POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

QUIC performance monitoring: implementation of Spin Bit in Chromium

Supervisors
Prof. Riccardo Sisto
Prof. Guido Marchetto

Candidate
Massimo Di Natale

Company Tutor
Telecom Italia LAB
Dott. Mauro Cociglio

April 2023

Summary

The objective of this thesis work is to implement a specific feature of the QUIC protocol, called Spin Bit, in the Chromium open source browser. QUIC Spin Bit feature allows network operators to perform QUIC traffic monitoring techniques: an on-path network observer analyzes the value of this specific bit and deduces the Round Trip Time RTT of the communication and the presence of possible network impairments. In particular, the implementation in Chromium will allow operators to perform this type of monitoring on real user traffic and usual navigation.

The first phase of the work involved the study of Chromium as an open source browser, with the analysis of the several tools that it provides to the developers in order to carry out the work, from the versioning tools, to the compiling and testing ones.

In a second phase, Chromium code architecture was explored, with a particular attention to the network modules and the library that implements the QUIC protocol, called Quiche. It is a Google developed library, written in C++, that does not support the Spin Bit feature natively. The core of the library was modified in order to implement the Spin Bit algorithm; more specifically, a Spin Bit variable was added in the data structure representing the QUIC Short Header, and the logic of the algorithm was implemented in packet reception and packet delivery modules. The steps are the following: after a packet is received, some checks are performed, as expressed in the algorithm (IETF RFC 9000), and if passed, the Spin Bit value is read, modified according to the algorithm and saved in the newly introduced field. Later, when a packet is sent, the header is filled with the current value of the Spin Bit.

A testing phase followed the implementation phase. The objectives of the tests were the evaluation of the newly implemented Spin Bit feature, together with the evaluation of the performance monitoring technique based on in-path network observers. More specifically, firstly it was verified if the

tools were able to detect the Spin Bit feature and in a second moment the RTT values produced, compared to Ping reference values. The tools used as in-path network observers were Spindump and TIMQuic, for the Linux and Android platform respectively. These tools allow to perform passive performance monitoring: being in-path network observers they read the Spin Bit value of the QUIC packets in the communication between client and server and compute the end-to-end RTT as the sum of 'upstream' or Left RTT and 'downstream' or Right RTT.

The first test aimed to verify if the on-path network observer Spindump was able to detect the correct behavior of the Spin Bit feature in the modified library Google Quiche. The setup was composed of two lightweight programs of the Google Quiche library that simulated the behavior of a client and a server on the same machine. The quic-client performed network requests to the quic-server through the QUIC protocol, and the server was configured to respond with static content. In the meanwhile, Spindump was running on the same machine and recording every packet that was passing through the loopback network interface. After the client requests and the server responses, it recorded "Spinning" on the QUIC connection, meaning that it was able to detect the support of the Spin Bit on both endpoints of the communication.

The second test used the implementation of third-party libraries, different from Google Quiche, as a server in order to test the interoperability of the Spin Bit with other implementations. The client used was the same as the first test. The remaining part of the setup was the same as the first test: quic-client that performed requests and Spindump as network observer. Also in this test, Spindump was able to detect "Spinning" events.

The third test involved Chromium browser, with the modified Google Quiche library, acting as a client and the litespeedtech.com website acting as a server. This was the first 'real-case' test, since it consisted of two production-ready products. The test was conducted both on the Android and Linux platforms, using TIMQuic and Spindump respectively as in-path network observers. Results showed that both tools successfully recorded "Spinning" events.

The fourth test consisted in the deployment of an open-source web server that implements the Spin Bit. The objective was to test the web server on an AWS instance firstly to detect the 'Spinning' events and then to obtain network measurements from different places around the world. The web server used was OpenLiteSpeed, the open-source version of LiteSpeed server,

that implements as well the Lsquic library with support of the Spin Bit.

Then RTT values produced by Spindump while downloading files of 20 MB, 100 MB and 200 MB were analyzed and compared with Ping measurements.

The tests showed that, in general, Ping measurements were lower than the ones computed with Spin Bit. As mentioned in the IETF RFC 9312, the measurements based on Spin Bit must be treated as indicative values, since they reflect the latency perceived by the application layer, differently from Ping. On the other hand, the IETF RFC 9312 states that the minimum RTT computed on the basis of Spin Bit values should reflect the network latency. In all tests, the reported minimum RTT was in the same range of Ping values, as it was expected: the difference between Ping values and RTT ones is of maximum 2 milliseconds, due to the application layer delay.

To conclude, in this thesis work it was implemented a passive on-path measurement technique of QUIC traffic, based on the Spin Bit, in Chromium. Then it was set up a testing environment, firstly to detect the correct behavior of the Spin Bit algorithm on the same machine and then to analyze RTT measurements from different locations around the world, by using AWS EC2 instances in London and Frankfurt and comparing results with Ping measurements.

The measurements obtained by the on-path network observer represented the latency perceived by the application, with the exclusion of some outliers. For this reason the average values based on the Spin Bit algorithm were slightly higher than Ping measurements, as stated in the IETF RFC 9312.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Objectives

This thesis aims to provide a network performance monitoring technique implementation of QUIC traffic in the Chromium browser.

QUIC is a network transport protocol that was firstly introduced by Google and then standardized in 2021 by the IETF, as decribed in IETF RFC 9000 [1].

It is based on UDP and encrypts transport protocol information: for this reason, network operators cannot use their standard techniques of network traffic performance monitoring which are based on information that is no longer available.

In the QUIC header, an unencrypted bit, called Spin Bit, has been reserved, allowing the computation of the RTT measurements for QUIC connections for network traffic analyses.

QUIC is nowadays widely supported by browsers and servers that establish connections with the HTTP/3 protocol.

On the contrary, it is really difficult to find implementations of QUIC protocol that support the Spin Bit feature. As described in the IETF RFC 9000, Spin Bit is an optional feature, and QUIC protocol implementations rarely develop Spin Bit algorithm.

The objective of this thesis is to implement Spin Bit in an open-source browser in order to make it possible to monitor QUIC traffic performances and set up a testing environment that supports Spin Bit from both endpoints of communication: client and server.

1.2 Thesis structure

Chapter 2

The second chapter describes the QUIC protocol in terms of its features, advantages, and disadvantages in contrast to the other network transport protocols, TCP, and its relation with the HTTP/3 protocol.

Afterwards, I will introduce the Spin Bit feature as a technique to perform passive network traffic performance monitoring, explaining its behavior and the state-of-the-art adoption of browsers and servers.

Finally, I will present the software tools used to test Spin Bit behavior and produce RTT measurements: Spindump and TIMQuic, for Linux and Android platforms respectively.

Chapter 3

This chapter aims to explain the choice of Chromium as an open-source browser where to implement the Spin Bit feature.

I will then provide an overview of the tools I used to work in the Chromium environment in the different stages of development: versioning, implementation, compilation and testing.

Finally, I will describe the network architecture of Chromium, focusing on the Google Quiche library, the Google implementation of the QUIC protocol, written in C++, not to be confused with Quiche library developed by CloudFlare, mainly written in Rust.

Chapter 4

The fourth chapter describes in details the Spin Bit algorithm and the differences in client and server behaviors.

Afterwards, I will present the modules of the Google Quiche library that have been modified to implement the Spin Bit algorithm and the more relevant parts of the code of my implementation.

Chapter 5

This chapter contains the evaluation methods used to test the Spin Bit implementation for Linux and Android platforms.

The testing phase is divided in two main parts: the first one is focused on the detection of the "Spinning" event by the network observers placed in the same machine as client and server.

The second phase analyzes the measurements reported by Spindump listening on network communication between client and server placed in different AWS Regions. Results are then compared to Ping values, in order to evaluate the proposed performance monitoring technique with respect to Ping reference values, together with Spin Bit implementation.

Chapter 6

The final chapter focuses on the achievements and results of the thesis, that paves the way to further works based on more than one bit of the QUIC Header.

Chapter 2

Background

This chapter describes the state of the art network protocols used in current connections - HTTP/3 and QUIC - through the anlysis of their features and characteristics.

Firstly, I will present important steps in the evolution of HTTP and I will make a comparison between the different protocol versions.

After that, the HTTP/3 will be described, in order to show the improvements that it has achieved with respect to the previous implementations.

The second part of the chapter focuses on the QUIC protocol, explaining its main features and peculiarities. In particular, a specific bit of the protocol is described in detail: the Spin Bit. This bit allows the computation of passive network performance measurements by means of on-path network observers.

In the last section of the chapter, the tools used to perform QUIC traffic analysis for Android and Linux platforms are introduced, with special regards to the features that allowed me to carry out the evaluations of the Spin Bit implementation.

2.1 HTTP/3

HTTP/3 is the current version of the Hypertext Transfer Protocol (HTTP), which is the application-layer protocol designed for communication between web browsers and web servers.

History The first version of the protocol, released in the 90s, allowed to transfer multimedia content and hyper-textual documents over the Internet

and became the underlying protocol of the World Wide Web, gaining extreme popularity [2].

Although it was widely used, it presented many limitations: the greatest disadvantage was the fact that it was a *connectionless* protocol without support for pipelining. This had a major impact on performances, especially over slow connections, since connections had to be opened multiple times to display resources embedded in the single original document.

A few years later HTTP/1.1 was introduced. It deeply improved protocol performance thanks to the implementation of *persistent connections*, *pipelining* support, and *conditional requests*. Specifically, pipelining permitted requests to be sent before the answer to the previous ones was fully transmitted, obtaining lower latency in the communication and improving performance.

The introduction of the security layer between TCP and HTTP represented one of the most relevant changes in the network protocol stack. Initially called SSL, then standardized in TLS, it has satisfied the need of having encrypted packets between client and server to guarantee the authenticity of the messages.

The release of HTTP/2 furtherly improved the protocol, especially in terms of performance optimization, to overcome the limitations caused by web pages which started to become more and more complex and heavy.

The features of the protocol are the following:

- *Binary encoding*, which permitted to transmit data in a more efficient way with respect to the text-based encoding used in the previous versions.
- The *server push* mechanism, which allows the server to send resources before the client requests them, with a significant improvement in the loading speed of websites.
- *Multiplexing*, that allowed multiple requests to be sent over the same connection, resulting in faster page load time than in the past.
- *Header compression and prioritization*, which reduced the overhead of data transmitted.

Finally, HTTP/3 revolutionized the protocol, since it completely changed the protocol stack. In fact, it is based on UDP and QUIC at the transport layer, differently form the previous versions.

This way, it overcomes the limitations of TCP, in particular TCP loss detection and retransmission: in HTTP/2 they can block all streams since multiplexed connections are bound to a single TCP connection.

Moreover, HTTP/3 improves the HTTP/2 multiplexing thanks to a better flow control and congestion management, leading to faster and more reliable data transfers. This version offers great improvements as far as performance, efficiency and security are concerned.

More in detail, as described in IETF RFC 9114 [3], multiplexing of requests is performed using the QUIC stream abstraction: each request-response pair is bound to a single QUIC stream. Streams are independent from each other, so one stream that is blocked or suffers packet loss does not prevent progress on other streams.

To sum up the major improvements of HTTP/3 are:

- reduced latency thanks to QUIC and UDP as transport layers, that allow requests to be sent in parallel over multiple streams.

- security: data in transit is all encrypted, based on TLS1.3 and use of QUIC as transport layer.

- performance: HTTP/3 is designed to improve performance with the use of multiplexing and parallel streams in the QUIC transport layer.

- interoperability: designed to be backward compatible with previous versions of HTTP protocol.

The image shows how the protocol stack has changed in the different versions.

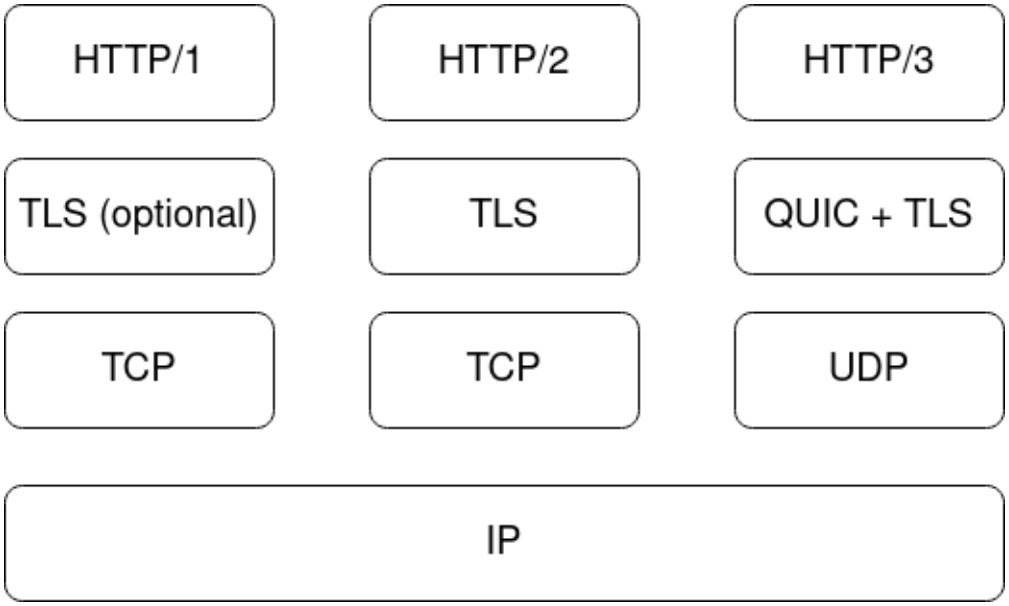


Figure 2.1: HTTP versions and network stacks

2.2 QUIC protocol

QUIC is a transport protocol that operates on top of UDP, and it is used as an alternative to TCP in the transport layer of HTTP/3. Its main objective is to provide faster setup times and reduced latency for HTTP connections.

Faster setup time is made possible thanks to the **QUIC handshake** mechanism. As mentioned in IETF RFC 9000 [1], QUIC handshake is structured to allow data exchange at the earliest: it combines negotiation of cryptographic and transport parameters, by integrating the TLS handshake, with an option that uses 0-RTT packets to send data immediately. 0-RTT is achieved through protocol parameters negotiated in previous connections. A client that attempts to send 0-RTT data has to know all transport parameters that the server is able to process and the server uses transport parameters to determine whether to accept 0-RTT data or not.

0-RTT allows application data to be sent by a client before receiving a response from the server. On the other hand, 0-RTT provides no protection against replay attacks. A server can also send application data to a client before it receives the final cryptographic handshake messages that allow it to confirm the identity and liveness of the client.

These capabilities allow an application protocol to offer the option of

trading some security guarantees for reduced latency.

Application protocols exchange information over a QUIC connection via **streams**, which are ordered sequence of bytes. Data is subject to flow control constraints and stream limits. The correct prioritization of resources allocated to stream leads to significant improvements in application performances, thanks to the *stream multiplexing*.

A QUIC **connection** is a shared state between client and server. Each connection is identified by a set of parameters that are independent from lower protocol layers (IP, UDP): this allows to perform seamless network path migration.

In order to improve user's Quality of Experience, QUIC allows independent retransmissions for sub-streams and decouples it from congestion control. This provides a great improvement: also in scenarios with poor network conditions it is possible to experience good website responsiveness.

As mentioned before, QUIC completely changes the network stack of the HTTP protocol: it is the first time that an HTTP version is based on UDP and embeds security in its own layer with TLS 1.3.

[4] states that the encryption of the majority of QUIC packets has a negative impact on network operators: it prevents them from performing passive network measurements using the standard techniques based on transport protocol information SEQ/ACK since it is no longer available.

For this reason, some bits of the QUIC Short Header packet were left unencrypted, allowing on-path network observers to record its values during the QUIC connection and estimate the Round Trip Time: among them, this thesis focuses on the Spin Bit.

QUIC connection establishment As reported in [5], to setup a typical QUIC connection the following messages are exchanged:

- Initial packet, sent from the client to the server, including a TLS 1.3 ClientHello.
- Initial packet, from the server to the client, if parameters of the previous message are accepted by the server, together with the TLS ServerHello.
- Handshake packet, from the server to the client, with the remaining TLS server messages.
- Handshake packet, from the client to the server, to terminate the QUIC Handshake.

- 1-RTT packets, from client to server and viceversa, where application data is exchanged. Here the Spin Bit feature is evaluated, in particular in QUIC packets having the Short Header.

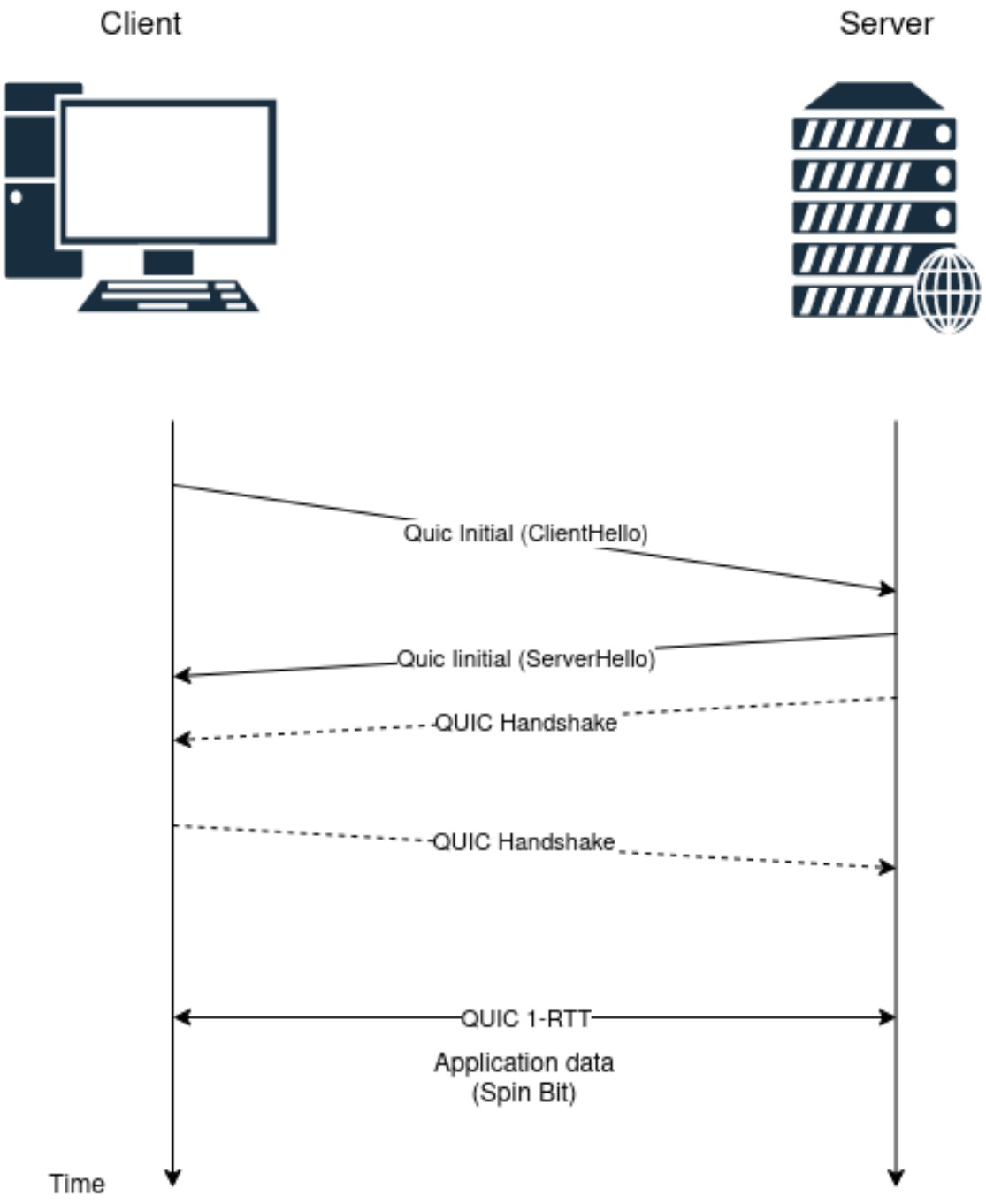


Figure 2.2: Messages exchanged for a typical QUIC connection establishment

2.2.1 Quic Header

IETF RFC 8999 [6] defines the two different headers for QUIC packets: Long Header and Short Header. They are identified by the most significant bit of the first byte being set (0 short, 1 long). The payload of QUIC packets is version specific and of arbitrary length.

QUIC LONG HEADER Packets with long header are Initial, 0-RTT, Handshake and Rerty. One of the purpose of using the Long Header is the specification of the version to be used in the connection, placed in 4 bytes: when a QUIC endpoint receives a packet with a Long Header, it is possible that it does not support that version, so it sends back a Version Negotiation Packet with a list of versions of QUIC that it supports.

```
0-RTT Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 1,
  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),
  Length (i),
  Packet Number (8..32),
  Packet Payload (8..),
}
```

Figure 2.3: 0-RTT Packet with Quic Long Header [1]

QUIC SHORT HEADER Packets using short header are designed for minimal overhead and are used after a connection is established. It is used in

packets after the connection has been established, for performances purposes. Here is where Spin Bit is located.

```
1-RTT Packet {
  Header Form (1) = 0,
  Fixed Bit (1) = 1,
  Spin Bit (1),
  Reserved Bits (2),
  Key Phase (1),
  Packet Number Length (2),
  Destination Connection ID (0..160),
  Packet Number (8..32),
  Packet Payload (8..),
}
```

Figure 2.4: 1-RTT Packet with Quic Short Header [1]

2.2.2 Spin Bit

Latency Spin Bit provides a method to compute **passive latency monitoring** measurements by on-path network observers: it makes it possible to measure per-flow RTT throughout the duration of a connection.

Spin Bit resides in the QUIC Short Header packet, more specifically in the reserved bits section, which is one of the few parts of the header that is unencrypted in the QUIC protocol.

In a nutshell, the Spin Bit value handled by the client and server generates a square wave that enables the performance measurements by on-path network observers. The algorithm will be explained in detail later, in chapter 4.

It is essential to underline that Spin Bit is an optional feature of the QUIC protocol, as defined in IETF RFC 9000 [1]: this means that an endpoint that does not support this feature must disable it. The RFC also states that QUIC implementations must allow administrators of client and servers to disable the Spin Bit either globally or on a per-connection basis. In addition to that, endpoints must disable their use of Spin Bit at least once every

16 connections, in order to ensure that QUIC connections that disable the Spin Bit are commonly observed in the network. As a result, since each endpoint disables the Spin Bit independently, the Spin Bit signal is disabled statistically on one in eight QUIC connections.

In IETF RFC 9312, [7] it is described how an on-path observer that can see traffic in both directions (from client to server and from server to client) can also use the Spin Bit to measure "upstream" and "downstream" RTTs. Upstream and downstream refer to the part of the communication between Client and observer, observer and Server respectively.

RTTs are computed by measuring the delay between the swapping of Spin Bit values on both upstream and downstream directions: this way it is possible to derive the end-to-end RTT of the communication.

Network observers then report RTT values that can be used to monitor QUIC traffic and generate useful network performance metrics.

Figure 2.5: Computation of Left and Right RTT by the network observer

Passive monitoring techniques Passive monitoring techniques allow to perform measurements of traffic on a large scale without introducing additional packets in the network. As mentioned before, Spin Bit enables passive measurements of QUIC traffic. In QUIC Short Header there are two additional bits that are reserved for future use and can be used in monitoring techniques, since they are unencrypted.

In [8] are presented two passive monitoring techniques of QUIC traffic that are based on more than one bit (Spin Bit) of the QUIC header: **Valid Edge Counter** (**VEC**) and **Delay Bit**.

- The VEC objective is to detect if an edge was valid when transmitted by the endpoint. VEC logic is the same for both endpoints. An edge is valid if it has a value greater than 0. VEC value is increased every

time a valid edge is reflected by one of the two endpoints: it represents the number of semi-paths (i.e., the path between client and server or between server and client) crossed by the edge without incurring network impairments. On the other hand, when an impairment is detected, the endpoint sets the VEC to 1, to prevent the observer computing incorrect measurements. Another case in which VEC is transmitted with value equal to 1 is when a delay threshold is exceeded: this way the VEC algorithm overcomes the Spin Bit limitation related to application delay. The observer, based on the VEC value it reads, decides whether that edge is used to start (VEC value equals to 1 or 2) or to complete (VEC value equals to 3) the RTT measurement.

- The delay measurement is based on an additional bit of the header, called *delay bit*. This measurement technique aims to overcome Spin Bit limitations of unreliable produced RTTs as soon as there are network impairments. Differently from Spin Bit, the Delay Bit is set once per round trip period. This way one packet, called *delay sample*, allows in-path network observers to measure the end-to-end RTT of the connection as the difference between two delay samples in time. Differently from VEC, this technique makes use only of one additional bit other than Spin Bit, allowing the addition of further features in the QUIC protocol.

2.3 Network performance measurement tools

In this section I will introduce the on-path network observers tools used to read the Spin Bit values throughout the communication between client and server and compute QUIC traffic RTTs.

2.3.1 Spindump

Spindump [9] is an open-source software tool developed by Ericsson used for latency monitoring for Linux operating systems. It performs passive in-network monitoring, by looking at the transport protocol characteristics and derives information about Round Trip Times for connections. It was chosen in this thesis because it supports the QUIC protocol.

In particular, the tool is able to detect the Spin Bit algorithm and its behavior. It points out the communication endpoints support of Spin Bit feature. They can be both, called Initiator and Responder, just one or none.

If both entities support the Spin Bit algorithm, the tool reports 'Spinning', together with the QUIC protocol version used in the communication; then it computes the Left and Right RTTs (previously referred to as downstream and upstream), based on Spin Bit values.

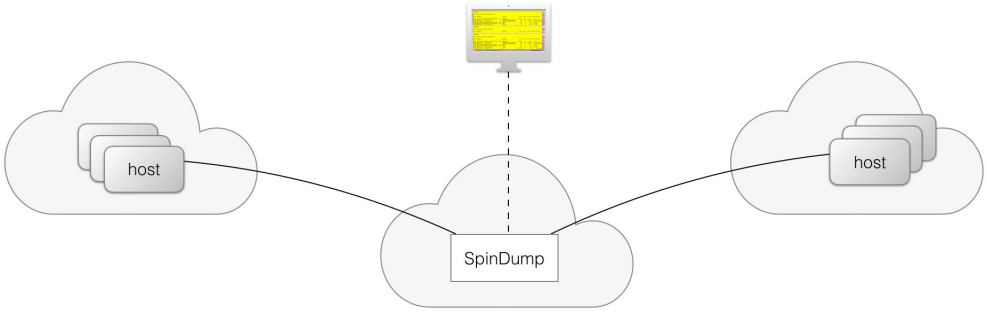


Figure 2.6: Spindump network observer [9]

Another important feature of the tool is its capability of reporting the output in JSON format. This allows to easily manage the traffic reports, which I used to compute more sophisticated analyses, as explained later in chapter 5.

In order to launch the tool the following command can be executed:

```
spindump udp and port 443 --interface lo
```

With this command a Spindump process starts, by listening and reading all packets passing through the loopback interface. In particular, it applies a filter on UDP packets on the port 443 (i.e. QUIC packets), using the same syntax of other PCAP-based tools.

The figure represents the output of the tool.

```
SPINDUMP

10 connections 98 packets 56.1K bytes (showing latest RTTs, not showing UDP, showing addresses)

TYPE    ADDRESSES                    SESSION                                    STATE   PAKS   LEFT RTT  RIGHT RTT  NOTE
QUIC    127.0.0.1 <-> 127.0.0.1      null-08a939ffad7e4e9b (47530:4443)           Up     11     466 us      90 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08b15553b5bab4b6 (44377:4443)           Up     11     578 us     202 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08291e7cb9689b6f (33066:4443)           Up     11     550 us     204 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08a06f254337f113 (41479:4443)           Up     11     567 us     201 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-083547c4ec1fad88 (43765:4443)           Up      9     312 us      40 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08441ae02a1fd446 (54951:4443)           Up      9     315 us      51 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08a891214a0ad9f9 (33202:4443)           Up      9     324 us      42 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-0881fa0ab05185b2 (43589:4443)           Up      9     309 us      42 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08494490e917475b (59335:4443)           Up      9     336 us      43 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-080052545a1b1b78 (56309:4443)           Up      9     337 us      43 us  RFC,spinning
```

Figure 2.7: Spindump report

2.3.2 TIMQuic

TIMQuic is an Android App based on Spindump open-source code.

In this thesis it was used to monitor QUIC traffic in communications including Android applications client. In particular, it was used to detect the support of Spin Bit algorithm in the Chromium application built for Android.

Like Spindump, TIMQuic is able to detect whether the spin bit feature is supported by both endpoints of communication or not and report the values of Left and Right RTTs.

To use it, it is necessary to open the app and start the capture of the packets. Then it is sufficient to navigate on the web and the application will start to report the same statistics of Spindump.

Chapter 3

Chromium open-source browser

Chromium is the open-source browser that is part of the Chromium projects together with Chromium OS. These two are the open-source versions of Google Chrome browser and Google Chrome OS.

3.1 Motivation

The choice of Chromium among the open source browsers for implementing the QUIC traffic measurements was due to three main reasons:

1. it is mainly developed and maintained by Google, which also firstly introduced the QUIC protocol.

2. its codebase is widely used by the most common browsers. For example, Chromium codebase provides the majority of code for the Google Chrome browser, which adds on top of it additional features that makes it proprietary software. Other examples of browsers based on Chromium are Microsoft Edge, Samsung Internet and Opera.

3. Quiche, the QUIC library implementation is the low level library common to all platforms: the implementation of the Spin Bit feature inside the library allows to have the feature available for the Chromium build of Android, Linux and Windows operating systems.

Figure 3.1: Chromium icon

3.2 Tools

The Chromium project provides several tools that make the programmer's job more manageable in terms of getting the code, building and testing it.

3.2.1 Versioning: Depot Tools

Depot tools is a script package to manage checkouts and code reviews. The suite contains many git workflow-enhancing tools that enable developers to manage the Chromium codebase. **Gclient** is a Python script to manage a workspace of modular dependencies that are each checked out independently from different subversion or git repositories. Dependencies can be specified on many levels: per-OS basis, relative to the parent, and variables to an abstract concept.

3.2.2 Chromium Code Search

In a vast code base like the Chromium one (about 30 millions of lines of code), it is beneficial to take advantage of this code browsing solution. It allows to navigate the code correctly by exploiting the possibility of reconstructing the call hierarchy of the functions, understanding the structure of the modules in the code, finding dependencies among different code blocks, and retrieve definitions and assignments, and intuitively viewing in the code all related calls. This was really important when inspecting the code: it allowed me to almost immediately find the network modules to be modified in order to implement the Spin Bit algorithm.

3.2.3 Building tools: ninja and gn

The project was built for both Android and Linux operating systems thanks to the usage of **gn** and **ninja** that speeded up the compilation process.

- GN is a meta-build system that, together with Ninja, composes the compilation process. It produces Ninja build files to be then compiled to produce the executable. It is designated for large projects and teams and scales efficiently to thousands of build files and tens of thousands of source files. It is also designed for multi-platform projects, and supports multiple parallel output directories, each with its configuration, in order for a developer to maintain builds targeting debug, release, or different platforms in parallel without being forced to rebuild when switching.

- Ninja is a small build system with a focus on speed. Unlike other build systems, it requires the input of higher-level files generated by other build systems, in this case gn.

3.2.4 Testing tools

Inside the Google Quiche library section, Chromium provides a client and a server executables that simulate the behavior of a network request using the QUIC protocol implementation of Chromium. In order to use them, it is necessary to first check out the Chromium source code and then build them.

quic server This is the executable that listens by default on port 6121 and serves static content. In order to run it the following steps must be performed:

- prepare static content to serve to clients;

- prepare the certificates needed to run the server. To do this it is possible to leverage the script in the net/tools/quic/certs directory, that generates server's certificate and keys. Afterwards, it is necessary to generate a CA certificate and add it to the OS's root certificate store.

The generated server's certificate will be valid for three days, so it is suggested modifying it in order to make the certificate valid for a longer period, without having to regenerate certificates frequently.

```
  quic_server \
--quic_response_cache_dir=/tmp/quic-data/www.example.org \
--certificate_file=net/tools/quic/certs/out/leaf_cert.pem \
--key_file=net/tools/quic/certs/out/leaf_cert.pkcs8
```

This is an example of command that can be used to run the server. The option –quic-response-cache-dir indicates the directory that contains the index.html file to be served by the server, while –certificate-file and –key-file contain the path to the previously generated certificate and key.

quic client This executable is used to perform requests to quic-server.

It can be launched using the following command, specifying IP address and port of the server:

```
quic_client --host=127.0.0.1 --port=6121
```

There are two important options for the quic-client that allow to perform tests in a quicker way:

- disable-certificate-verification, if the server has not valid certificate;
- allow-unknown-root-cert, if the certificate is valid, but is related to a user installed CA, like the one generated by the script mentioned before

This setup offers two main benefits: on one hand it provides control of both communication endpoints; on the other it allows to test the library implementation without having to build the entire browser.

Together with a network observer, this setup allows to perform a first QUIC traffic network performance monitoring test to detect the presence of Spin Bit algorithm support.

3.3 Network architecture

Chromium network stack is a mostly single-threaded cross-platform library. In this thesis work, I implemented the Spin Bit algorithm in the Google Quiche library.

3.3.1 Overview

Chromium supports many protocol implementations that can be found under the *net* directory, located in the root of the source code. In addition to QUIC, which is the subject of this thesis, it is possible to find other protocols: FTP, HTTP, OCSP, SPDY.

3.3.2 Google Quiche library

Google Quiche [10] is an implementation of the QUIC protocol developed by Google.

Written in C++, it is the library upon which HTTP/3 works in the Chromium browser. Despite having the same name, it is important to outline that Google Quiche is different from the Quiche library developed by Cloudflare and written in Rust.

It supports all the core features of QUIC, including:

- connection establishment;
- packet construction and parsing, with a special focus on QUIC peculiarities such as header compression, encryption, and integrity protection;
- flow and congestion control mechanism;
- stream management.

On the other hand, it does not support the Spin Bit feature.

In the Chromium project, the Quiche library, being a low level library, is shared between all the builds for the different platforms. This means that a new feature implemented in the library will be present in all platforms, as I will describe in the next chapters, together with the implementation and test of Spin Bit for the Android and Linux operating systems.

Google Quiche Call Chain In this section I report the main functions of the Chromium code that are involved in the two phases of the Spin Bit algorithm: packet reception and packet delivery.

The Chromium Code Search functionality described in Chapter 3 was extremely useful to navigate the estimated 30 millions lines of code of Chromium; it also allowed me to reconstruct the flow of incoming and outgoing packets.

In the following lines the functions that represent the flow of incoming and outgoing packets are listed, together with the file they are located in.

Incoming packets

- ProcessUDPPacket(session.cc)
- ProcessUDPPacket(connection.cc)
- ProcessPacket(framer.cc)
- ProcessPacketInternal(framer.cc)
- ProcessPublicHeader(framer.cc)
- ProcessPacketHeader(framer.cc)
- ProcessIETFPacketHeader(framer.cc)

Outgoing packets

- FillPacketHeader(packet-creator.cc)
- SerializePacket(packet-creator.cc)
- BuildDataPacket(packet-creator.cc)
- AppendPacketHeader(packet-creator.cc)
- AppendIETFPacketHeader(packet-creator.cc)

Chapter 4

Spin Bit algorithm and implementation

In this chapter I will present the algorithm of the latency Spin Bit and my implementation in Chromium from the perspective of the client and the server.

The implementation resides in the Google Quiche library, the open source library developed by Google which implements the QUIC protocol.

4.1 Spin Bit algorithm

As described in chapter 2, the Spin Bit enables passive latency monitoring from on-path network observers during a QUIC connection.

The Spin Bit algorithm starts after the completion of version negotiation and connection establishment, since Spin Bit is available in Short Header packets only.

The algorithm involves two entities, the client and the server, which behave in the following way:

1. The client sends QUIC packets having the Short Header, with Spin Bit value equal to zero.

2. The server *reflects* the Spin Bit, meaning that it reads the value of the Spin Bit of incoming packets and builds response packets having the same Spin Bit value.

3. The client, after receiving packets from the server, *flips* the Spin Bit: it reads the incoming Spin Bit value and builds new packets with Spin Bit value equal to the opposite received.

These steps are repeated for the duration of the entire QUIC connection. In this way client and server generate a square wave that an on-path network observer is able to intercept, and deduce RTT values of both sides of communication.

This way, for each QUIC connection, client and server generate a square wave of values, as reported in 4.1, where the the black lines represent packets having spin bit value equal to 0, while the green ones represent packets with spin bit set to 1. The square wave allows on-path network observers to monitor QUIC traffic. More in detail, on-path network observers are able to read the Spin Bit value throughout the duration of the connection and compute the end-to-end Round Trip Time between client and server, by looking at Spin Bit values that comes from the client and the server.

As described in IETF RFC 9000 [1], some checks must be performed before *flipping* or *reflecting* the Spin Bit value, from the client and server perspective respectively:

- **Short-Header**: the Spin Bit value is present only in packets having Short Header, hence 1-RTT packets, after the version negotiation and connection establishment are completed;

- **Packet number**: the logic of the Spin Bit is implemented in both cases only if the received packet increases the highest packet number seen on the network path;

If the checks are evaluated positively, the logic is implemented according to the perspective of the client or the server; otherwise, the Spin Bit value of next packets is not updated.

The Spin Bit algorithm, in this way, allows on-path network observers to monitor and perform QUIC traffic measurements based on a single bit of the QUIC Short Header left unencrypted for this purpose.

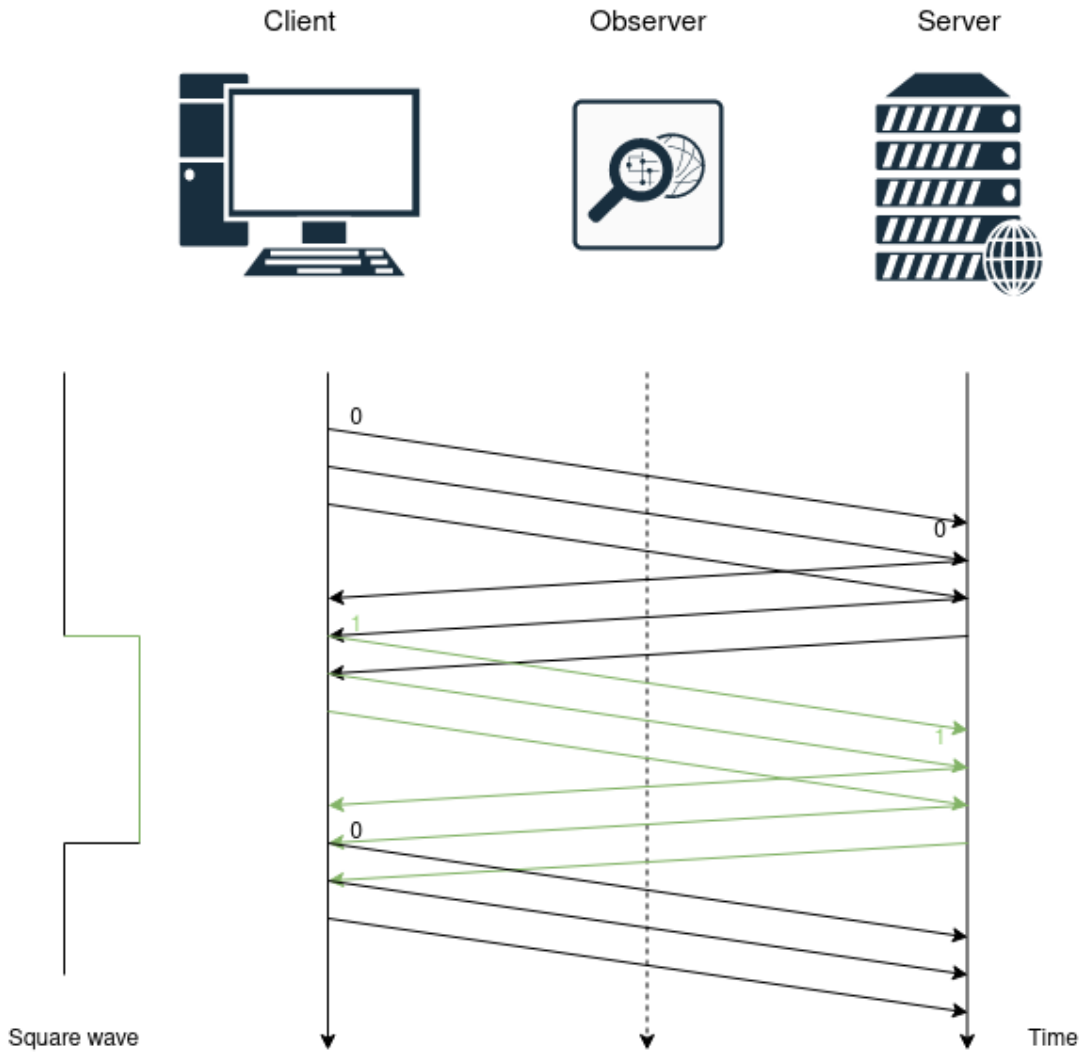


Figure 4.1: Representation of spin bit algorithm

4.1.1 Limitations

As explained before, Spin Bit represents a technique of QUIC performance measurements and traffic monitoring.

IETF RFC 9312 [7] points out the major limitations that are subject to this technique, in terms of produced measurements.

In particular, this measurement, being a passive measurement, includes all **transport protocol delay** and **application layer delay**. RTT measurements hence indicate to on-path network observers an instantaneous

estimate of the RTT experienced by the application.

For this reason, application limited and flow-control limited senders can have application and transport layer delay that are much greater than network RTT.

Another issue to be considered in the RTT measurements is the **reordering**: the Spin Bit logic itself does not have problems with it, since it is implemented only if packets increment the largest packet number received. The problem touches the network on-path observers, causing spurious edge detection and inaccurate RTT measurements, if reordering occurs across a Spin Bit flip in the stream.

4.2 Implementation

The implementation of the Spin Bit algorithm followed these major steps:

- addition of Spin Bit variable in the QUIC Header data structure and initialization in the constructor;

```
//[quic_packets.h]
struct QUIC_EXPORT_PRIVATE QuicPacketHeader {
...
bool spin_bit;
}

//[quic_packets.cc]
//default constructor
QuicPacketHeader::QuicPacketHeader()
    : ...
    spin_bit(false)
```

- addition of Macro SPIN BIT;

```
//[quic_types.h]
enum QuicPacketHeaderTypeFlags : uint8_t {
   FLAGS_DEMULTIPLEXING_BIT = 1 << 3,
   // Bits 4 and 5: Reserved bits for short header.
   FLAGS_SHORT_HEADER_RESERVED_1 = 1 << 4,
```

```
SPIN_BIT = 1 << 5,
// Bit 6: the 'QUIC' bit.
FLAGS_FIXED_BIT = 1 << 6,
```

- addition of spin bit variable in packet creator

```
[quic_packet_creator.h]
class QUIC_EXPORT_PRIVATE QuicPacketCreator {
...
void SetCurrentSpinBit(bool spin_bit) {current_spin_bit = spin_bit;}
bool GetCurrentSpinBit() const {return current_spin_bit;}
bool current_spin_bit = false;
 };
```

- checks to be performed before evaluating the Spin Bit value received: Short-Header and packet number.
- check on the perspective side: client and server
- read current value of spin bit and save in the corresponding variable;

```
bool QuicConnection::OnPacketHeader(const QuicPacketHeader& header) {

   --stats_.packets_dropped;

  //update current_spin_bit
  if(header.form == IETF_QUIC_SHORT_HEADER_PACKET){
    if(!GetLargestReceivedPacket().IsInitialized()
    || header.packet_number > GetLargestReceivedPacket()){
      if(perspective_ == Perspective::IS_CLIENT) {
        packet_creator_.SetCurrentSpinBit(!header.spin_bit);
      }
      else{
        packet_creator_.SetCurrentSpinBit(header.spin_bit);
      }
    }
  }
```

```
[quic_framer.cc]
//read incoming spin_bit
bool QuicFramer::ProcessIetfPacketHeader(QuicDataReader* reader,
                                         QuicPacketHeader* header)
    ...
if(header->form == IETF_QUIC_SHORT_HEADER_PACKET){
      QUIC_DVLOG(1) <<  ENDPOINT << "Processing short packet header";

      header->spin_bit = header->type_byte & SPIN_BIT;

      ...
}
```

- build the outgoing packet with the same value of current spin bit or dual value, if server or client respectively. Note that header.version flag tells whether the packet is a Short Header or Long Header.

```
//write spin_bit
bool QuicFramer::AppendIetfHeaderTypeByte(const QuicPacketHeader& header,
                                          QuicDataWriter* writer) {
  uint8_t type = 0;
  if (header.version_flag) {
    type = static_cast<uint8_t>(
        FLAGS_LONG_HEADER | FLAGS_FIXED_BIT |
        LongHeaderTypeToOnWireValue(header.long_packet_type, version_) |
        PacketNumberLengthToOnWireValue(header.packet_number_length));
  } else {
    type = static_cast<uint8_t>(
        FLAGS_FIXED_BIT | (header.spin_bit ? SPIN_BIT : 0) |
        (current_key_phase_bit_ ? FLAGS_KEY_PHASE_BIT : 0) |
        PacketNumberLengthToOnWireValue(header.packet_number_length));

  }
  return writer->WriteUInt8(type);
}
```

- Here the logic is implemented saving the correct value of spin bit in the header data structure.

```
[quic_connection.cc]
// Called when the complete header of a packet has been parsed.
bool QuicConnection::OnPacketHeader(const QuicPacketHeader& header) {
...
  //update current_spin_bit
  if(header.form == IETF_QUIC_SHORT_HEADER_PACKET){
    if(!GetLargestReceivedPacket().IsInitialized()
     || header.packet_number > GetLargestReceivedPacket()){
      if(perspective_ == Perspective::IS_CLIENT) {
        packet_creator_.SetCurrentSpinBit(!header.spin_bit);
      }
      else{
        packet_creator_.SetCurrentSpinBit(header.spin_bit);

      }
    }
  }
```

To sum up, in this chapter I presented the Spin Bit algorithm, pointing out the measurements that it can provide together with the limitations that are subject to passive measurement techniques.

I then provided my implementation of the Spin Bit feature in the Google Quiche library, explaining the major steps that I followed throughout the study of the code and the implementation itself.

Chapter 5

Evaluation

In this chapter I will present the methods I used to evaluate the Spin Bit implementation in terms of reliability and performance, presenting for each test:

- Environment
- Methodology and setup
- Results

The tests performed can be divided into two main categories: the first one has the objective to check that the presented on-path network observer tools are able to correctly detect the Spin Bit algorithm throughout the QUIC connections. The expected results for these tests are the report of *"Spinning"* notes by the two tools, both deployed on the same machine of the client.

The second types of test wants to verify if the RTT measurements reported by Spindump are coherent with Ping measurements, considering the Spin Bit limitations explained in Chapter 4.

Overview The tests were performed on both Android and Linux environment: on the Chromium application for Android it was verified the presence of Spin Bit implementation by means of the TIMQuic network observer (Test 3, Android section). On Linux environment, on the other hand, different executables were tested: first of all, the quic client and quic server implementations provided by the Google Quiche library (Test 1). In order to test the interoperability of the Spin Bit implementation it was verified whether the

Spin Bit algorithm was detected by Spindump using quic client and a server provided by a different QUIC implementation, called *picoquic* (Test 2). From the Test 3 on, all tests involved the usage of the full Chromium browser as client, by leveraging the Spindump feature to report measurements in JSON format to compute statistics about the RTT values produced and compare them to Ping reference values.

5.1 Test 1: quiche library

5.1.1 quic client and quic server

The first test has been performed by using the Linux executables provided by the Google Quiche library used in Chromium, with my new implementation. More in detail, after having checked out the source code of Chromium, it is possible to build the binaries quic client and quic server. The two executables are sample client and server implementations and allow to simulate the behavior of a network request using the Google Quiche library, without having to deal with the complexity of building the full browser.

Environment This first test aims to verify if the on-path network observer Spindump is able to detect the Spinning notes throughout the communication between client and server. The network communication uses the QUIC protocol implemented in the Google Quiche library, modified with the new Spin Bit feature. Both client and server have been placed in the same machine, together with the network observer Spindump.

Setup In order to run the test it was necessary to perform the following steps for each actor:

- Server

    1. download a copy of a static website in order to serve its content locally after a network request;
    2. generate valid certificates for the TLS handshake;
    3. run the server, specifying the directory with the content to be served, the certificate file and the key file

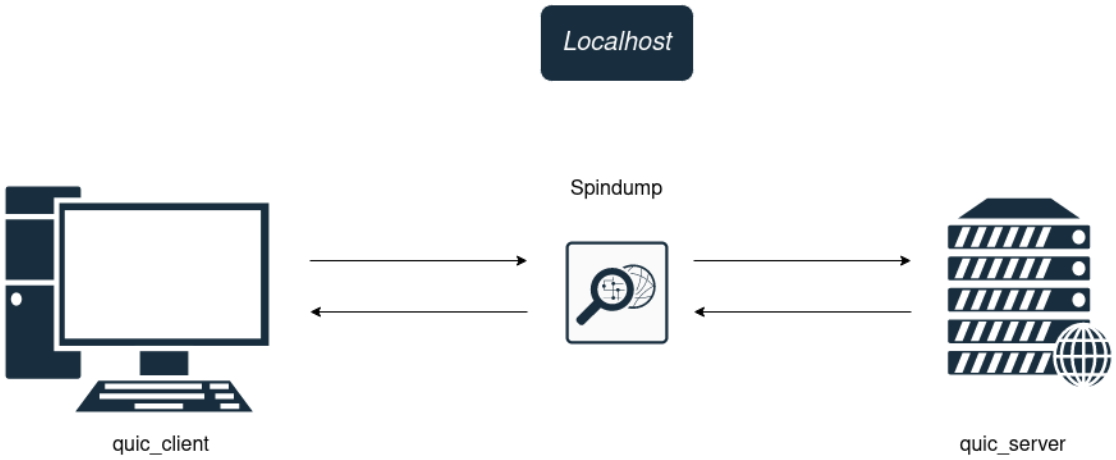


Figure 5.1: Test 1: client and server on localhost

```
 quic_server \
 --quic_response_cache_dir=/tmp/quic-data/www.example.org \
--certificate_file=net/tools/quic/certs/out/leaf_cert.pem \
--key_file=net/tools/quic/certs/out/leaf_cert.pkcs8
```

In this way we have the server listening on the default port (6121) of the loopback interface.

- Spindump

  1. Launch the on-path network observer tool in order to access packets on the loopback interface, considering only packets using UDP protocol and passing through port 6121

     ```
     spindump udp and port 6121 --interface lo
     ```

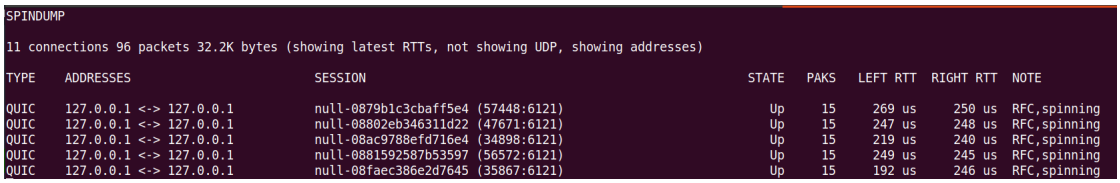
- Client

  1. Launch client specifying hostname port and URL

     ```
     quic_client --host=127.0.0.1 --port=6121
     ```

After having set Spindump to listen on the loopback interface, the client performs a request to the server, listening on a specific port (6121 by default).

Results The results of performing different network requests show that Spindump is able to detect "Spinning" behavior of the Spin Bit. Values of Left and Right RTTs are not relevant for this kind of tests since both client and server are placed in the same machine.



SPINDUMP

11 connections 96 packets 32.2K bytes (showing latest RTTs, not showing UDP, showing addresses)

TYPE	ADDRESSES	SESSION	STATE	PAKS	LEFT RTT	RIGHT RTT	NOTE
QUIC	127.0.0.1 <-> 127.0.0.1	null-0879b1c3cbaff5e4 (57448:6121)	Up	15	269 us	250 us	RFC,spinning
QUIC	127.0.0.1 <-> 127.0.0.1	null-08802eb346311d22 (47671:6121)	Up	15	247 us	248 us	RFC,spinning
QUIC	127.0.0.1 <-> 127.0.0.1	null-08ac9788efd716e4 (34898:6121)	Up	15	219 us	240 us	RFC,spinning
QUIC	127.0.0.1 <-> 127.0.0.1	null-0881592587b53597 (56572:6121)	Up	15	249 us	245 us	RFC,spinning
QUIC	127.0.0.1 <-> 127.0.0.1	null-08faec386e2d7645 (35867:6121)	Up	15	192 us	246 us	RFC,spinning

Figure 5.2: Test 1: Spindump report

5.2 Test 2: quiche and other libraries

After having tested the behavior of the Spin Bit implementation using client and server implementing the same Google Quiche library, this test aims to verify if Spindump is able to detect "Spinning" also in communications with servers implementing other QUIC libraries. In this case the setup slightly changes: while client and on-path network observer tool remain the same, the server setup is different.

5.2.1 quic client and picoquic server

In order to find libraries implementing Spin Bit feature of QUIC protocol, I referred to the QUIC Test Grid [11]. The Quic Test Grid is a test suit for QUIC, written in Go, that reports the features found in each QUIC implementation, including the Spin Bit. In particular, the test suits exchanges packets with IETF-QUIC implementations to verify whether an implementation conforms with the IETF specification of QUIC.

The grid showed that the libraries that implemented Spin Bit are **picoquic** and **lsquic**. The first library is tested in this phase, while the second is tested later.

picoquic It is an implementation of QUIC protocol compliant with the IETF standard description, developed in C. The project consist of a core library, picoquic, a test library, picoquictest, and a test program, picoquicdemo. The latter is the one I used to simulate the QUIC communication between Google Quiche client and picoquic server.

Environment Google Quiche client, picoquic server and Spindump are all placed on the same machine.

Setup

- Server

    1. run the server picoquicdemo, which is an executable that acts either as a client or as a server based on the arguments passed when launching it. By default, without specifying any argument, it acts as a server listening on port 4434.

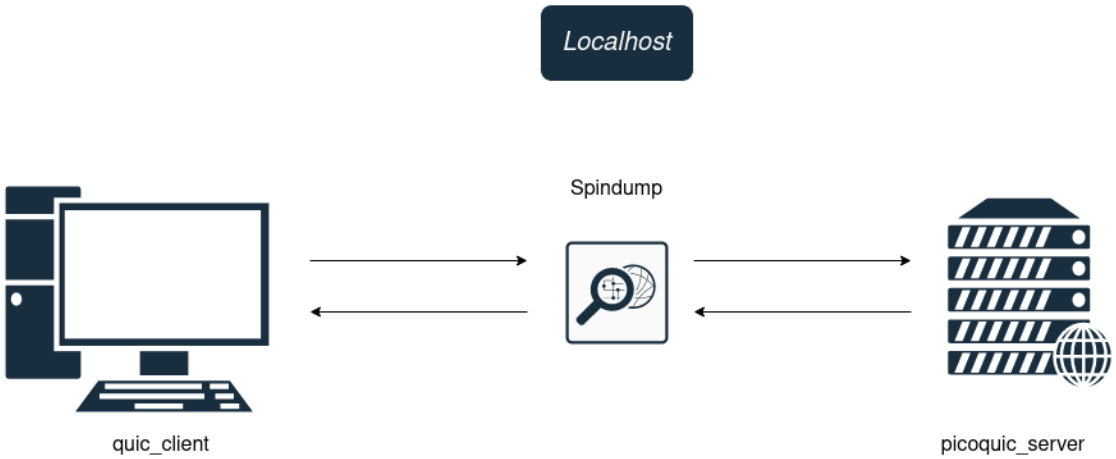


Figure 5.3: Test 2: quic client and picoquic server

```
./picoquicdemo
```

- Spindump

  1. Launch the on-path network observer tool in order to access packets on the loopback interface, considering only packets using UDP protocol and passing through port 4434

     ```
     spindump udp and port 4434 --interface lo
     ```

- Client

  1. Launch client specifying IP address and port

     ```
     quic_client 127.0.0.1 --port=4434
     ```

The setup changes from the previous test only regarding the server side.

Results Also in this case Spindump has been able to recognize Spinning events, again with similar value of Left and Right RTT since both endpoints of communication were on the same machine.

```
SPINDUMP

10 connections 98 packets 56.1K bytes (showing latest RTTs, not showing UDP, showing addresses)

TYPE    ADDRESSES                    SESSION                                          STATE   PAKS   LEFT RTT  RIGHT RTT  NOTE

QUIC    127.0.0.1 <-> 127.0.0.1      null-08a939ffad7e4e9b (47530:4443)                 Up     11     466 us      90 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08b15553b5bab4b6 (44377:4443)                 Up     11     578 us     202 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08291e7cb9689b6f (33066:4443)                 Up     11     550 us     204 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08a06f254337f113 (41479:4443)                 Up     11     567 us     201 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-083547c4ec1fad88 (43765:4443)                 Up      9     312 us      40 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08441ae02a1fd446 (54951:4443)                 Up      9     315 us      51 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08a891214a0ad9f9 (33202:4443)                 Up      9     324 us      42 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-0881fa0ab05185b2 (43589:4443)                 Up      9     309 us      42 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-08494490e917475b (59335:4443)                 Up      9     336 us      43 us  RFC,spinning
QUIC    127.0.0.1 <-> 127.0.0.1      null-080052545a1b1b78 (56309:4443)                 Up      9     337 us      43 us  RFC,spinning
```

Figure 5.4: Test 2: Spindump report

5.3 Test 3: Full Browser

Starting from this test, the entire browser will be used to verify the correct behavior of Spin Bit implementation.

5.3.1 Chromium and litespeedtech.com

This tests aims to verify the presence of Spinning events detected by Spindump, using the entire browser as client of the communication and the production ready website litespeedtech.com as a server. This server uses **lsquic** library as implementation of QUIC protocol, which supports the Spin Bit feature, as reported in the Test Grid mentioned before.

Environment

- Client: Chromium on localhost;
- Server: litespeedtech.com;
- On-path network observer: Spindump on localhost.

Setup

- Client: Chromium, build of the entire browser, with the modified Google Quiche library implementing the Spin Bit feature.
- Server: litespeedtech.com web server.
- Spindump: listening on network interface:

```
spindump upd and port 443 --interface wlp1s0
```

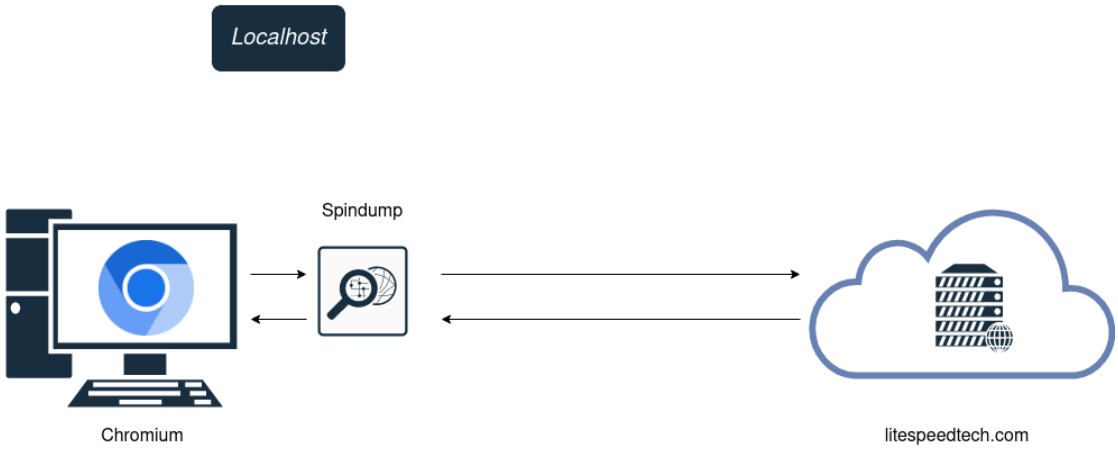


Figure 5.5: Test 3: Chromium and litespeedtech.com (Linux)

55 connections 1.2K packets 808.9K bytes (showing latest RTTs, not showing UDP, showing addresses)

TYPE	ADDRESSES	SESSION	STATE	PAKS	LEFT RTT	RIGHT RTT	NOTE
QUIC	2a01:e11:5401:3790:d228:c250:212..	null-null (51033:443)	Starting	472	n/a	n/a	V.0xffffffff,no spin
QUIC	192.168.1.176 <-> 52.55.120.73	null-0846c49d64da9a18 (53002:443)	Up	255	4,2 ms	124,8 ms	RFC,spinning
TCP	192.168.1.176 <-> 52.55.120.73	48506:443	Closed	71	55 us	101,5 ms	
TCP	2a01:e11:5401:3790:d228:c250:212..	46904:443	Up	42	27 us	10,7 ms	
QUIC	2a01:e11:5401:3790:d228:c250:212..	null-08e88b9cfe0114e8 (60972:443)	Up	41	n/a	8,9 ms	RFC,no R-spin
QUIC	2a01:e11:5401:3790:d228:c250:212..	null-08cef3e4f8ebb814 (51894:443)	Up	32	n/a	6,8 ms	RFC,no R-spin
QUIC	2a01:e11:5401:3790:d228:c250:212..	null-08f50e47111e94df (45770:443)	Up	32	n/a	6,2 ms	RFC,no spin
TCP	2a01:e11:5401:3790:d228:c250:212..	59398:443	Up	27	1,2 ms	9,6 ms	
QUIC	2a01:e11:5401:3790:d228:c250:212..	null-08ed3abb25e01ad1 (47941:443)	Up	25	n/a	6,5 ms	RFC,no R-spin
TCP	2a01:e11:5401:3790:d228:c250:212..	59404:443	Up	24	1,6 ms	11,5 ms	
TCP	2a01:e11:5401:3790:d228:c250:212..	38474:443	Up	24	184 us	10,3 ms	
TCP	2a01:e11:5401:3790:d228:c250:212..	44774:443	Up	23	70 us	21,0 ms	
QUIC	2a01:e11:5401:3790:d228:c250:212..	null-08c377eb8773843b (60059:443)	Up	19	n/a	6,4 ms	RFC,no spin
QUIC	2a01:e11:5401:3790:d228:c250:212..	null-08d53b067bbebe15 (36615:443)	Up	18	n/a	5,2 ms	RFC,no R-spin
QUIC	2a01:e11:5401:3790:d228:c250:212..	null-08fe6d4c8434abcd (43432:443)	Up	13	n/a	8,4 ms	RFC,no spin
TCP	2a01:e11:5401:3790:d228:c250:212..	37310:443	Up	12	41,1 ms	5,3 ms	
TCP	2a01:e11:5401:3790:d228:c250:212..	36170:443	Up	10	46,2 ms	5,2 ms	
TCP	2a01:e11:5401:3790:d228:c250:212..	36176:443	Up	10	41,4 ms	5,9 ms	

Figure 5.6: Test 3: Spindump report

Results Here we can see a different behavior with respect to previous tests: Chromium opens different connections while loading the web page. Spinning is detected in the connection with the web server endpoint, which implements the Spin Bit. By performing several requests it is possible to notice that Spindump continues to detect Spinning and updates accordingly the values related to Left and Right RTT.

Android platform The same test has been then conducted on the Android platform: in this case the client is the full browser built for Android platform, which implements the same Google Quiche library of the build for Linux, hence with the Spin Bit feature. The on-path network observer in this case is the previously described TIMQuic.

Results reported by TIMQuic are the same: it is possible to notice *Spinning*

events, meaning that also Chromium for Android correctly implements Spin Bit feature.

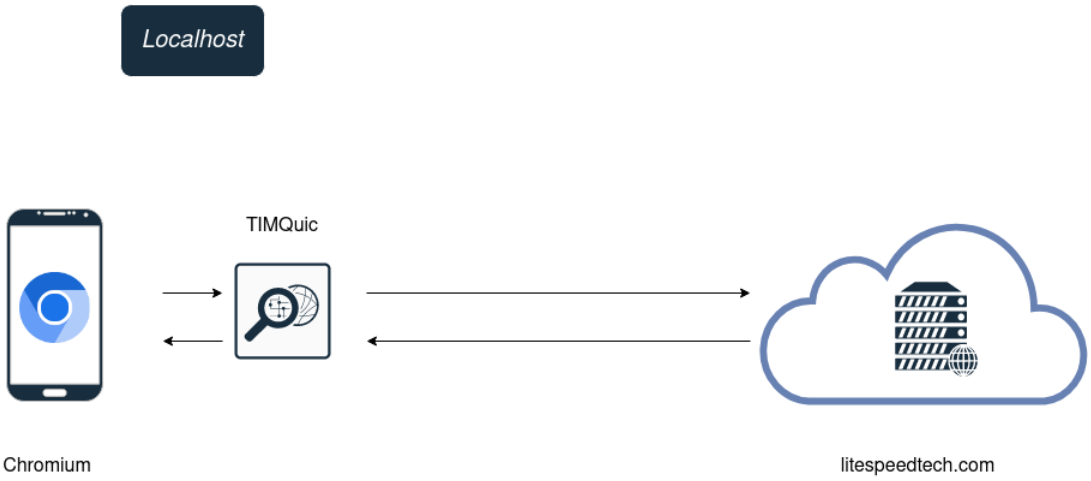


Figure 5.7: Test 3: Chromium and litespeedtech.com (Android)

5.4 Test 4: Chromium and OpenLiteSpeed

In this test I want to verify the presence of Spin Bit feature in the **lsquic** protocol implemented in OpenLiteSpeed.

OpenLiteSpeed (OLS) is the open-source web browser of the enterprise LiteSpeed server (LSWS). OLS and LSWS have in common the LSCache engine, HTTP/3 (and QUIC) and many other features, while they differ in advanced ones, available only in the enterprise edition. In this context the support of HTTP/3 and QUIC is granted, so it is sufficient to use the open-source edition to test the Spin Bit behavior. In particular, the library implementing QUIC is lsquic, that showed the support of the Spin Bit in the previous test.

I installed the OLS web server on an EC2 instance of the AWS infrastructure, in order to have full control of both endpoints of communication. This allowed me firstly to detect Spinning events and then to analyze produced RTT measurements.

5.4.1 Chromium and OLS - AWS

Environment

- Chromium browser on localhost
- Spindump on loclahost
- OpenLiteSpeed web server on AWS EC2 instance in North Virgina

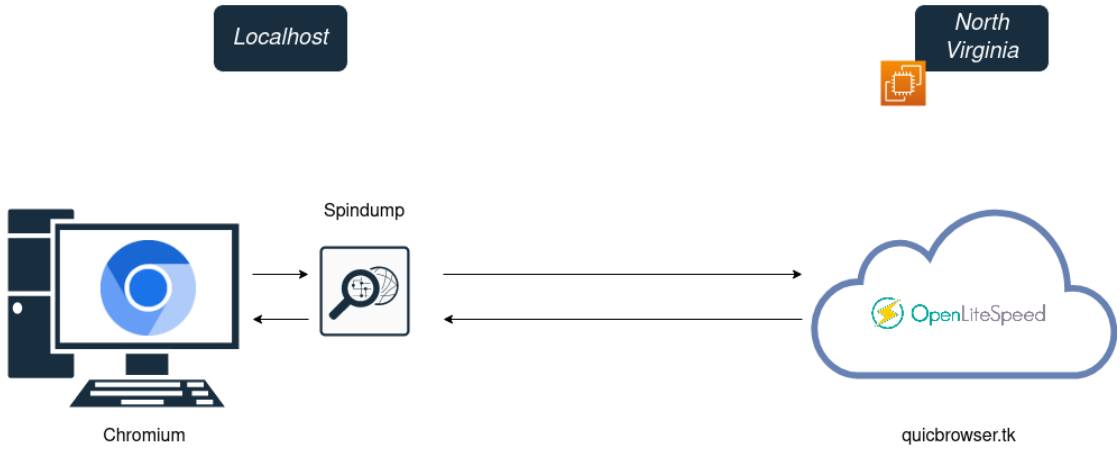


Figure 5.8: Test 4: Chromium and OLS (North Virginia)

Setup OLS web server was installed in an AWS EC2 instance in the North Virginia region. In order to do so, it was necessary to carry out the following steps:

- create a valid certificate with Let's Encrypt, based on a valid domain name. Notice that this step is necessary since Chromium does not accept self-signed certificates, so it would not be possible to complete the QUIC Handshake and establish the connection.
- setup the Route 53 service to point at the newly deployed instance with the OLS web server, with related nameservers managed by Route 53.
- Launch an EC2 instance in North Virginia region with Security Groups configured to accept inbound UDP traffic from port 443 from any IPv4 address.

Results Spindump is able to detect *Spinning* events: it is possible to establish a connection with the QUIC protocol, after having completed the QUIC handshake, using the certificates issued by Let's Encrypt and a valid domain. This way, I have proved that open-source version OLS uses the lsquic library with the support of Spin Bit feature as well as the enterprise LSWS edition, and it is possible to use the OLS web server in different AWS Regions in order to analyze the RTT measurements produced and to compare them with Ping reference values.

5.5 Test 5: Measurements

This is the second part of the tests, and the most important one, in which I analyze the RTT values reported by Spindump in the communication between Chromium and OLS web server.

The objective of these tests is to compare the RTT values recorded by Spindump with Ping values, on different QUIC connections. In all tests Spindump was placed on localhost, while OLS web server was installed in AWS EC2 instances in London and Frankfurt regions.

For each region, the tests involved downloads of files of different sizes (20MB, 100MB and 200MB) over QUIC connections.

Measurements This paragraph describes how I analyzed the QUIC connections report produced by Spindump. I took advantage of Spindump ability to export its output in JSON format.

```
spindump udp and port 443 --interface wlp1s0 --textual --format json
```

This way, it was possible to save the Spindump report in a log file, to be analyzed in a second moment.

In order to analyze JSON files, I wrote a script in Python, leveraging the Pandas and Plotly libraries for big data analysis and plotting graphs respectively. More in detail, Pandas uses the concept of Data Frame to represent tables and perform aggregated statistics on this specific data structure.

The script performs the following high-level functions:

- reads the log file in JSON and save measurements in a DataFrame data structure;
- filters the DataFrame by considering only *"Spinning"* events;
- computes statistics about Right and Left RTTs (min, avg, max, count). Each row of the DataFrame represents a "Spinning" event, to which is linked the corresponding Left or Right RTT. The Pandas library is used to compute for the Left and Right RTTs the following measures:
  - minimum: the minimum value present in the DataFrame;
  - maximum: the maximum value present in the DataFrame;

  - count: the number of non-null values present in the DataFrame;
  - average: the average value of the DataFrame, as the sum of the values divided by the *count*.

  All these values are reported for 50, 75, 80, 90 and 95 percentile, in order to provide a more complete view on the measurements and not to be affected by outliers in the statistics. More in detail, the *quantile* function of Pandas library was used to find the corresponding value, and later the DataFrame was filtered to consider only values lower or equal to it.

- plots graphs with the obtained results;
- exports results in xlsx files.

Environment

- Client: Chromium browser
- Network observer: Spindump on localhost
- Server: OLS on AWS EC2 instances (London and Frankfurt)

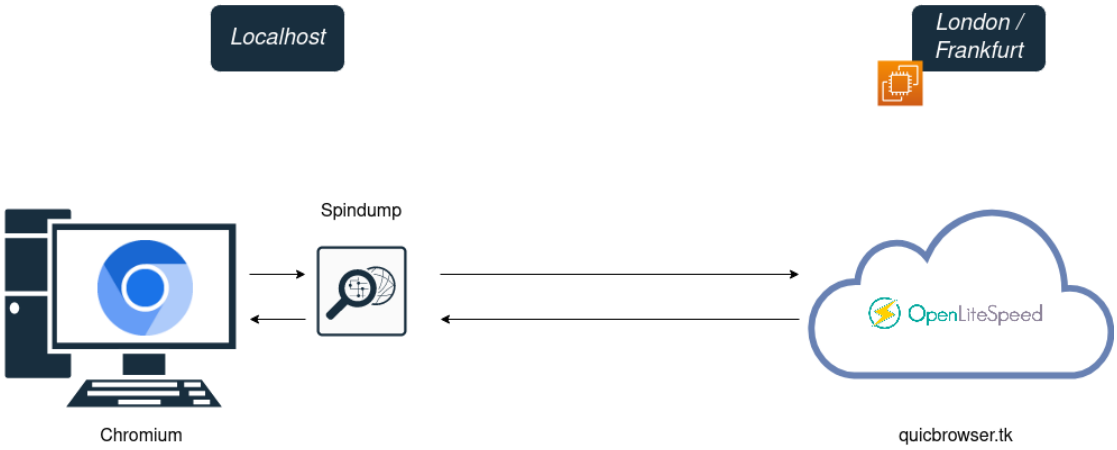


Figure 5.9: Test 5: Chromium and OLS (London/Frankfurt)

Setup The setup of Chromium and Spindump is the same of the previous tests.

In order to deploy the server on EC2 instances it has been necessary to:

- Create an EC2 instance in the given region (London or Frankfurt)
- Install OLS on the EC2 instance
- Configure the AWS Security Group in order to allow inbound and outbound connections over the QUIC protocol (UDP port 443)
- Configure the AWS Route 53 service with the EC2 IP address and correct nameservers
- Configure the index page of the server to download 20MB, 100MB and 200MB.

Spindump is then configured to export JSON files.

After the 'listening' phase, the script computes the statistics and generates graphs.

More in detail, the script generates a report with Right and Left RTT values for every communication. These values are then compared to Ping values generated over the relative EC2 instance, in order to evaluate the differences. Ping values were obtained by increasing the default frequency of the tool: instead of sending ICMP packets every 1 second, the ICMP packets were sent every 100 milliseconds. This way it was easier to detect small network impairments, which would not be detected otherwise. The tool finally reports minimum, maximum and average values detected, based on the number of ICMP packet sent in the test: this number was set to 100.

Results In the following graphs the results obtained in the different measurements are presented.

- The *y-axis* represents the RTT values expressed in microseconds [1x10-6 s].
- The *x-axis* is divided in five groups, corresponding to the percentile values. Each group represents the subset of the entire DataFrame related to the percentile value. For each group three bars are plotted, representing minimum (min), average (mean) and maximum (max) RTT.

The horizontal lines represent the *minimum*, *average* and *maximum* Ping reference values.

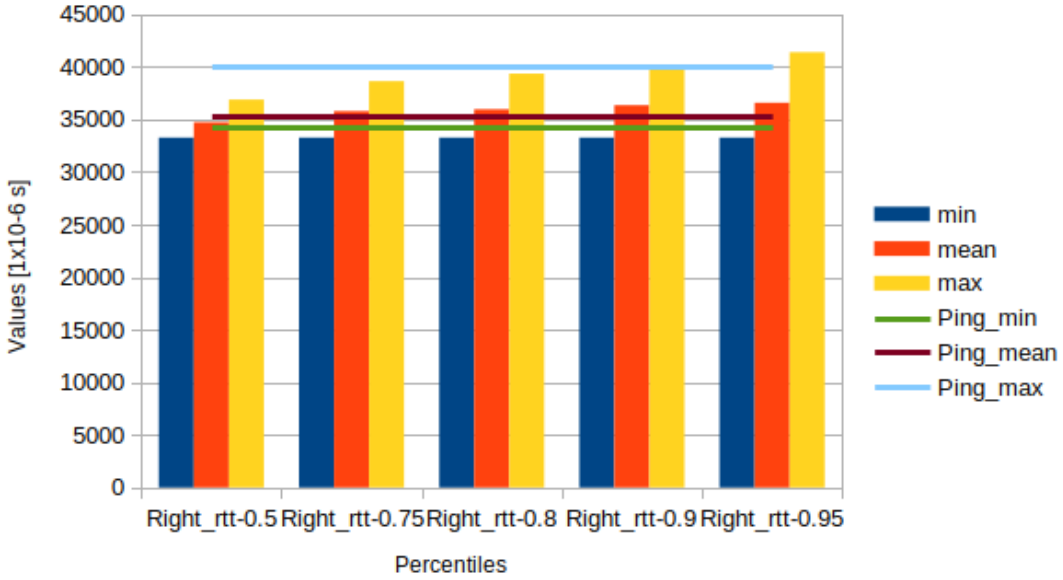


Figure 5.10: Download of 20 MB (London)

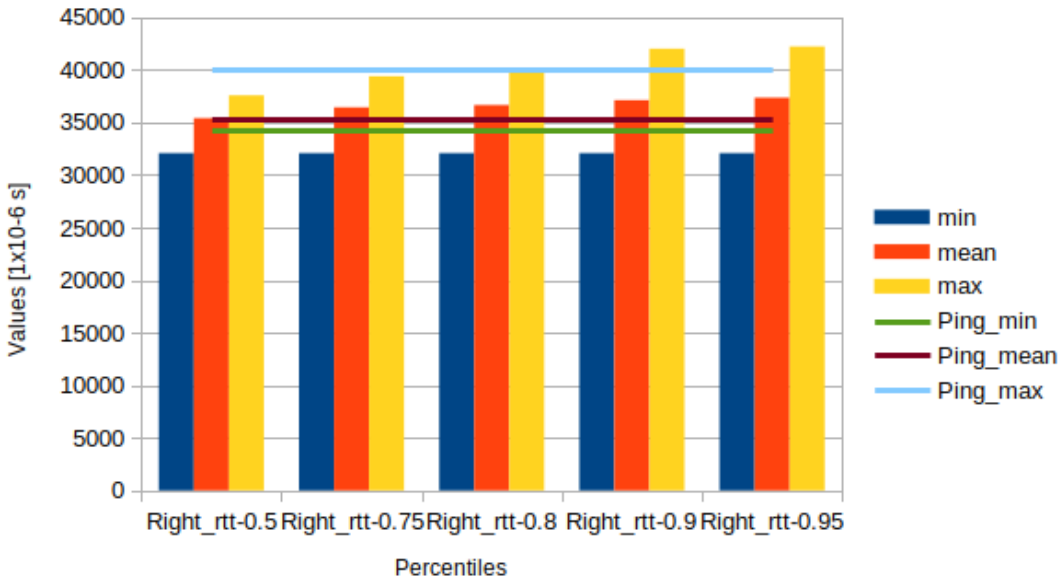


Figure 5.11: Download of 100 MB (London)

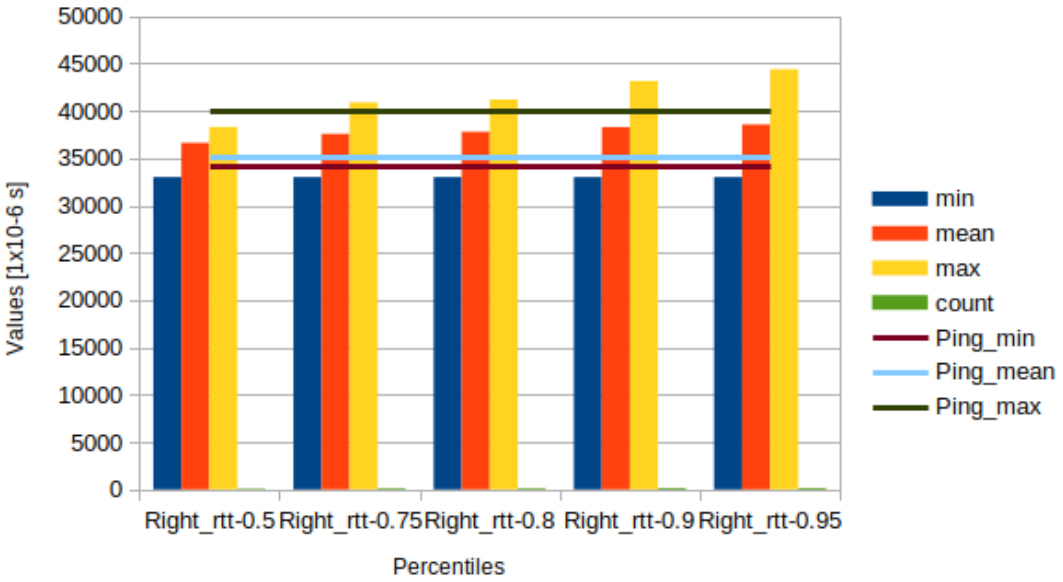


Figure 5.12: Download of 200 MB (London)

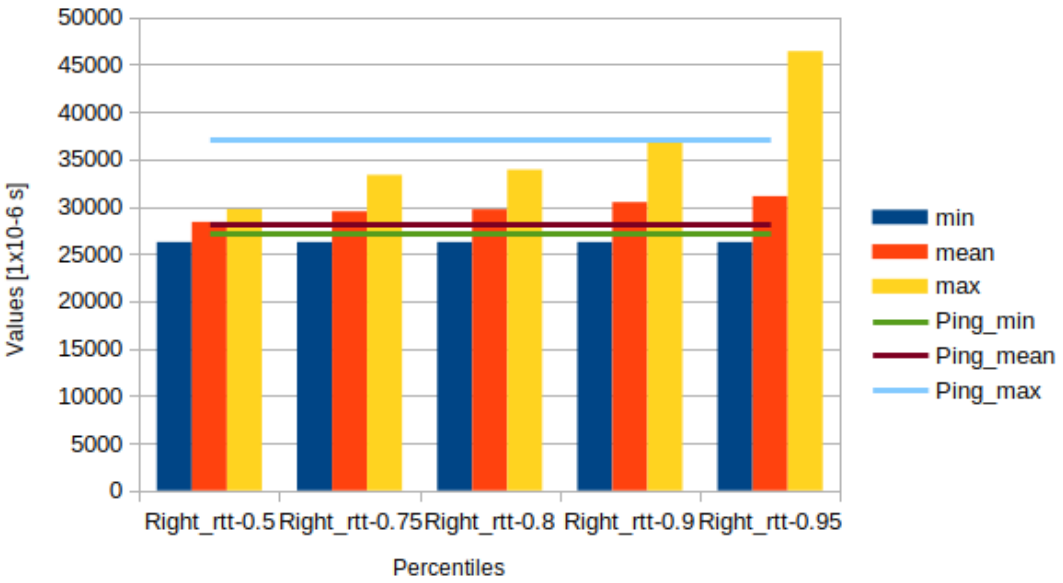


Figure 5.13: Download of 20 MB (Frankfurt)

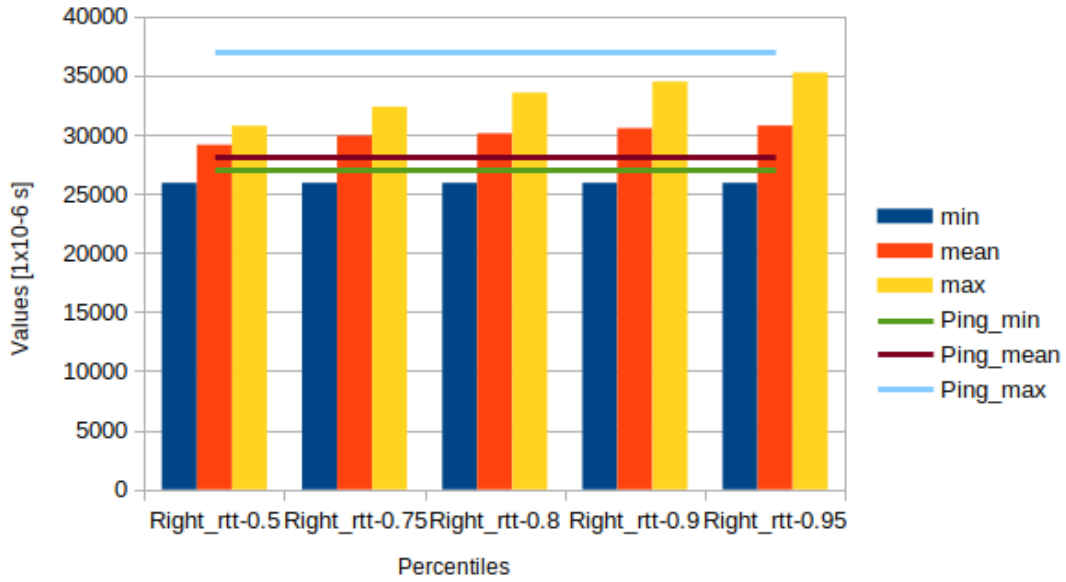


Figure 5.14: Download of 100 MB (Frankfurt)

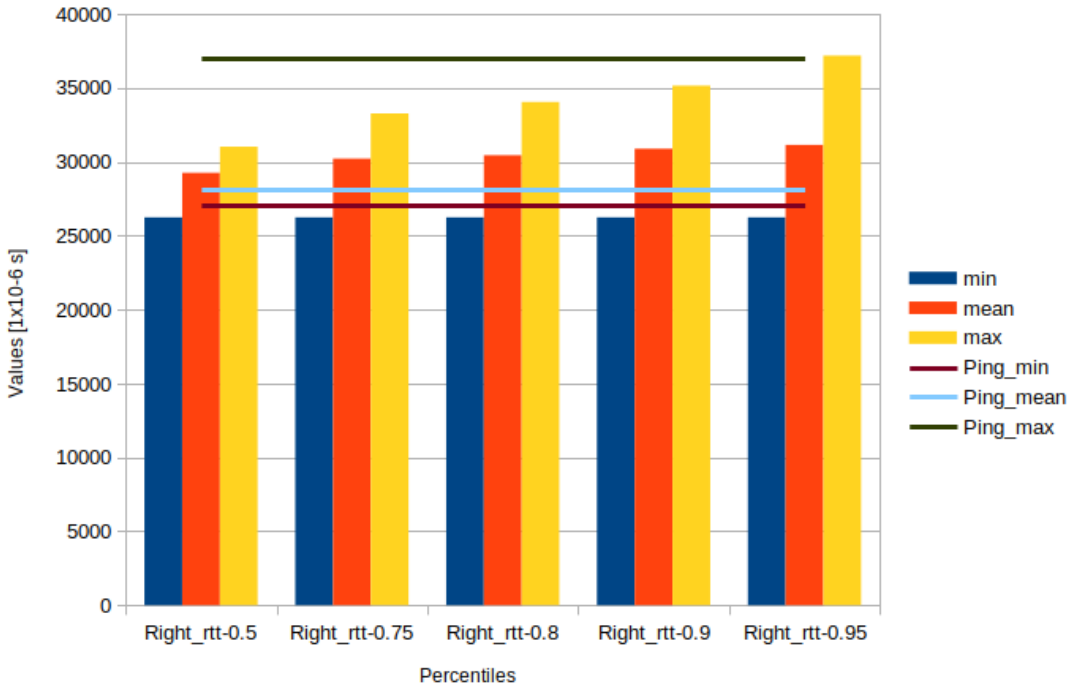


Figure 5.15: Download of 200 MB (Frankfurt)

Results show that average Ping values are slightly lower than Left and Right RTT measurements reported by Spindump based on Spin Bit values. It is important to outline that Left RTT values are not relevant in this setup since they represent the RTT between network observer and client, both placed in the same machine. For this reason Left RTT values are negligible with respect to end-to-end RTT, which is represented by the Right RTT.

The difference between Right RTT and Ping is on average of about few milliseconds. This is due to the different protocol we are analyzing, ICMP and QUIC, and the payload of the reqests: in the first case, with Ping, few packets are sent, while on the other case with QUIC protocol are sent many more packets, that can fill queues in the network path.

As mentioned before, another aspect to consider is the transport and application layer delays that affect QUIC packets, differently from the ICMP ones.

For these reasons results are the expected ones, since QUIC measurements based on Spin Bit represent the instantaneous delay perceived by the application. Analyzing the different percentile values, it is possible to notice how the RTT values increase, even if they remain similar to Ping values until 95 percentile. Beyond that point the differences in the values are more relevant: this is caused by few inaccurate RTT values which strongly affect the analysis when considering all the measurements.

Another piece of evidence of the reliability of the measurements is the fact that the minimum RTT reported by Spindump, which represents the network latency as mentioned in IETF RFC 9312 [7], is coherent with Ping values.

Chapter 6

Conclusions

This thesis has provided a QUIC traffic performance measurement implementation, based on the Spin Bit value, in the Chromium browser for both Android and Linux platforms.

After a general overview of QUIC and HTTP/3 protocols, I focused on the analysis of Chromium codebase and tools in order to detect the right components to modify for the implementation of the new feature.

With the purpose of evaluating the implementation, different tests were conducted.

The first ones aimed to verify the correct behavior of Spin Bit algorithm, with an on-path network observer. Here the focus was on the detection of *Spinning* events reported by the tools, meaning that Spin Bit feature was supported by both client and server.

The last tests aimed to analyze the RTT measurements produced by the network observer. I installed the OLS web server which supported Spin Bit feature in AWS EC2 instances in different regions in order to check if the values produced by the tool were coherent with the values produced by Ping measurements.

The tests involved the download of files of different sizes (20MB, 100MB, 200MB) from the AWS London and Frankfurt regions. Results showed that the values obtained by Spindump, hence based on Spin Bit, were on average slightly higher than the ones produced by Ping. This is an expected result, since Spin Bit-based measurements are subject to network and application layer delays, differently from ICMP values.

To conclude, this thesis has provided a first implementation of QUIC traffic measurement for the Chromium browser, based on one of the few bits

available in the QUIC packet header.

Both the Spin Bit implementation and the testing environment pave the way for further researches and development of new QUIC traffic measurement techniques in Chromium, based on the combination of more than a single bit, in order to overcome Spin Bit limitations.

Bibliography

[1] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000 (cit. on pp. 11, 18, 22, 23, 36).

[2] Mozilla Developer Network. *Evolution of HTTP*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP (cit. on p. 16).

[3] Mike Bishop. *HTTP/3*. RFC 9114. June 2022. DOI: 10.17487/RFC9114. URL: https://www.rfc-editor.org/info/rfc9114 (cit. on p. 17).

[4] Martino Trevisan, Danilo Giordano, Idilio Drago, and Ali Safari Khatouni. «Measuring HTTP/3: Adoption and Performance». In: *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*. 2021, pp. 1–8. DOI: 10.1109/MedComNet52149.2021.9501274 (cit. on p. 19).

[5] Eva Gagliardi and Olivier Levillain. «Analysis of QUIC session establishment and its implementations». In: *13th IFIP International Conference on Information Security Theory and Practice (WISTP)*. Paris, France, Dec. 2019, pp. 169–184. DOI: 10.1007/978-3-030-41702-4_11. URL: https://hal.archives-ouvertes.fr/hal-02468596 (cit. on p. 19).

[6] Martin Thomson. *Version-Independent Properties of QUIC*. RFC 8999. May 2021. DOI: 10.17487/RFC8999. URL: https://www.rfc-editor.org/info/rfc8999 (cit. on p. 22).

[7] Mirja Kühlewind and Brian Trammell. *Manageability of the QUIC Transport Protocol*. RFC 9312. Sept. 2022. DOI: 10.17487/RFC9312. URL: https://www.rfc-editor.org/info/rfc9312 (cit. on pp. 24, 37, 60).

[8] Fabio Bulgarella, Mauro Cociglio, Giuseppe Fioccola, G. Marchetto, and Riccardo Sisto. «Performance measurements of QUIC communications». In: July 2019, pp. 8–14. ISBN: 978-1-4503-6848-3. DOI: 10.1145/3340301.3341127 (cit. on p. 24).

[9] URL: https://github.com/EricssonResearch/spindump (cit. on pp. 25, 26).

[10] Google. *Quiche*. https://quiche.googlesource.com/quiche. Accessed March 12, 2023. 2021 (cit. on p. 33).

[11] Université catholique de Louvain. *QUIC Tracker*. https://quic-tracker.info.ucl.ac.be/grid. Accessed March 11, 2023 (cit. on p. 47).