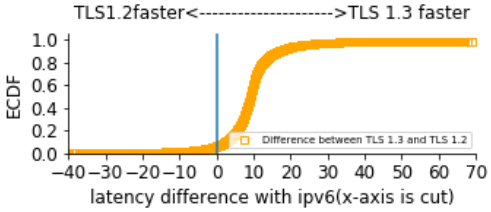


Figure 4.12: Difference between TCP/TLS1.3 and TCP/TLS1.2 over IPv6 in Alexa Top 100.000 measured with vmott17

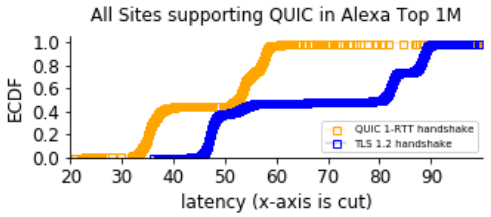


Figure 4.13: Latency of QUIC and TLS 1.2 from 267 websites measured over several days on emilia

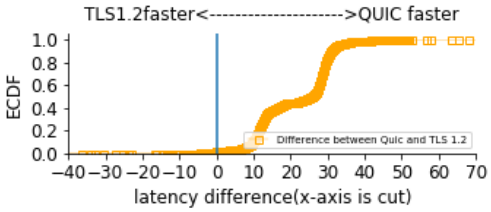


Figure 4.14: Difference between QUIC and TCP/TLS1.2 on emilia

4.3 QUIC vs TLS 1.2

Plotting the dataset *emilia_v4_QUIC_quic_perf_first.csv* results in figure 4.13 which shows Google QUIC consistently outperforming TLS 1.2. You can see 2 different sections in the graph probably due to different locations of the severs.

Just like in section 4.2 we made plots that show the difference in performance. Figure 4.14 shows that QUIC is around 10 ms faster than TLS 1.2 ~40% of the time and around 30 ms faster ~60% of the time.

When plotting the set *vmott17_v4_QUIC.csv* TLS 1.2 has a higher performance again compared to emilia measurements, see figure 4.15. QUIC on the other hand did not have a significant performance difference between the two test environments. Therefore as shown in 4.16 the performance advantage QUIC has shrinks to ~5 ms ~40% of the time and ~20 ms ~60% of the time.

Figures 4.17 and 4.18 plot data from *vmott17_v6_QUIC.csv*. This time IPv6 shows a difference in performance compared to IPv4. Instead of 60% of the time QUIC gains a ~20 ms advantage ~90 % of the time. This however is likely not due to an influence of

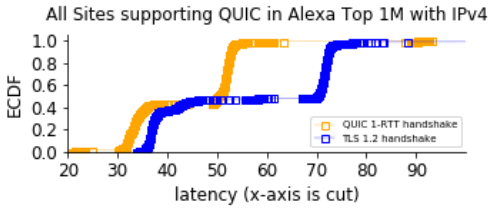


Figure 4.15: Latency of QUIC and TCP/TLS1.2 on vmott17 with IPv4

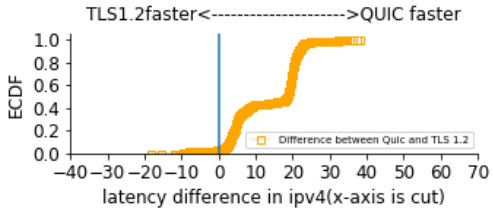


Figure 4.16: Difference between QUIC and TCP/TLS1.2 on vmott17 with IPv4

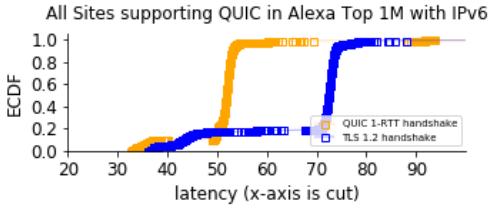


Figure 4.17: Latency of QUIC and TCP/TLS1.2 on vmott17 with IPv6

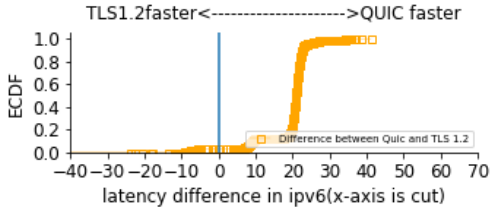


Figure 4.18: Difference between QUIC and TCP/TLS1.2 on vmott17 with IPv6

the protocol but rather because of the different numbers of sites being measured. In the IPv4 set there are 267 different websites while in IPv6 there are only 169. With the TLS 1.3 set there were also less websites that supported IPv6 than IPv4 but there we had a much higher number of total websites (13663 and 15805 respectively) which diminishes the influence of individual websites on the outcome.

4.4 QUIC vs TLS 1.3

Comparing QUIC with TLS 1.3 proved to be difficult as there are only 3 websites in Alexa Top 1 Million that support both protocols namely www.google.com, www.googlevideo.com and www.newmoney.gr. The following plots therefore contain little amount of data.

Figures 4.19 and 4.20 plot the set *emilia_v4_QUIC_tls_perf_first.csv* and show the performance gain QUIC had when measured on emilia. Here QUIC has an performance advantage of ~10 ms ~90 % of the time.

Again as figures 4.21 and 4.22 which plot *vmott17_v4_QUIC.csv* show TLS 1.3 (and 1.2) had a performance increase on vmott17 decreasing the advantage QUIC has over TLS 1.3 to ~2 ms 99% of the time.

IPv6 also has no significant impact on the connection times of QUIC compared to

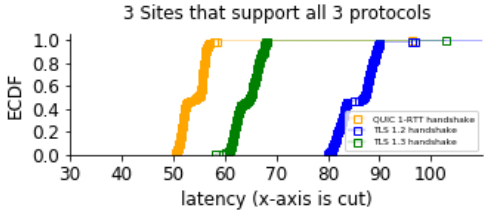


Figure 4.19: Latency of all protocols on 3 different sites collected on emilia

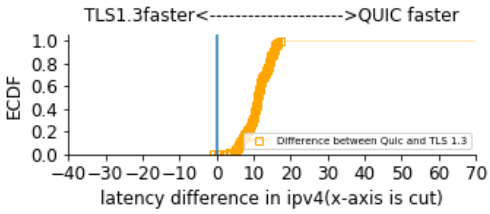


Figure 4.20: Difference between QUIC and TCP/TLS1.3 on emilia

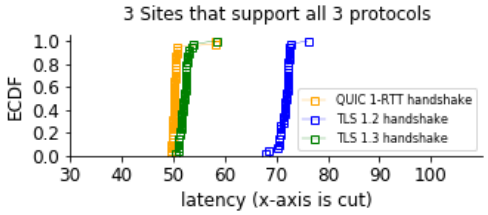


Figure 4.21: Latency of all protocols on 3 different sites collected on vmott17

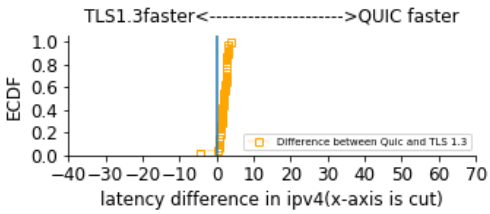


Figure 4.22: Difference between QUIC and TCP/TLS1.3 on vmott17

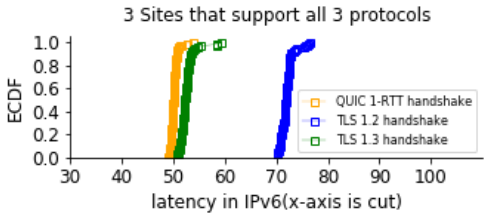


Figure 4.23: Latency of all protocols on 3 different sites collected on vmott17

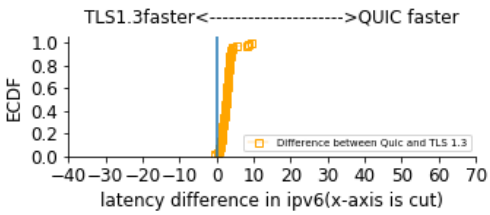


Figure 4.24: Difference between QUIC and TCP/TLS1.3 on vmott17

TLS 1.3 as figures 4.23 and 4.24 who plot the set *vmott17_v6_QUIC.csv* show.

5 Conclusion and Discussion

Conclusion
This work studied the adoption of TLS 1.3 and evaluated the latency of Google QUIC, TLS 1.2 and TLS 1.3 as well the impact of IPv6 on those protocols. The following research questions were answered:

RQ 1: How many websites adopted TLS 1.3 already?

Out of the Alexa Top 1000 list at least 10,7% of the websites support TLS 1.3 (on 27.06.2018).
Out of the Alexa Top 100.000 at least 15,9% of the websites support TLS 1.3 (between 19.07.2018 and 25.07.2018).

RQ 2: How much faster/slower does TCP/TLS 1.3 connect compared to TCP/TLS 1.2?

TLS 1.3 connects approximately ~10ms faster than TLS 1.2 ~80% of the time in both test environments (see 3.2) while outperforming TLS 1.2 in almost 99% of the cases. TLS connection establishment times between IPv6 and IPv4 were comparable.

RQ 3: How much faster/slower does QUIC connect compared to TCP/TLS 1.2?

Depending on the test environment: QUIC had a performance advantage of ~30 ms over TLS 1.2 ~60% of the time and an advantage of ~10 ms ~40% of the time in one of our measurements. On the second test environment QUIC had a performance advantage of ~5 ms ~40% of the time and ~20 ms ~60% of the time. Also on the second test environment QUIC gains a ~20 ms advantage ~90 % of the time with IPv6. This change in performance might be a result of fewer websites being measured. QUIC also outperformed TLS 1.2 in almost 99% of the tests.

RQ 4: How much faster/slower does QUIC connect compared to TCP/TLS 1.3?

QUIC has a performance advantage of ~10 ms over TLS 1.3 ~90% of the time in the first test environment. On the second test environment QUIC has an advantage of ~2 ms ~99% of the time. QUIC outperforms TLS 1.3 in ~99% of the test results. IPv6 again did not show any significant change in the result compared to IPv4.

Limitations

The results in this thesis have several limitations. To start with all measurements where done from two vantage points in Munich Germany. The data analysed therefore does not have geographical diversity.
The second limitation resulting from the first is that websites were counted as not supporting TLS 1.3 if it was not possible to connect to the website within a 30 second time-out frame. Therefore websites which were very slow to connect to or were unreachable from our vantage point are not counted as supporting TLS 1.3. Lastly the amount of different websites within Alexa Top 1 Million that support both TLS 1.3 and Google QUIC is only 3. Therefore the sample size and diversity of the data comparing TLS 1.3 and Google QUIC is very low.

Lessons Learned

An important lesson we learned was that local computations can have a large impact on the performance of connection protocols. For example when we compared protocol performance with programs run in Debug mode or build without optimisations we saw an significant (~30%) drop in performance compared to Release mode / compiler optimisations turned to max. When trying to set up a Virtual Machine that had an IPv6 connection we ran into a similar problem in a different incarnation. When testing the performance of TLS 1.2 on different VM's the connection speed was largely different on different VM's. The command we used to test this was:
`curl -o /dev/null -s -w '%{time_appconnect}\n' https://www.youtube.com`
Some VM's had the "correct" performance of 50 ms connection time while others had three times as much (150 ms). The problem turned out to be related to crypto libraries. The VM's that performed worse had GnuTLS/3.4.10 installed while the VM's that had a good performance where running on different versions of OpenSSL. The third case was like described in chapter 4 our two different test environments showed different performance for the TLS protocols. Our takeaway from this is that test environments can have a considerable impact on the outcome of measurements and should therefore be optimized as much as possible, documented precisely and ideally test should be run from different test environments.

During doing the set up of the emilia VM we had some problems establishing a encrypted TLS connection. Thanks to the help of Steffen Ullrich [35] we found out that the problem was a proxy and we had to do HTTP tunneling in order to establish a secured connection through the proxy. After we successfully established a TLS connection we noticed that the TCP handshake times measured by out program where way to low (~500 microseconds). The cause for this was that the TCP handshake that was

measured was the one done with the proxy server. We visualized this in figure 5.1

client<--TCP-->Proxy<---TCP---->Server/ISP middle-box
client<----------TLS---------->Server

Figure 5.1: The impact of Proxys on TCP measurements

Only the first TCP part in figure 5.1 is measured so the measurement was missing the second part. To avoid this we removed the proxy server as it was not necessary. Our takeaway from this is that you should omit adding proxies on your connection when evaluating TCP/TLS based protocols. The measured TCP times might still be the once to a proxy as Internet Service Providers (ISP) use middle-boxes to split TCP connections.

After setting up all the programs correctly on the emilia VM we unfortunately could not establish a QUIC connection with any website at first. We talked about this problems with the administrators and it turned out to be the firewall who was blocking all UDP connections. This firewall only allowed incoming packets when there was an outgoing connections first. However UDP is connectionless so no incoming connection was allowed. Some firewalls manually track UDP connections to avoid this problem but this one was more strict. To solve this problem the VM was moved to another network with a different firewall that allowed UDP connections. Firewalls blocking UDP connections could be a challenge for the adoption of IETF QUIC going forward.

Libcurl the library that `tls_perf` is based on has a mailing list [24]. Since we were on that list because of the DNS lookup time problem mentioned in chapter 4 we noticed a change that was made to the library option that set the TLS version (`CURLOPT_SSLVERSION`). Instead of the exact TLS version the option set the minimum TLS version beginning with the next release. This could have silently falsified the output of `tls_perf` if we wouldn't have noticed and changed the code accordingly. Our takeaway from this is that it is useful to keep up with the current development of libraries you are using.

Future Work
Future research can do measurements from a multitude of vantage points to assess the different protocols form a more global and diverse perspective. In this work we measured 1-RTT QUIC. Future work can be done analysing the effect of the 0-RTT feature in QUIC and TLS 1.3. We also only looked at QUIC that is deployed in the Internet right now. Future work can analyse the performance of IETF QUIC once

the standard is finalized. Another opportunity for future work is the analysis of the penetration of TLS 1.3 in the entire Alexa Top 1 Million. This work looked at individual libraries that support TLS 1.3 and Google QUIC. There are also other libraries that support these features. Doing an exhaustive survey of all libraries that support TLS 1.3 and QUIC is left for future work. Analysing the impact of Content Delivery Networks on the protocols is another opportunity for future work.

Acknowledgements
We would like to thank Thomas Paul and Simon Zelenski for their help with setting up the test environments. Furthermore we would like to thank Dmitri Tikhonov, Daniel Stenberg, Sergey Podanev and Steffen Ulrich for their help on technical questions.

6 Reproducibility Considerations

6.1 Installation of the programs

The first part will be a tutorial on how to set up the different libraries and programs we used to do our tests. All measurements were done using a cable connection to avoid delays coming from wireless connections. For the analysis we used *Python 3.6* with Anaconda Jupyter Notebook. The notebook files are available at [17]. Some things such as the hardcoded paths might have to be adjusted to draw the individual plots. Table 6.1 shows which figure was drawn using which notebook.

6.1.1 LS-QUIC client

(Part of this information comes from the LSQUIC readme) This tutorial is made for Ubuntu 16.04 LTS (code is compiled with gcc version 5.4.0) The first step is to download the program from the git repository:

```
cd $HOME
git clone https://github.com/Kait0/lsquic-client.git
```

Change to the directory:

```
cd lsquic-client
git checkout master (if you are not already on it)
```

To build LSQUIC you need CMake, zlib, libevent and BoringSSL (Google QUIC uses the BoringSSL crypto):

```
sudo apt-get update
sudo apt-get install cmake
sudo apt-get install zlib1g-dev
sudo apt-get install libevent-dev
```

Installing BoringSSL is a bit more complicated as you have to build it yourself. To build BoringSSL you will need to install the Go language.

```
sudo apt-get install golang
cd .. (if you are still in the lsquic-client folder)
```

```
git clone https://boringssl.googlesource.com/boringssl
cd boringssl
git checkout chromium-stable
cmake -DCMAKE_BUILD_TYPE=Release . && make
(we want the maximum performance so we turn on optimisations)
BORINGSSL_SOURCE=$PWD
mkdir -p $HOME/tmp/boringssl-libs
cd $HOME/tmp/boringssl-libs
ln -s $BORINGSSL_SOURCE/ssl/libssl.a
ln -s $BORINGSSL_SOURCE/crypto/libcrypto.a
```

We will also need a crypto that supports TLS 1.3. In our measurements we used OpenSSL 1.1.1. You can download OpenSSL from their webpage:

```
cd $HOME
wget https://www.openssl.org/source/openssl-1.1.1-pre8.tar.gz
tar -xvzf openssl-1.1.1-pre8.tar.gz
cd openssl-1.1.1-pre8
./config --release
make
make test (optional)
sudo make install
```

Now that we have the library we can link it to lsquic and build the lsquic library (again we want maximum performance so we compile in release mode)

```
cd $HOME/lsquic-client
(The next two lines are one line.)
cmake -DBORINGSSL_INCLUDE=$BORINGSSL_SOURCE/include
-DBORINGSSL_LIB=$HOME/tmp/boringssl-libs -DDEVEL_MODE=0 .
make
```

You can test if the installation worked (program should print the server response) Be aware that QUIC builds upon UDP so your firewall needs to have that enabled in order to get a working connection:

```
./quic_perf -s www.google.com:443 -p /
```

6.1.2 tls_perf

To install `tls_perf` we need to install a version of curl that supports TLS 1.3 (and Openssl 1.1.1 which we installed earlier)

```
cd $HOME
wget https://curl.haxx.se/download/curl-7.61.0.tar.gz
tar -xvzf curl-7.61.0.tar.gz
cd curl-7.61.0
./configure --disable-threaded-resolver
(we want the non-threaded DNS resolver since it's slightly faster)
make
make test (optional)
sudo make install
sudo ldconfig (update libraries so that we get the new OpenSSL)
```

Now that we have all the libraries we can install `tls_perf`:

```
cd $HOME
git clone https://github.com/Kait0/tls_perf.git
cd tls_perf
make
```

You can test the installation with e.g. :

```
./tls_perf -u www.youtube.com -p 443
```

Both programs we used for measurements are now installed.

To collect data with them we used a simple bash script:

```
#!/bin/bash
while IFS='' read -r line || [[ -n "$line" ]]; do
        cd $HOME/lsquic-client
    ./quic_perf -t -s $line:443 -p / >> $HOME/output.csv
        cd $HOME/tls_perf
        ./tls_perf -u $line -p 443 >> $HOME/output.csv
    ./tls_perf -3 -u $line -p 443 >> $HOME/output.csv
done < "$1"
```

It takes a text file as an argument that should contain the list of websites you want to test. The websites need to be separated by a line break `\n`. Make sure there are no `\r` at the end of each line because the script won't work if that's the case. The script than calls every website with the 3 different protocols and redirects the stdout output into the file output.csv . The header we put in the output.csv file is:

```
DnsLookupTime;TimeOfMeasurement;Url;Path;Ip;Port;
ConnectionEstablishmentTime;HttpResponse;Protocol;TcpHandshakeTimeOptional
```

The lists of websites we tested against are available at [36].

To execute the script periodically we used the program cron:

```
crontab -e
(add this line to execute the script every hour)
0 * * * * $HOME/measureScript websites-29-06.txt
```

6.2 Plots

All CSV files needed to draw the plots are stored in [13] and described in 3.4

Table 6.1: Mapping of Notebooks and Figures

Path to Notebook	Figure
~/data/plot_QUIC.ipynb	4.1 4.2 4.3 4.4 4.13 4.14 4.15 4.16 4.17 4.18 4.19 4.20 4.21 4.22 4.23 4.24
~/data/plot_TLS13.ipynb	4.5 4.6 4.7 4.8 4.9 4.10 4.11 4.12

List of Figures

List of Tables

Bibliography

[1] V. Bajpai. *happy eyeballs*. 2018. URL: https://github.com/vbajpai/happy (visited on 08/01/2018).

[2] V. Bajpai and J. Schönwälder. "IPv4 versus IPv6 - who connects faster?" In: *Proceedings of the 14th IFIP Networking Conference, Networking 2015, Toulouse, France, 20-22 May, 2015*. Ed. by R. Kacimi and Z. Mammeri. IEEE, 2015, pp. 1–9. ISBN: 978-3-901882-68-5. DOI: 10.1109/IFIPNetworking.2015.7145323.

[3] V. Bajpai and J. Schönwälder. "Measuring the Effects of Happy Eyeballs." In: *Proceedings of the 2016 Applied Networking Research Workshop, ANRW 2016, Berlin, Germany, July 16, 2016*. Ed. by L. Eggert and C. Perkins. ACM, 2016, pp. 38–44. ISBN: 978-1-4503-4443-2. DOI: 10.1145/2959424.

[4] G. Carlucci, L. D. Cicco, and S. Mascolo. "HTTP over UDP: an experimental investigation of QUIC." In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*. Ed. by R. L. Wainwright, J. M. Corchado, A. Bechini, and J. Hong. ACM, 2015, pp. 609–614. ISBN: 978-1-4503-3196-8. DOI: 10.1145/2695664.2695706.

[5] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. *TCP Fast Open*. 2014. URL: https://datatracker.ietf.org/doc/rfc7413/ (visited on 08/10/2018).

[6] S. Cook, B. Mathieu, P. Truong, and I. Hamchaoui. "QUIC: Better for what and for whom?" In: *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*. IEEE, 2017, pp. 1–6. ISBN: 978-1-4673-8999-0. DOI: 10.1109/ICC.2017.7997281.

[7] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind. "Innovating Transport with QUIC: Design Approaches and Research Challenges." In: *IEEE Internet Computing* 21.2 (2017), pp. 72–76. DOI: 10.1109/MIC.2017.44.

[8] *curl_easy_getinfo*. URL: https://curl.haxx.se/libcurl/c/curl_easy_getinfo.html (visited on 08/01/2018).

[9] S. Deering and R. Hinden. *IPv6 Internet Standard*. 2017. URL: https://tools.ietf.org/html/rfc8200 (visited on 08/08/2018).

[10] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz. "Measuring HTTPS Adoption on the Web." In: *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. Ed. by E. Kirda and T. Ristenpart. USENIX Association, 2017, pp. 1323–1338.

[11] Google. *IPv6 Adoption Statistics*. URL: https://www.google.com/intl/en/ipv6/statistics.html (visited on 08/08/2018).

[12] *IETF QUIC*. 2017. URL: https://datatracker.ietf.org/wg/quic/about/ (visited on 08/08/2018).

[13] B. Jaeger. *CSV Datasets*. 2018. URL: https://github.com/Kait0/quic-bachelorthesis/tree/master/data/CSV%20Data (visited on 08/11/2018).

[14] B. Jaeger. *LSQUIC issue 10*. 2018. URL: https://github.com/litespeedtech/lsquic-client/issues/10 (visited on 08/02/2018).

[15] B. Jaeger. *LSQUIC issue 13*. 2018. URL: https://github.com/litespeedtech/lsquic-client/issues/13 (visited on 08/02/2018).

[16] B. Jaeger. *LSQUIC Pull Request 30/34*. https://github.com/litespeedtech/lsquic-client/pull/30 and https://github.com/litespeedtech/lsquic-client/pull/34. 2018. (Visited on 08/02/2018).

[17] B. Jaeger. *Python Files*. 2018. URL: https://github.com/Kait0/quic-bachelorthesis/tree/master/data (visited on 08/07/2018).

[18] B. Jaeger. *quic_perf*. 2018. URL: https://github.com/Kait0/lsquic-client (visited on 08/01/2018).

[19] B. Jaeger. *tls_perf*. 2018. URL: https://github.com/Kait0/tls_perf (visited on 08/01/2018).

[20] L. Jerabek. *getopt for windows*. 2012. URL: https://www.codeproject.com/KB/cpp/getopt4win/getopt_mb_uni_src.zip (visited on 08/02/2018).

[21] A. M. Kakhki, S. Jero, D. R. Choffnes, C. Nita-Rotaru, and A. Mislove. "Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols." In: *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*. Ed. by S. Uhlig and O. Maennel. ACM, 2017, pp. 290–303. ISBN: 978-1-4503-5118-8. DOI: 10.1145/3131365.3131368.

[22] M. Kuehlewind and B. Trammell. *IETF QUIC applicability*. 2018. URL: https://datatracker.ietf.org/doc/draft-ietf-quic-applicability/?include_text=1 (visited on 08/11/2018).

[23] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. R. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W. Chang, and Z. Shi. "The QUIC Transport Protocol: Design and Internet-Scale Deployment." In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*. ACM, 2017, pp. 183–196. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098842.

[24] *Libcurl Mailing List*. URL: https://cool.haxx.se/mailman/listinfo/curl-library (visited on 08/09/2018).

[25] P. Megyesi, Z. Kramer, and S. Molnár. "How quick is QUIC?" In: *2016 IEEE International Conference on Communications, ICC 2016, Kuala Lumpur, Malaysia, May 22-27, 2016*. IEEE, 2016, pp. 1–6. ISBN: 978-1-4799-6664-6. DOI: 10.1109/ICC.2016.7510788.

[26] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. M. Munafò, K. Papagiannaki, and P. Steenkiste. "The Cost of the "S" in HTTPS." In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*. Ed. by A. Seneviratne, C. Diot, J. Kurose, A. Chaintreau, and L. Rizzo. ACM, 2014, pp. 133–140. ISBN: 978-1-4503-3279-8. DOI: 10.1145/2674005.2674991.

[27] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446.

[28] J. Rüth and O. Hohlfeld. *QUIC Statistics*. URL: https://quic.comsys.rwth-aachen.de/stats.html (visited on 08/08/2018).

[29] J. Rüth and O. Hohlfeld. *Websites with QUIC support*. 2018. URL: https://quic.comsys.rwth-aachen.de/quic-website-data/lists/quic-grabber_alexa-A-www_truesupport_2018-06.tar.br (visited on 08/02/2018).

[30] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld. "A First Look at QUIC in the Wild." In: *Passive and Active Measurement - 19th International Conference, PAM 2018, Berlin, Germany, March 26-27, 2018, Proceedings*. 2018, pp. 255–268. DOI: 10.1007/978-3-319-76481-8_19.

[31] R. H. S. Deering. *IPv6 Draft Standard*. 1998. URL: https://tools.ietf.org/html/rfc2460 (visited on 08/08/2018).

[32] Q. Scheitle, O. Hohlfeld, J. Gamba, J. Jelten, T. Zimmermann, S. D. Strowes, and N. Vallina-Rodriguez. "A Long Way to the Top: Significance, Structure, and Stability of Internet Top Lists." In: *CoRR* abs/1805.11506 (2018). arXiv: 1805.11506.

[33] D. Stenberg. *Curl DNS lookup time fix*. 2018. URL: https://curl.haxx.se/mail/lib-2018-06/0117.html (visited on 08/05/2018).

[34] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei. "Community contribution award - Measuring and mitigating web performance bottlenecks in broadband access networks." In: *Proceedings of the 2013 Internet Measurement Conference, IMC 2013, Barcelona, Spain, October 23-25, 2013*. Ed. by K. Papagiannaki, P. K. Gummadi, and C. Partridge. ACM, 2013, pp. 213–226. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504741.

[35] S. Ulrich. *Http Tunneling*. 2018. URL: https://stackoverflow.com/questions/50840101/curl-35-error1408f10bssl-routinesssl3-get-recordwrong-version-number (visited on 08/07/2018).

[36] *Website Lists*. 2018. URL: https://github.com/Kait0/quic-bachelorthesis/tree/master/data/Website%20Lists (visited on 08/09/2018).