UNIVERSITÀ
DI TRENTO

Department of Information Engineering
and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Real-Time Monitoring of the QUIC Protocol

Supervisors
Domenico Siracusa
Gianni Antichi

Student
Matteo Franzil

Academic Year 2021/2022

When the game is over,
the pawn and the king
go back to the same box.

Acknowledgements

For the thesis work, I would like to express my deepest gratitude to my two supervisors, prof. Domenico Siracusa and prof. Gianni Antichi. They have been instrumental in providing support and advice throughout the journey. Domenico's sheer patience and encouragement have been a source of strength and have always made me explore new paths. On the other hand, Gianni's passion and enthusiasm went far to instil a deep curiosity in me and set me on the way to investigating this exciting research area.

Secondly, I must also thank Simone Magnani, who has been nothing short of invaluable to me during the project. He took the time to answer all my questions when I was in need, providing me with crucial technical insights and serving as an invaluable teacher. His kindness and support are difficult to repay.

No word in Italian, English, or any other language, could remotely describe these past five years.

I arrived in Trento as a 19th-year-old full of expectations, hungry for change. There were periods of exaltation and happiness, followed by frustration, anguish and torment. I have made friendships that will (hopefully) last my whole life, others a bit less. Some of them were lost, and will not come back. I have understood what I do not want to do in my life, and perhaps also what I want to do. I made mistakes, several of them. I strived for change. I took the best from the people I met, and I hope I gave them back what they gave to me.

I sincerely thank everyone I met in these five years, for everything, be it happiness, sorrow, or just plain normality.

Finally, I must also thank all my family from the bottom of my heart. In particular: my sister, Alessia; my mom and dad, Maria Grazia and Daniele; my uncles Massimo, Antonello, Pierpaolo, and my aunts Lorenza, Bianca, Eleonora and Graziella.

Abstract

Modern network infrastructure increasingly requires real-time monitoring solutions to safeguard it from possible anomalies and incoming attacks, ensuring system health and performance. This is especially important in large-scale networks, in which an increasing number of users depend on high-throughput, low-latency services. Service providers are responsible for the security and reliability of the offered service and must deploy monitoring systems with a careful balance between their capacity, expressivity, precision, and computational efficiency. Most monitoring systems concentrate on TCP and HTTP, arguably the most utilised Internet transport and application protocols. In the last few years, the two protocols have demonstrated some shortcomings in flexibility, multiplexing, and latency management, both in high-speed and congested networks. A recent standard, QUIC, is emerging as a possible replacement for TCP in these environments: it reimplements TCP's reliable data delivery while guaranteeing faster performance, thanks to the usage of UDP as the underlying protocol. Not only does it bring new features to the table, but also different semantics, algorithms, and cryptographic properties, rendering classic monitoring approaches unviable and requiring new choices when designing monitoring software tailor-made for QUIC. To address this novel challenge, we propose the design of a new monitoring system for QUIC-based network flows, QUIC Watchful Information Collector (QWIC). Our solution proposes an innovative, query-based approach based entirely in userspace. Packets are first acquired from the network using state-of-the-art tools; then, a separate module deals with packet parsing, gathering relevant data, and generating alerts when certain conditions are met. This thesis describes the system design and implementation in detail and documents a set of experiments taken from the literature to evaluate the tool's performance. We designed our system to be as flexible and decoupled as possible, facilitating its expansion and further development. Results are promising, showing memory overheads in the order of a few megabytes per flow and negligible delays. The thesis concludes with a discussion of future work and opportunities in terms of technical improvements and research directions.

Contents

List of Figures

List of Tables

List of Codes

1
Introduction

Since first seeing the light, the Internet has undergone many changes in its architecture, capacity and capabilities. First conceived as a U.S.A.-owned communication network for strategic military purposes, over the years, the Internet has grown into a communication network of planetary proportions, with millions of users, communities and organisations relying exclusively on it to communicate.

With such a rapid pace of change and development, the complexity of the global network has also increased dramatically. Today's Internet is a complex system of numerous components through which information is disseminated and communication is enabled globally. Modern systems require speed, reliability, and security to work effectively. To accomplish and maintain these qualities, modern networks need to be constantly and meticulously monitored to identify and, if necessary, mitigate problems as they arise.

Throughout the last thirty years, the Transmission Control Protocol (TCP) has been at the Internet's core, providing the fundamental means for reliably transferring data between endpoints. Despite its age and several improvements, its design has seen little change. Well-defined semantics and predictable behaviour made TCP easy to monitor and diagnose. However, this status quo is set to be disrupted.

First developed at Google, then proposed as an IETF Draft and finally standardised in 2021, the QUIC protocol offers several significant improvements to the TCP protocol. Running on top of the User Datagram Protocol (UDP), QUIC is designed to enable data multiplexing, reduce round trip times, and offer greater flexibility, among other improvements. However, such features come with a price. Designed from the start to be encrypted by default and to expose as little information as possible to external parties, with radically different semantics and greater complexity, QUIC poses significant challenges to observability tools and technologies.

While QUIC may not seem to be a particularly prominent protocol choice for the Internet of the present, an overwhelming amount of research points out that QUIC usage is rising quickly. In addition, major web browsers have already started implementing support for QUIC. Soon, more application layer protocols could use QUIC as the underlying transport layer. In light of these new trends, monitoring QUIC protocols could prove vital in the future.

In this work, we will explore the disruptive changes imposed by the QUIC protocol and how they affect current monitoring technologies. Indeed, the protocol's semantics, properties, and structure are so different from TCP's that classic approaches are no longer viable. New strategies must be devised and implemented, and current monitoring technology must be adapted to capture QUIC's semantics and properties. We do not aim to provide a done-and-done solution for the problem but rather to establish the foundations upon which such a system could be built: what compromises must be made, which metrics to focus on and which problems could be expected.

1.1 Thesis structure

The work is organised as follows. Chapter 2 presents the current state of the art of the research area and discusses the main motivations behind the project, including the challenges of designing an efficient and functional monitoring system for QUIC. Moreover, it provides a brief historical overview of the subject. On the other hand, chapter 3 focuses on the theoretical background of the topics discussed in this thesis. This chapter is strictly technical and provides a description of QUIC's main properties, semantics, key operations and features.

Next, Chapter 4 proposes a Python-implemented proof-of-concept, focusing on its main design features and strengths. This chapter is the core of the thesis and details the system's functionalities along with several code snippets. Chapter 5 describes the experiments performed to evaluate the tool's

performance. It includes several case studies taken from the literature, which have been chosen to demonstrate the capabilities of the alert system. Next, some graphs are included, which show the numerical results of the experiments that have been run. Finally, in chapter 6 some general conclusions are drawn, and some possible future work is presented.

2
Motivation and related work

This chapter provides the motivations for this research and briefly reviews the related literature. First, a brief history of QUIC is presented. Then, examples of the deployment of QUIC and its usage in the industry are presented. Next, details are provided on how the QUIC protocol brings novel challenges to the table, how they can be overcome, and at what cost. We then discuss the state of the research in the field of QUIC monitoring and what are the leading tools. Finally, we provide a brief overview of the novelties of this work.

This chapter does not focus on the technicalities of the QUIC protocol. These are instead covered in chapter 3.

2.1 History of QUIC

2.1.1 Reliable data delivery

Over the years, the Transmission Control Protocol (TCP) has cemented itself as the *de-facto* standard for communication over the Internet [99]. First established as RFC 793 in 1981 [70], TCP was the first Internet protocol to offer reliable, ordered, and error-corrected data delivery. In the mid-1990s, TCP was paired with the HyperText Transfer Protocol (HTTP) — an application layer protocol — to provide access to the world wide web. As its usage grew, so did the shortcomings of what was now a several-decade years old protocol.

In the same period, HTTP underwent several revisions to increase its performance. HTTP/1.1 was released in 1999 [22–24, 65], supporting multiple TCP connections per request, and allowing them to be reused to make multiple resource requests. However, it also introduced a serious performance flaw, the so-called *head-of-line blocking*. Head-of-line blocking occurs during network congestion when a network flow is saturated, or one or more packets are dropped. In turn, this causes all subsequent packets to be delayed [96]. In HTTP/1.1, when the number of allowed parallel requests in the browser is exceeded, subsequent requests are forced to wait for the first ones to finish.

HTTP/2 was released in 2015 [3, 86] and notably introduced the possibility to multiplex requests over a single TCP connection. This approach addressed the head-of-line blocking issue at the application level but not at the transport level. Indeed, this allowed multiple HTTP requests to be intertwined, increasing efficiency. However, a single TCP packet lost in a flow would still cause a delay in all the multiplexed requests in that connection.

This situation — along with the difficulty of modifying TCP's core logic — contributed to the need to develop a new transport layer protocol.

2.1.2 The birth of QUIC

The QUIC protocol (pronounced *quick*; initially an acronym for "Quick UDP Internet Connections") was first formalised in 2012 at Google by Jim Roskind [46, 98]. It was first envisioned as a full replacement for both TCP and HTTP, a cross-layer protocol that would take care of both the transport and application layers. The design goals were made clear from the start. Internet bandwidth could arbitrarily increase, but round trip times, governed by the speed of light, could not. The evolution of the Internet into a low-latency, high-speed system would require new approaches that evolved with this new paradigm [74].

To reach this goal, Google chose to build the QUIC protocol on top of UDP rather than TCP.

While TCP is a reliable, ordered, and loss-correction protocol, UDP, on the other hand, provides no guarantees on the reliability or the order of the data. Such an approach results in inherently faster delivery. The QUIC protocol re-implements the reliable delivery capabilities of the TCP transport layer while offering reduced latency, and additional optimisations [97].

First, QUIC was deployed internally at Google in the Chromium project and within the company network. At the time, it was designed to be a general-purpose protocol, i.e., not mainly focused on a particular layer or application. The application layer logic was later decoupled, removed from the main project and presented as a separate draft for the HTTP protocol [79]. This work would later become version 3 of the HTTP protocol (HTTP/3) (section 2.2).

In 2016, the now transport-layer protocol was submitted to the IETF for standardization [38, 50]. It would take over 30 drafts and five years to be fully standardised and released as an RFC [45].

2.2 Deployment of QUIC

During the years of development, as each new specification was released, new third-party tools supporting the protocol appeared [72]. As the protocol was standardised, these implementations eventually converged. However, some parts of the protocol were deliberately written not to be restrictive. Indeed, not all QUIC features are mandatory. Third-party vendors can choose to provide protocol implementations with a wide range of features while maintaining the same core characteristics and support. Moreover, being a userspace protocol, it was easier for companies and developers alike to write each own's implementation of the protocol using different programming languages and libraries.

The first non-Google third party to provide a QUIC implementation was Facebook [21]. Its project — dubbed "MVFST" and open-sourced in 2019 — provided several features on top of the QUIC protocol, such as support for zero-downtime restarts, full 0-RTT support, and multithreading [100]. MVFST proved popular within Facebook. By October 2020, over 75% of Facebook's internal traffic used QUIC [13]. More companies started adopting QUIC as its standardisation was nearing [5], and a plethora of open-source implementations started appearing:

- Quiche [10], by Cloudflare;
- MSQUIC [60], by Microsoft, with greater support for the Windows platform;
- picoquic, [71], a minimal implementation of the protocol;
- aioquic, [1], a minimal Python library.

Some researchers went further, proposing flexible implementations such as pQUIC [14, 15], an extension of QUIC which supports the injection of "plugins", or small pieces of code which are executed within the protocol context. Other work focused instead of extending QUIC's capabilities, such as managing the ACK frequency [43], allowing it to unreliably send data [68], protocol tunneling [69], and more.

Finally, in 2022, HTTP/3 was finalised and released as RFC 9114 [6], with the headline feature replacing TCP with QUIC. This announcement served as a springboard to the mass adoption of the protocol [19]. By July 2022, HTTP/3 was globally used by over 25% of all websites [94], with almost all mainstream browsers — bar Safari — fully supporting HTTP/3 [67]. With such numbers at hand, it became clear how the QUIC protocol could now be considered a mainstream technology.

2.3 The challenges of QUIC monitoring

As Internet protocols increase in popularity and become widely used by applications, so does the need to be able to monitor them closely. No matter where or when performance must be assessed 24/7 to ensure that systems keep performing optimally. Furthermore, attacks must be detected and possibly mitigated. For example, in data centres, tenants usually negotiate a Service Level Agreement (SLA) with their customers. Thus, tenants are then required to monitor the performance of their systems and ensure that they remain within the SLA, avoiding possible disruptions and consequent monetary losses.

Tools wishing to monitor a particular network rely on the fact that most protocols have a very precise and straightforward wire image[1]. Encryption techniques reduce such images, although systems capable of decrypting communications before the endpoints are widely available [95]. Historically, transport protocols have had either small, compact headers (such as TCP and UDP), while application protocols were text-based with a simple syntax, such as HTTP. This approach eased parsing immensely: network monitoring tools could efficiently manage network traffic and detect issues.

QUIC was designed from the start to be both encrypted and with an extremely small — if not close to zero — wire image [12]. This approach poses several new challenges for tools wishing to monitor the protocol with the same accuracy and efficiency as before. Although fingerprinting of the protocol has become easier [29, 82, 104], deep network monitoring of QUIC traffic has so far proven very challenging [16].

To begin with, QUIC has arguably more complex semantics. While for users, the switch from HTTP/2 to HTTP/3 is seamless [40, 75, 76], network engineers and developers will quickly notice that the protocol is much more difficult to parse efficiently.

TCP used a single, compact header with some features (such as acknowledgements, explicit congestion notifications, and more) provided as bit flags, the source and destination port provided at the start of the header, and the other fields being mandatory and fixed size except for the options. On the other hand, QUIC uses two distinct types of headers. Long headers, sent at the start of each new connection, contain unmissable information. If not correctly parsed and the information saved, further packets cannot be matched to the flow and information is lost. This is because short headers, which are used for packets sent over an existing connection, intentionally omit some fields to save bytes. Details on what fields are omitted can be found in sections 3.4.1 and 3.4.2.

To worsen things, the size of these headers varies greatly: endpoints can choose connection IDs of different sizes or omit them directly (section 3.3.1). In addition, endpoints must be able to correctly map incoming connection IDs — whose size is not provided in short headers — to the proper network flow. If no connection IDs are provided, endpoints must fall back to classic IP addresses for mapping, increasing the amount of information that must be saved. Finally, the two types of headers map to five different protocol types plus a particular case for connection termination. This range of possibilities further complicates the code implementation.

Second, QUIC swaps the classic TCP payload for a complex system based on *frames*, which are designed as packets within packets and carried in the packet's payload section. QUIC ships with more than thirty types of frames, each with its own structure and semantics. Data is provided using `STREAM` frames; acknowledgements are carried with `ACK` frames; connection signals are carried with several types of frames; and more. All these different frame types require a significant amount of code to parse and handle, given that the payload can contain an arbitrary amount of them. Moreover, the sequential nature of the frame means that each packet has to be processed in its entirety. Entirely unpacking and parsing the payload may pose privacy concerns, depending on where the system is running.

Third, QUIC uses another abstraction for exchanging data, known as the *stream*. QUIC allows multiplexing of an arbitrary amount of data flows into a single connection, requiring little to no intervention from the application layer. `STREAM` frames carry offset numbers, can be of arbitrary length, can be acknowledged out of order, and are divided into four number spaces depending on their characteristics (section 3.2). Endpoints are expected to track which streams are open, which are closed, and what data has been sent and received on each stream. Again, this requires a significant amount of code and state to manage.

Monitoring QUIC does not only require maintaining a tremendous amount of state for each connection. Several other features of QUIC actively or inadvertently hinder tools wishing to monitor the protocol:

- headers are encrypted with keys derived from the TLS handshake;
- connection IDs can change, or new ones can be generated, necessitating tracking of each set of connection IDs for each endpoint;

[1]The *wire image* of a network protocol is the view of the protocol as observed by an entity not participating in the communication [90].

- special encodings are used for integers and acknowledgement numbers, which require additional code for parsing.

Moreover, QUIC is a protocol designed to be easily updated. The current specification of QUIC from RFC 9000 [45] is defined as version `0x00000001`. However, the actual "version-independent" characteristics of QUIC, specified in a different RFC [84], are a small subset of the ones provided in the current version. Mechanisms to negotiate the version are still in development [18]. However, tools wishing to monitor QUIC will soon need to keep track of the version for each connection flow, possibly using different parsing mechanisms. Moreover, such a versioning mechanism could allow the appearance of "rogue" versions, which could be used to bypass network monitoring tools.

The nature of QUIC itself exasperates all these challenges. Indeed, QUIC was designed from the start to be a userspace protocol. Usually, the operating system's network stack would parse all network layers to the transport layer and pass the data to the application layer at the defined port. This process was designed to be highly efficient but hard to update and customise. On the other hand, QUIC — being delivered on UDP — requires a userspace application to manage the traffic.

Finally, as mentioned in section 2.2, many QUIC implementations are available, each with different performance, quirks, and a different set of capabilities. Developers benefit from this, having maximum control over their choices. However, monitoring applications are hindered even more, as they might need to be aware of each implementation's details to be fully able to parse the protocol.

2.4 Related work

From a technical standpoint, network monitoring can be divided into two categories:

- **performance** monitoring, which looks for problems caused by network congestion, latency, crashes, and other network errors;
- **intrusion** monitoring, which looks for potential malicious attacks on the network.

Specialised tools exist for both types. The latter category is efficiently addressed by intrusion prevention systems (IPS) and intrusion detection systems (IDS). Such systems analyse traffic at a flow rate as packets traverse them and can detect attacks and mitigate them (IPS) or emit an alert (IDS). They are usually deployed well before the endpoints, in such a position that they can detect an attack as soon as possible. [93].

On the other hand, performance monitoring tools track metrics such as the round trip time (RTT), the throughput, the packet loss, and more. They have been extensively studied in the past few years. State-of-the-art switches supporting terabits of traffic per second have been released. They often support domain-specific languages, which allow the separation of the data plane from the control plane and the implementation of complex traffic shaping and policing. Nevertheless, commodity hardware has come to support such languages in recent years, revolutionising the monitoring landscape. Query-based systems have appeared, allowing tenants to perform complex operations on network flows [4, 37]. Other work has instead focused on monitoring TCP failures in the data plane [33, 39].

QUIC is in relative infancy protocol and has yet to receive significant attention from the monitoring perspective. Unlike TCP, it has been described as relatively datacenter-unfriendly. Most studies focused on other aspects of the protocol; mainly, the performance gains when tied with HTTP/3 [61, 64, 66, 92, 102]. Others performed comparative studies between endpoints [103]. Nevertheless, some preliminary work on monitor-oriented aspects of the protocol has been published, both regarding firewall compatibility [32] and NIC offloading [101].

During the years between its release and the standardisation, several studies have been performed Internet-wide, collecting data on which versions were used and their share of the total traffic [11]. Other work instead focused on how each website implemented QUIC in practice by analysing the endpoints themselves and looking for their capabilities [30].

Other work instead focused on *qlog*, a parser-friendly JSON format for exporting QUIC logs and *qvis*, a visualization tool for such logs [56–58]. Finally, work has been carried out on the spin bit, such

as implementing tracking of said bit on P4 hardware [49] or changing the algorithm used for the spin bit, switching to implementations with greater accuracy [80, 81].

2.5 Contributions

This thesis aims to highlight the opportunities, challenges, and potential solutions for monitoring QUIC within an end-host in a userspace setting. We believe it is a matter of time before QUIC becomes the most popular network protocol on the Internet. However, its unique properties make it challenging to build monitoring tools for it. The state-of-the-art is currently very limited: it is not clear whether this is because of QUIC's relative youth or it is due to the difficulty in monitoring it. We believe this study could be an essential building block for future work to address the challenges posed to the monitoring community by QUIC.

A proof-of-concept implementation of such a monitoring system is provided, written in Python. This application is divided into two parts. The first part concerns the acquisition of packets from a network interface and uses widely used solutions. The second part focuses on parsing such packets, retrieving relevant information, and finally emitting alerts when it meets certain conditions. Users can granularly control the amount of parsing done on the packets and the amount of data collected. Users also choose which attacks they want to monitor in a query-based style.

The program is tested against a series of real-world use cases that feature possible protocol attacks. The monitor's performance is then analysed and compared between different settings. The results demonstrate the capabilities of the system but also the trades between accuracy and performance that must be considered when building such a system.

3
Background

This chapter provides a theoretical background on the topics explored in this thesis. The QUIC protocol is presented, along with the technical details necessary to understand the later sections of this thesis.

3.1 QUIC

QUIC (section 2.1) is a secure general-purpose transport layer protocol. Jim Roskind first designed it at Google in 2012. It was later deployed and proposed as a standard in the IETF in 2015. After a lengthy development, it was finally standardised by IETF as RFC 9000 in 2021. [45].

The QUIC protocol was explicitly designed to address TCP's shortcomings while allowing for greater performance, efficiency, and flexibility. Its headline features include:

- **Easy deployment**: despite being treated as a transport layer protocol, it actually runs over another transport protocol, the User Datagram Protocol (UDP). Indeed, QUIC has been designed from the ground up as a userspace protocol. As a result, middleboxes do not require updates to support it, nor do they even need to be aware of it. To a non-QUIC-aware machine, QUIC traffic is indistinguishable from any other UDP traffic. A comparison between the classic TCP/IP stack and the "QUIC" stack is provided in table 3.1.
- **Header and payload protection**: QUIC packets feature both payload and header protection. The former is achieved with the use of Transport Layer Security (TLS) [73], which is deeply integrated within the QUIC protocol [87]. Indeed, unlike HyperText Transfer Protocol Secure (HTTPS) — which embeds HTTP packets in TLS packets — QUIC works hand in hand with TLS, embedding the TLS handshake and options within the payload of QUIC packets. To further strengthen the protocol, headers are protected with a key derived from the protected packet [72].
- **Reduced RTTs**: QUIC connections are first established with a handshake, which allows the negotiation of cryptographic and transport parameters. Unlike TCP's three-way handshake, QUIC features a cryptographic exchange that allows data to be sent as soon as possible. This effectively reduces the required round-trip times (RTTs) from 2 to 1. Moreover, it includes an option to send data immediately at the start of the connection, called 0-RTT [34].
- **Stream multiplexing**: QUIC packet payloads contain frames, a logical structure that contains control information and application data. Frames containing such data are called stream frames. In QUIC, a stream is an ordered flow of bytes. It can be either unidirectional or bidirectional. Either the client or the server may initiate it. QUIC implements reliable delivery and congestion control for streams, effectively taking care of UDP's inherent unreliability. Additionally, QUIC includes an algorithm for detecting and recovering from the loss of data [44].
- **Connection migration**: QUIC connections are not strictly bound to a single network path. Unlike TCP/IP's classic "5-tuple" approach, QUIC flows are identified by a pair of variable length identifiers called connection IDs. They allow connections to continue after changes in network topology or address mappings.
- **Extensibility**: QUIC was built with versioning in mind. The header has a version field, allowing faster upgrades and version negotiation mechanisms. Moreover, the protocol supports extensions that can be easily negotiated during the initial handshake.

The following sections focus on the technical details of QUIC. First, Section 3.2 focuses on streams, the basic abstraction provided by QUIC to transmit data. Next, section 3.3 explains how QUIC

<table>
<tr><th>Protocol</th><th>Layer</th><th>Protocol</th><th>Layer</th></tr>
<tr><td>HTTP/1.1
HTTP/2</td><td rowspan="2">Application (L7)</td><td>HTTP/3</td><td>Application (L7)</td></tr>
<tr><td>TLS</td><td>“Security” (L4/L7)</td><td rowspan="2">QUIC
(with TLS)</td><td rowspan="2">“Transport” (L4/L7)</td></tr>
<tr><td>TCP</td><td rowspan="2">Transport (L4)</td></tr>
<tr><td></td><td></td><td>UDP</td><td>Transport (L4)</td></tr>
<tr><td>IP</td><td>Network (L3)</td><td>IP</td><td>Network (L3)</td></tr>
<tr><td>Ethernet</td><td>Data Link (L2)</td><td>Ethernet</td><td>Data Link (L2)</td></tr>
<tr><td>...</td><td>Physical (L1)</td><td>...</td><td>Physical (L1)</td></tr>
</table>

Table 3.1: The TCP/IP stack and QUIC's pseudo-stack.

connections are established and how they are managed and briefly talks about header and payload protection and TLS integration. Finally, section 3.4 focuses on the QUIC packet format.

3.2 Streams

QUIC uses streams as its core abstraction to transmit data. A stream is an ordered flow of bytes from one endpoint to another. Streams are managed with `STREAM` frames, which are sent in the payload of QUIC packets and contain all the necessary information to initiate the transfer, interpret the data, and close the stream. Any endpoint can open an arbitrary number of streams and use them at it pleases, possibly interleaving them [47].

Streams can be either unidirectional or bidirectional. Either the server or the client may initiate them. To distinguish between these types, QUIC reserves the two least significant bits of the stream ID (a 62-bit integer). The first bit is 0 for streams initiated by the client and 1 for streams initiated by the server. The first bit is 0 for bidirectional streams and 1 for unidirectional streams. Therefore, streams have four different number spaces, each one starting from:

- `0x00` (..., `0x04`, `0x08`, ...) for client-initiated, bidirectional;
- `0x01` (..., `0x05`, `0x09`, ...) for server-initiated, bidirectional;
- `0x02` (..., `0x06`, `0x0A`, ...) for client-initiated, unidirectional;
- `0x03` (..., `0x07`, `0x0B`, ...) for server-initiated, unidirectional.

Stream IDs cannot be repeated during a connection and are globally unique. An endpoint can choose to open a stream with an arbitrary stream ID, as long as it is within a range negotiated during the handshake. However, in doing so, it commits to open all streams with a lower ID than the one it chooses. For example, on a new connection, opening stream no. `4000000` opens 1 million and 1 client-initiated bidirectional streams.

Once a stream is open, endpoints transmit data using `STREAM` frames. The data is split into an arbitrary number of segments by the sender. The offsets to each segment are included in the frame to allow the receiver to reassemble the data.

Additionally, QUIC includes logic for handling stream state transitions and flow control. Indeed, other frames can be sent to control the flow of data, such as `STREAM_DATA_BLOCKED` or `RESET_STREAM` frames. More details can be found in Section 3 of the QUIC RFC [45].

3.3 Connections

Two QUIC-equipped endpoints must first establish a connection before actually exchanging data. As mentioned before, QUIC is not strictly bound to a single network path and discards the classic 5-tuple used to identify a connection by TCP/IP. Instead, QUIC uses a pair of variable-length identifiers called

connection IDs. They allow connections to continue after changes in network topology or address mappings.

Connections are established with a handshake reminiscent of TCP/IP's 3-way handshake but with more flexibility and the possibility of exchanging data immediately (0-RTT) or after the first exchange (1-RTT). QUIC uses TLS to encrypt the connection and negotiate transport parameters during the handshake.

When there is a sudden change in the network topology, QUIC can migrate connections to new network paths and validate them to determine if the migration allows the peer to continue exchanging data. However, connection migration is outside the scope of this document and is described in more detail in [45].

Finally, connections can be terminated, either gracefully or abruptly, similarly to TCP.

3.3.1 Connection IDs

Each connection has a source and destination connection ID. They vary between 0 and 160 bits in length.[1]. Their length depends on the implementation and does not have to be the same for both endpoints.

When a connection is about to be established, and the client sends the first packet, it embeds its source connection ID in the packet, i.e., the ID at which it wishes to receive data. Some implementations also embed the destination connection ID in the first packet, but the endpoint can choose to accept it or discard it by sending a Retry packet. Connection IDs should be unpredictable and unique, generated as randomly as possible by endpoints, to avoid tracking by middleboxes and potential attackers.

Once a connection is established, the client and the server can choose to issue additional connection IDs for the remainder of the connection within the limits negotiated during the handshake. Using `NEW_CONNECTION_ID` and `RETIRE_CONNECTION_ID` frames, endpoints can exchange new connection IDs or retire them. Once this is done, both endpoints can freely use new or old IDs altogether as they wish.

When a new connection ID is issued, an endpoint includes in the frame or as a transport parameter a unique token called *stateless reset token*. This 16-byte token allows the other endpoint to reset the connection unilaterally. Stateless Reset packets are a particular type of packets that are sent when the connection has become unmanageable or the endpoint cannot parse received packets. Their structure is deliberately chosen to be difficult to distinguish from a classic short header packet. It embeds the stateless reset token at the end, preceded by an amount of random padding. When an endpoint receives such a packet and detects the stateless reset token, it resets the connection. Stateless reset tokens are cryptographically protected, like the rest of the protocol, avoiding misuse by attackers and middleboxes. However, as random as they can be, middleboxes and attackers can still try to guess them.

3.3.2 Handshake

QUIC connections are established with a handshake, which deliberately combines cryptographic information, transport parameters, and possibly data to minimise the establishment time. In the current version of QUIC, TLS is used as the underlying cryptographic protocol, and its messages are carried in `CRYPTO` frames.

The actual packet exchange comprises a pair of Initial packets, followed by a pair of Handshake packets. Once the Initial packets are exchanged, the data can be sent. Figure 3.1 illustrates the handshake.

The diagram shows multiple packets being sent in a specific direction at once. QUIC supports *packet coalescing*, i.e., inserting multiple QUIC packets into a single UDP datagram. Therefore, an endpoint can generate the keys for sending 0-RTT data and then send both an Initial packet and a 0-RTT packet clamped together in a single datagram.

During the handshake, QUIC exchanges transport parameters in addition to the cryptographic

[1] QUIC allows zero-length connection IDs and has specific methods for handling them, but this thesis will not discuss that. Moreover, the QUIC RFC clearly states that connections made with zero-length connection IDs are considered fragile and should be avoided if possible.

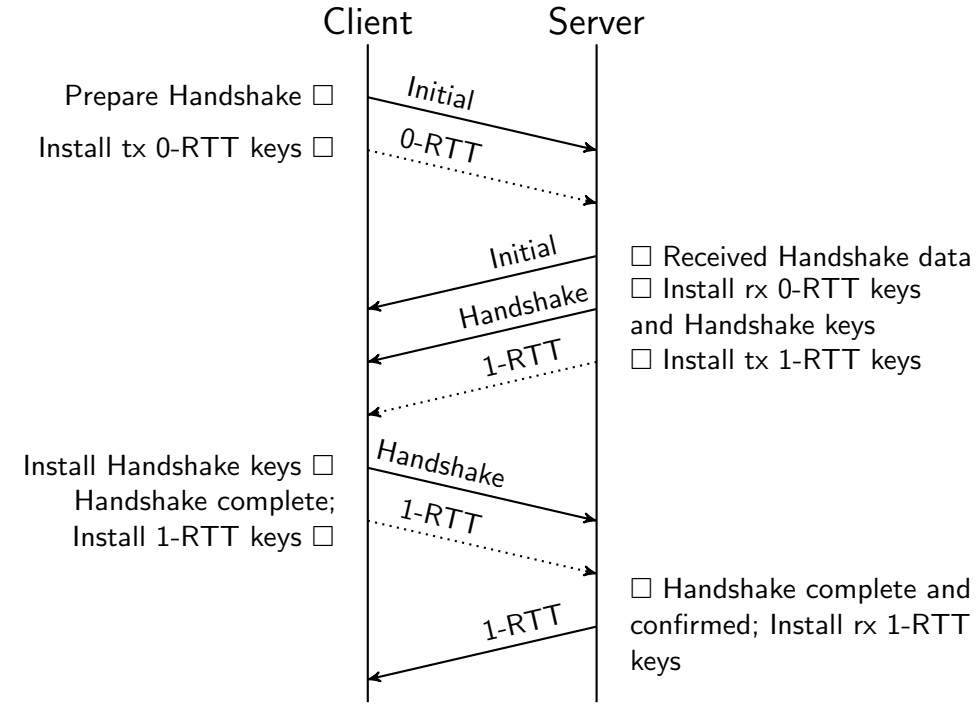


Figure 3.1: The QUIC handshake.

information. Transport parameters are declarations by the endpoint about some aspect of the protocol. The other endpoint must respect their choice but may, in turn, also choose a different value for his parameter. The parameters exchanged are several; some of them include `initial_max_data` or `active_connection_id_limit`. The former establishes the maximum number of bytes sent in the connection, and the latter limits the number of concurrent connection IDs that can be used. Section 18 of the RFC [45] provides a comprehensive list of transport parameters.

Further details on the handshake, including how TLS is actually implemented and how are packets encrypted, are described in [17, 45, 73, 83, 87] and are shown in figure 3.2.

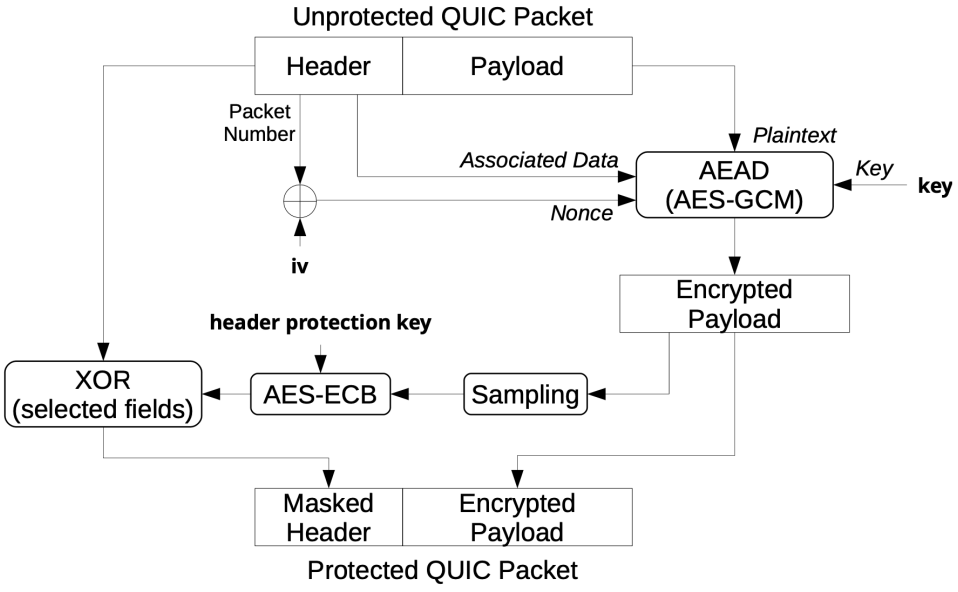


Figure 3.2: How QUIC packets are encrypted and protected.

There is an important aspect to be noted in the diagram. Once the keys are installed, and the endpoints begin exchanging data on 1-RTT packets, the payload is augmented at the end by 16 bytes.

This step is done in order to include the AEAD tag. The endpoint removes the tag when receiving and decrypting the packet, effectively reducing the available payload size.

In addition to encryption of the payload, QUIC provides a mechanism called header protection. Header protection protects part of the QUIC headers, like the packet number, by encrypting them with a key derived from the packet protection key and IV. Therefore, endpoints assemble the packets, encrypt the payload, and finally encrypt the headers.

This protection applies to the least significant bits of the first byte, plus the Packet Number field. The four least significant bits of the first byte are protected for packets with long headers; the five least significant bits of the first byte are protected for packets with short headers. For both header forms, this covers the reserved bits and the Packet Number Length field; the Key Phase bit is also protected for packets with a short header.

Finally, the process does not apply to Retry or Version Negotiation packets, which do not contain a protected payload or any of the fields that are protected by this process.

3.4 Packets

At its core, QUIC exchanges data between endpoints using packets. Such packets, as mentioned before, travel on UDP datagrams and are confidentiality and integrity protected. QUIC distinguishes between four main types of packets, all composed of a header and a payload. Such packets are:

- **Initial** packets, used at the very start of the connection handshake. It carries the first `CRYPTO` frames sent by the client and server to perform key exchange, and it carries `ACK` frames in either direction.
- **0-RTT** packets, used to send data prior to handshake completion;
- **Handshake** packets, similar to Initial packets and also used during the start of a connection; their difference is explored in section 3.3.2.
- **Retry** packets, used for validating endpoints and retrying a handshake;
- **1-RTT** packets, used to send data after handshake completion.

The first four packets of the list use the so-called *long header* format, while 1-RTT packets use the *short header*. The latter is designed to be as small as possible. As a result, it intentionally omits some information expected to have already been memorised by endpoints.

Furthermore, 0-RTT and 1-RTT packets share their packet number space, i.e. data sent in either of these packets are ordered and acknowledged together. On the other hand, retry packets have no packet number space since they can be sent at most once. Finally, Initial and Handshake packets have distinct packet number spaces.

In addition to the aforementioned packet types, QUIC also supports Version Negotiation packets and Stateless Reset packets. The former uses the long header format but is unencrypted and does not have packet numbering. The latter is used to signal a connection reset and is explained in section 3.3.1.

3.4.1 Long header format

Figure 3.3 shows the long header format:

- the **F** bit indicates the header form, and is set to `1` for long headers and `0` for short headers;
- the **B** bit is the fixed bit, always set to `1` in this version of the protocol;
- the **TY** (Type) field contains the type of the packet, which is
  - `0` for Initial,
  - `1` for 0-RTT,

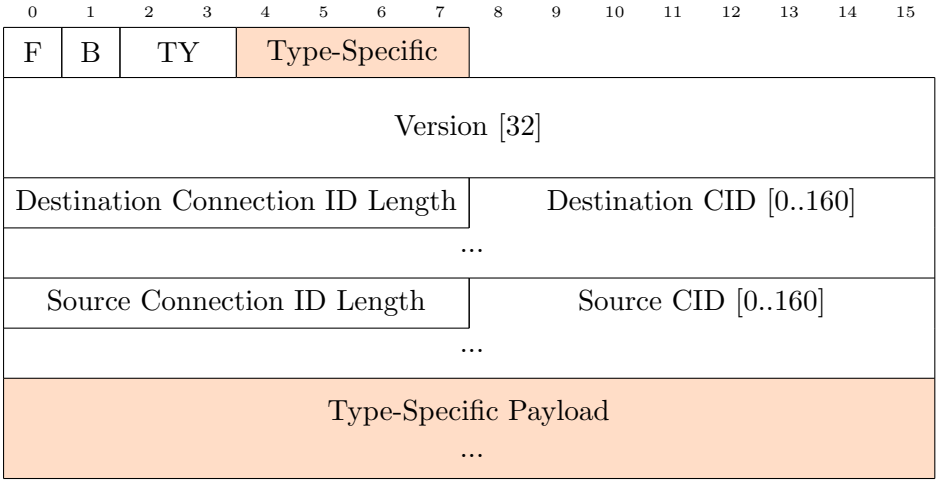


Figure 3.3: The long header format of a QUIC packet. Coloured fields are protected.

  - `2` for Handshake,
  - `3` for Retry;

- **Type Specific** bits are used by Initial, 0-RTT, and Handshake packets to specify the length of the packet number, which is later included in the payload. They are always protected.

The header then presents a 32-bit long version field. QUIC uses version numbers to indicate the protocol version and includes a version negotiation mechanism that is being actively developed [18]. For the purposes of this document, we focus on version `0x00000001`, which is the IETF-standardised version of QUIC.

Then, the header contains the destination and source connection IDs, whose role is explained in section 3.3.1. First, their length is provided as an unsigned 8-bit integer. Then, the connection IDs are presented as a variable-length string of bytes. This is done first for the destination ID, and then for the source ID.

Finally, the payload of the packet is also type-specific. At the start of this payload, `Initial`, `0-RTT` and `Handshake` include a field with the length of the data. This is followed by the packet number — whose length was previously specified in the *Type Specific* field — and finally, the actual data itself. Retry packets do not have a packet numbering space and contain different fields. More information on the full structure of each long header packet can be found in the RFC.

3.4.2 Short header format

On the other hand, figure 3.4 shows the short header format:

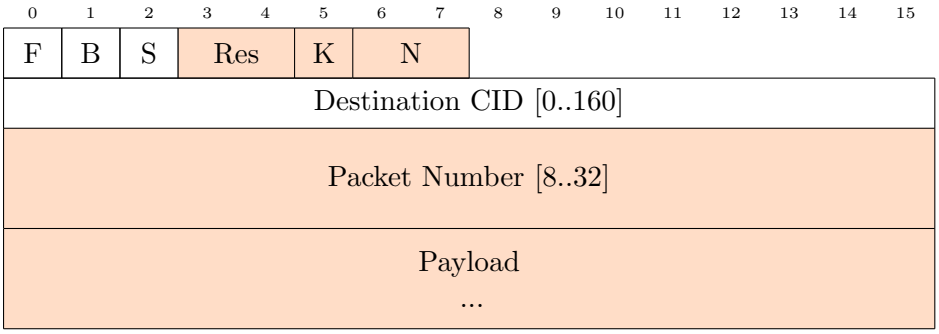


Figure 3.4: The short header format of a QUIC packet. Coloured fields are protected.

- the **F** bit indicates the header form, set to `1` for short headers;
- the **B** bit is the fixed bit;
- the **S** bit is the spin bit, whose purpose is explored in section 3.4.4;
- the next two bits are reserved;
- the **K** bit is the key phase bit, outside the scope of this document;
- the next two bits encode the packet number length.

The header then presents the destination connection ID field, which is intentionally not preceded by its length, as QUIC expects endpoints to have already memorised this information. The source connection ID is also omitted. Finally, 8 to 32 bits are used to encode the payload number.

3.4.3 Frames

Every QUIC packet contains in its payload one or more frames. Version Negotiation, Stateless Reset, and Retry packets do not contain frames. The payload of a packet cannot be zero bytes long: QUIC instead allows filling the payload with dummy data, using `PADDING` frames (which are 0-filled bytes). A sender can minimise bandwidth by including as many frames as possible in each QUIC packet, although this is optional.

QUIC supports many frame types, ranging from flow control to connection management, connection ID issuance and retirement, and many others. Each frame type is identified by a variable-length frame type field — encoded as in section 3.4.5 — which is the first byte of the frame. Following this, the frame itself is encoded with a syntax specific to that particular frame type. Therefore, parsers will first need to determine the frame type and then parse the frame, which will be of a variable length. Finally, it will continue parsing the following frames, if any.

This document will focus on the most important ones:

- `STREAM` frames: shown in figure 3.5. The first field in `STREAM` frames takes the form `0b00001OLF`, where:
  - `O` is the offset bit, which is set to `1` if the frame is not the first in the packet; if set, then the offset field will be encoded after the stream ID;
  - `L` is the length bit, which is set to `1` if there is a length field following the stream ID (or offset, if `O` is set); if not set, the length of the frame will span the entire packet;
  - `F` is the fin bit, which is set to `1` if this is the last frame in the packet.

  `STREAM` frames are then followed by the stream ID, as described in section 3.2, the aforementioned optional offset and length fields, which are variable-length[2], and finally the frame payload.
- `ACK` frames: unlike TCP, which relied on a single bit in the header for signalling the acknowledgement of packets, QUIC acknowledgements trade flexibility for complexity. Figure 3.6 shows the structure of an `ACK` frame. Frame fields comprise:
  - the `E` bit, set to `1` if the frame contains the cumulative count of QUIC packets with associated ECN marks received during the connection;
  - the Largest Acknowledged field, which indicates the largest packet number that has been acknowledged;
  - the ACK Delay field, which indicates the number of microseconds that the sender should wait before acknowledging packets;
  - the ACK Range Count field, which indicates the number of packet ranges being acknowledged (at least one).

[2]Fields encoded as variable length integers are depicted as having length "(i)" in diagrams.

The frame contains the first ACK range, followed by zero or more ranges as indicated by the ACK Range Count field. An ACK range is a special substructure of the `ACK` frame, which contains first a variable-length integer indicating a contiguous range of unacknowledged packets and then a variable-length integer indicating another contiguous range of acknowledged packets. ACK ranges use an articulated encoding scheme, which is thoroughly explained in section 19.3.1 of [45]. Finally, the frame contains some fields for ECN feedback, which are not explored in this document.

- `NEW_CONNECTION_ID` and `RETIRE_CONNECTION_ID` frames: these two frames help in the issuance and retirement of connection IDs as explained in section 3.3.1. `NEW_CONNECTION_ID` frames contain the connection ID to be issued along with a unique, increasing sequence number. Sequence numbers are assigned to connection IDs by the endpoint and are used as a quick way to identify them. Then, the frame may optionally contain a field with a sequence number of a past connection ID to be retired. Finally, the stateless reset token tied to the connection ID is included. On the other hand, `RETIRE_CONNECTION_ID` frames contain the sequence number of the connection ID to be retired.

- `CONNECTION_CLOSE` frames are sent by the endpoint to signal that the connection is no longer active and will be closed. The least significant bit of the frame type distinguishes between regular `CONNECTION_CLOSE` frames, signalling errors at the transport layer, and `APPLICATION_CLOSE`[3], signalling errors at the application layer. The frame contains a variable-length integer indicating the error code and then an optional frame type indicating the offending frame. Finally, a textual reason for the error is provided, with first a length field followed by the reason text itself.

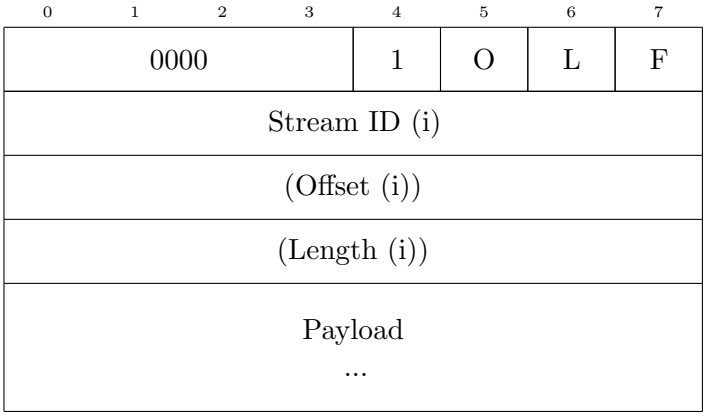


Figure 3.5: The format of a `STREAM` frame.

3.4.4 Spin bit and RTT estimation

As mentioned in section 3.4.2 and shown in figure 3.4, short header QUIC packets contain a spin bit, which is used to assist middleboxes in estimating the round-trip time. The spin bit is optional [45, 89], and is a feature specifically tied to version 1 of QUIC. Thus, it may not appear in future versions [84].

The latency spin bit enables latency monitoring from observation points on the network path throughout a connection. Each endpoint maintains a spin value for each connection, 0 or 1. This value is initialised to 0 for each endpoint at the creation of a connection. When a short header packet is sent, the currently stored value is sent along with the packet. Moreover, endpoints must remember the highest packet number sent by their peer.

When a short header packet is received, the endpoint first checks if the packet increments the higher packet number seen. If so,

[3]The `APPLICATION_CLOSE` moniker is not found in the RFC and is used here only for simplicity.

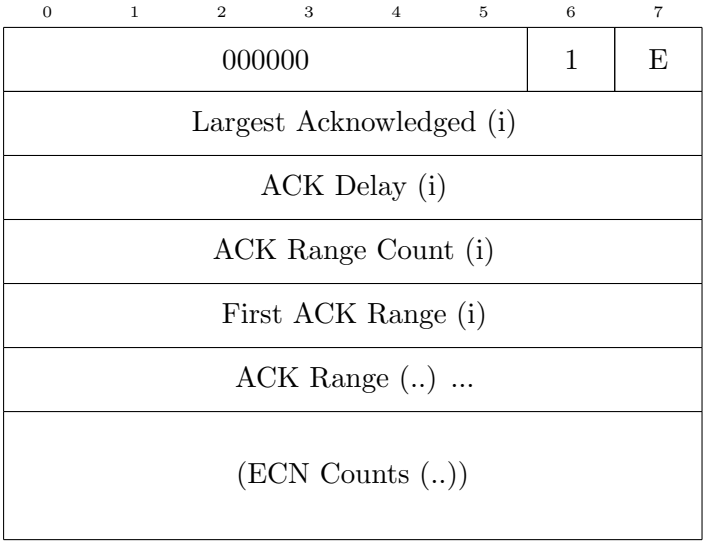


Figure 3.6: The format of an `ACK` frame.

- if the endpoint is a client, it sets the stored spin value to the opposite of the received spin value;
- if the endpoint is the server, it sets the stored spin value to the same value as the received spin value.

This procedure causes the spin bit to change its value in each direction once per round trip. Observation points can estimate the network latency by observing these changes in the latency spin bit [91].

The spin bit provides a quick and inexpensive way of monitoring the network round trip time, being easy to monitor at both endpoints and middleboxes with relatively few lines of code [48], but is prone to errors, especially in the case of network congestion. Some variants to the spin bit have been proposed [80, 81], but they remain outside the scope of this document, and at the current time, they remain as proposals.

3.4.5 Variable-length integer encoding

QUIC is a very complex protocol, mixing classic bit fields that can be easily parsed with an `OR` operator, with complex semantics requiring several processing steps.

One of them is the so-called *variable-length integer encoding*. Variable-length integer encoding reserves the two most significant bits of the first byte to encode the base-2 logarithm of the integer encoding length in bytes. The remaining bytes are used to encode the integer. Therefore, *varint* fields encode the type of values listed in table 3.2.

Bits	Length	Usable bits	Range
00	1	6	$0 - 2^6 - 1$
01	2	14	$0 - 2^{14} - 1$
10	4	30	$0 - 2^{30} - 1$
11	8	62	$0 - 2^{62} - 1$

Table 3.2: The encoding of a variable-length integer.

4
Implementation

This chapter proposes a simple implementation of a QUIC monitor application called **QUIC Watchful Information Collector** (QWIC).

The application is logically divided into two parts. The first part uses state-of-the-art libraries to sniff incoming traffic over a network interface, filter the required QUIC traffic, and convert the obtained packets into a byte array representation. The second part — written in pure Python — receives such packets as input, fully decodes and parses them, and dynamically extracts information relevant to the given configuration. Finally, the application can be configured to emit alerts when certain events occur.

A proof-of-concept of this application is available on GitHub at the link `https://github.com/mfranzil/qwic/`. The core of the application was developed starting from eBPF open-source projects by Simone Magnani [53–55].

4.1 Assumptions

In this section, we present some assumptions about the application.

First, the application is designed to be used on a single machine, sniffing a single network interface. This setup allows the application to ideally run both on single-host machines or within the hypervisor of a machine hosting multiple virtual machines, potentially intercepting all incoming and outgoing traffic.

As highlighted in the previous chapters, QUIC is a complex protocol with several corner cases and features that are hard to implement in practice in a monitor application. Moreover, the protocol is encrypted by default, with little to no intention by the IETF and vendors to provide a plaintext option for QUIC[1].

Thus, to avoid the need to reimplement the protocol and streamline the application fully, the following design choices were made:

- we assume that protocol communications are completely unencrypted. This assumption is reasonable, as network interface cards performing decryption are widely available. The extra 16 bytes appended to the payload are kept and completely ignored.

- coalesced packets are not supported. The application will assume that each incoming UDP datagram contains one and only one QUIC packet. This assumption makes tracking parsing times easier and avoids corner cases that might be hard to handle.

- Handshake, Initial and 0-RTT packets (with long headers) are used by the application only to derive the necessary state for managing the connections and are otherwise ignored as the TLS handshake does not affect the packets.

  As a result, all transport parameters exchanged during the handshake are not tracked. This is done to simplify and make the application faster, as it would otherwise spend a lot of CPU time on parsing and tracking the handshake content and managing the packet spaces of these packets. However, this means that the first stateless reset token for the first connection ID cannot be tracked, as it is included as a transport parameter.

  Retry packets, on the other hand, are correctly handled as they can cause the connection IDs to change.

[1] An attempt was made to provide such a version as an IETF draft, but it received little attention [2].

- we assume the connection IDs provided in packets are exactly 20 bytes long (160 bits). Currently, most implementations allow users to disable the randomisation of the length of the connection IDs. Saving the lengths of the connection IDs is, therefore, not needed. Moreover, headaches are avoided during the parsing of short headers, as provided connection IDs do not come with their length[2].

- we assume connection IDs are generated randomly. Hash map collisions are reasonably excluded, given the 160-bit space of the connection IDs. To save space further and avoid collisions, connections that have either been closed or remained silent for more than 30 seconds have their information removed from the program.

- we assume endpoints correctly handle the spin bit. Although it is not guaranteed to be set in future versions [84], and some extensions even allow its modification [85], we observed that most implementations set it.

- we focus the frame parsing on a small subset of frames, namely:

  - `PADDING` frames
  - `STREAM` frames
  - `ACK` frames
  - `CONNECTION_CLOSE` frames
  - `STREAM_DATA_BLOCKED`, `STREAMS_BLOCKED_BIDI`, `STREAMS_BLOCKED_UNI`
  - `NEW_CONNECTION_ID` and `RETIRE_CONNECTION_ID` frames

- advanced QUIC features such as connection migration and path validation are not supported. However, the mechanism which allows the issuance and retirement of new connection IDs is fully supported. Hash maps are correctly updated when this happens.

Finally, we assume all traffic is based on version `0x00000001`. As mentioned in section 2.3, QUIC is expected to evolve rapidly, and thus traffic in the future may use a version different from this one. However, for the immediate time being, most, if not all, QUIC traffic uses version 1, and thus the application will assume that all parsed traffic match this version.

Figure 4.1 provides a high-level visual representation of the system.

4.2 Packet acquisition

As mentioned before, the application is logically divided into two parts. This part is responsible for sniffing incoming traffic and converting it into a byte array representation. To do so, we experimented with various technologies with different tradeoffs and limitations.

In the final program, each of the following three solutions is available and can be easily selected as a command line parameter by the user (code 4.1). Either way, all three will emit a stream of `bytearrays`, which is then used as input to the second part of the application.

[2] A longest-prefix matching algorithm would have been needed to do so.

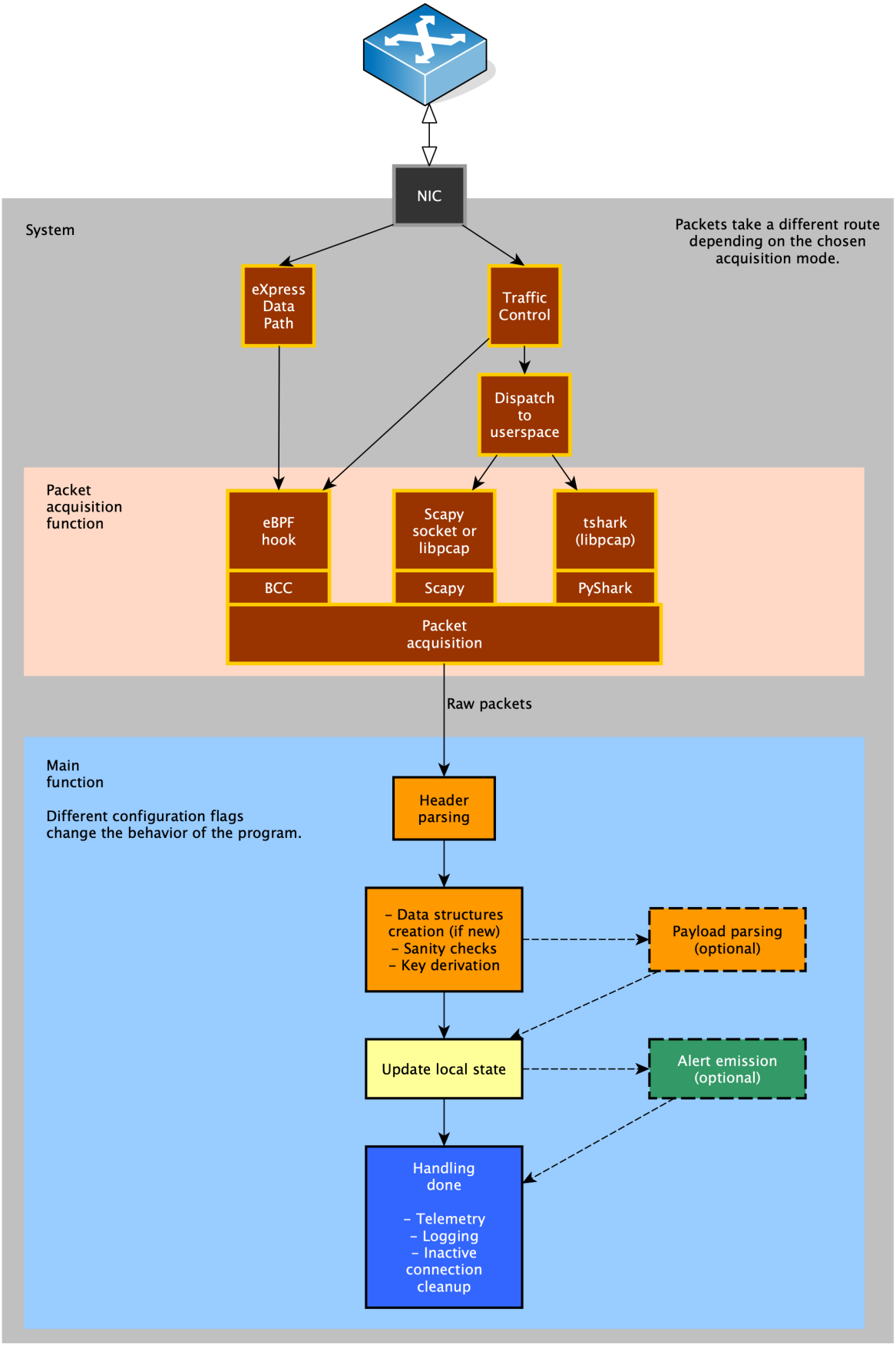


Figure 4.1: System architecture of the QWIC application.

```
if mode_str == "XDP":
    monitor_ebpf(
        mode=BPF.XDP,
        device=device,
        offload_device=None,
        ingress_fn_name="ingress_filter_xdp"
    )
elif mode_str == "TC":
    monitor_ebpf(
        mode=BPF.SCHED_CLS,
        device=device,
        offload_device=None,
        ingress_fn_name="ingress_filter_tc"
    )
elif mode_str == "PY":
    monitor_pyshark(device=device)
elif mode_str == "SC":
    monitor_scapy(device=device)
else:
    raise ArgumentError(f"Invalid mode: {mode_str}")
```

Code 4.1: Selection of the sniffing mode at the start of the program.

4.2.1 eBPF

The first solution we came up with involved the use of eBPF (extended Berkeley Packet Filters), a Linux kernel module that allows running userspace code in the operating system kernel [20, 27, 28]. These programs are written in C and compiled just in time. Such programs can be attached to tracepoints in system calls, sockets, or — in recent kernel versions — even to network datapaths. However, to ensure kernel integrity and prevent malicious programs from modifying the system, the programs are sandboxed by the kernel. Programs are run in a lightweight virtual machine and are checked against a verifier, which performs static analysis on the program.

eBPF is very powerful, but the documentation is not comprehensive and has a high entry barrier. To make it easier to use eBPF, we settled on using the BPF Compiler Collection (BCC) [42]. BCC is a set of tools built on top of eBPF, which s an easier-to-use C API, and a higher-level programming interface written in Python. It allows users to write programs which are then compiled on the fly to eBPF and whose output is piped to Python structures, ready to be used.

To begin with, we came up with a simple filter and parser which would filter out all QUIC traffic and parse some primary header fields. Code 4.2 shows how filters are attached to the network datapath and how data is then passed to userspace. In particular, we used both the Traffic Control (TC) and eXpress Data Path (XDP) datapaths to capture packets. The former is the classic Linux datapath used by packets, available in both ingress and egress [52], and the latter is a new, fast datapath that skips the network stack and directly manipulates the packet data [9, 31, 41].

```
int ingress_filter_tc(struct __sk_buff *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
    int res = udpfilter(data, data_end, TC_ACT_OK);
    if (res == -1) {
        int ingress_magic = INGRESS_MAGIC;
        skb_events.perf_submit_skb(ctx, ctx->len, &ingress_magic, sizeof(
                                        ingress_magic));
        res = TC_ACT_OK;
    }
    return res;
}
```

Code 4.2: A filter attached to the ingress datapath.

Code 4.3 shows the actual filter put in place, with the packet being parsed with pre-defined structs. In particular, flags present in the first QUIC byte are obtained using bit masks and bit shifts, which are very efficient operations.

```
static int udpfilter(void* data, void* data_end, __u8 pass_code) {
    // ...
    struct ethhdr *eth = data;
    if (data + sizeof(*eth) > data_end) {
        return pass_code;
    }
    // ...
    struct quichdr_init *quic = data + sizeof(*eth) + ip_header_len +
                                        udp_header_len;
    if ((void *) quic + sizeof(*quic) > data_end) {
        return pass_code;
    }
    if (! ((quic->flags & 0b01000000) >> 6)) {
        return pass_code;
    }
    if ((quic->flags & 0b10000000) >> 7) {
        struct quichdr_long *quic_long = data + sizeof(*eth) + ip_header_len +
                                        udp_header_len;
        if ((void *) quic_long + sizeof(*quic_long) > data_end) {
        // ...
        }
    // ...
    }
}
```

Code 4.3: The UDP filter used to match against packets.

After being partially parsed, the obtained packets are sent to the userspace for further (and deeper) analysis (section 4.3).

We obtained a very efficient solution, but flawed. First of all, we settled on using ring buffers, a data structure explicitly designed to transfer network packets to userspace [25, 26, 36].

However, as soon as the program was tested with high throughputs ($\geq$ 10Mbps), the ring buffer

would be unable to keep up, causing older packets to be dropped and never read by the program.

As the network flow was being read, we estimated that up to 80% of the traffic was being dropped[3]. To make things worse, packets overwriting each other would cause all sorts of problems once parsed. Very often, parts of bigger packets would remain in memory once a smaller packet was saved on top of them, tricking the system into thinking the packet was bigger than it was. To avoid wasting time, we chose to switch to a different approach.

4.2.2 Pyshark

To try to fix the issues mentioned in section 4.2.1, we switched to an implementation based on Pyshark [35]. Pyshark is a Python library that provides a high-level API for tshark [51], a command-line version of Wireshark that works much like tcpdump, but with greater flexibility.

```
cap = pyshark.LiveRingCapture(
    interface=device,
    bpf_filter="udp",
    use_json=True,
    use_ek=True,
    include_raw=True,
    ring_file_size=1024 * 20,
    num_ring_files=3
)

try:
    cap.apply_on_packets(packet_callback_pyshark)
except KeyboardInterrupt:
    exit(0)
```

Code 4.4: Call to `LiveRingCapture()`, used to start a live capture with several parameters.

Code 4.4 shows how the Pyshark LiveRingCapture is used to read packets from an interface. Once the capture is set up, Pyshark allows a function to be called as a callback each time a packet is captured, which in turn calls the parser function. With this setup, no packet was ever lost, although we detected some lag during high-throughput tests as the packets were being copied and analysed. However, a bug in Pyshark[4] affected the saving of the very last packet of each network flow. The bug was still open at the time of writing, so we had to use a workaround and switch to another solution.

4.2.3 Scapy

Finally, we settled on using Scapy [78], another packet sniffing and manipulation library. It is very similar to Pyshark, and we were able to use it in an almost plug-and-play fashion.

After packets are sniffed, they go through a final "middleware" function responsible for calling the parser, gathering the output data, and eventually printing it.

4.3 Packet manipulation

The second — and core — part of the program revolves around the parsing of the packets.

It is done by a parsing function, referred to *main function* from now on, which takes as input:

- a packet object, of type `bytearray`;

[3] eBPF BCC kindly informs the user when the ring buffer is full and takes note of the number of dropped packets.
[4] `https://github.com/KimiNewt/pyshark/issues/390`

- the timestamp of the packet when it was received, of type `float`;
- an optional boolean that indicates whether the packet is an ingress or egress packet, of type `bool`;
- an optional boolean that is set to `False` if the packet contains just the QUIC packet as extracted from the UDP payload or to `True` if it contains the full Ethernet frame.

With the last optional argument set, the parser function will fully parse the whole frame, deriving data from the Ethernet frame and the UDP datagram. This is the default behaviour, as all filtering solutions outlined in section 4.2 are set up to emit complete Ethernet frames.

Once the QUIC packet is available, the parser function proceeds with the header parsing, which is mandatory (section 4.3.1). This process is critical since it yields the core information about the QUIC packet: the connection IDs.

The connection IDs are used for identifying QUIC connections, and malformed or absent connection IDs will cause the parsing to abort. For managing data about QUIC connections, the application uses a global hashmap. Such hashmap is called `local_dictionary` and is updated every time a new packet is parsed. The map's key is the union of the source and destination IP addresses. They are put together as a tuple such that:

- if the numeric representation of the source connection ID is smaller than the numeric representation of the destination ID ($\text{connID}_s < \text{connID}_d$), then the key is the tuple $(\text{connID}_s, \text{connID}_d)$;
- else, the key is the tuple $(\text{connID}_d, \text{connID}_s)$.

To assist in mapping incoming packets to established flows, a support hashmap `connection_ids` is provided. This hashmap takes as a key a connection ID and as a value the tuple of the source and destination IP addresses. This map is updated every time a Retry packet is received or the set of active connection IDs changes. Code 4.5 shows how this is implemented in the application.

```
if quic_header['src_conn_id_int'] < quic_header['dest_conn_id_int']:
    key = (src_conn_id, dest_conn_id)
else:
    key = (dest_conn_id, src_conn_id)

if src_conn_id not in connection_ids:
    connection_ids[src_conn_id] = key
if dest_conn_id not in connection_ids:
    connection_ids[dest_conn_id] = key

if key in local_dictionary:
    local_state = local_dictionary[key]
else:
    # ...
```

Code 4.5: Deriving the key from the source and destination IDs.

Once the key is sorted out, the local state is either created or retrieved from the hashmap. The actual data saved in the local state depends on the flags set in the program (appendix A.3). Code 4.6 shows how the local state is created for a new flow. All the variables stored in the local state are described in section 4.4.

```
local_state = {
    "src_conn_id": src_conn_id,
    "dest_conn_id": dest_conn_id,
    'last_1rtt_seen': 0
}
# ...
if flags["TRACK_RTT"]:
    local_state['rttvar'] = 0
    local_state['smoothed_rtt'] = 0
    local_state['spin_bit'] = 0
    local_state['last_rtt_time'] = packet_time
# ...
if flags["TRACK_ACKS"]:
    local_state["ack"] = {
        "avg_ack_delay": 0,
        "acked": (0, 0),
        "missing": set()
    }
# ...
```

Code 4.6: Creating the local state dictionary for a new flow. Only some of the fields are shown.

With the state available, the parser continues its work with the (optional) parsing of the payload (section 4.3.2), the gathering of the relevant data, and finally the emission of the enabled alerts.

Finally, the function returns three different objects:

- a dictionary, called `info`, containing a JSON representation of the packet fields; this output is provided only if the relevant flag `SAVE_ADDITIONAL_INFO` is provided (appendix A.1);
- a snapshot of the local state of the connection flow this packet belongs to;
- the number of CPU cycles it took to parse the packet, of type *float*.

The first and third variables are provided only for debugging purposes and are not actively used by the program. On the other hand, the second variable is a shorthand for `local_dictionary[key]`.

The following sections describe each of the aforementioned stages of the parsing process in more detail.

4.3.1 Header parsing

The parser function's first step is parsing the various headers. As mentioned before, users have the option to use the parser to parse a whole Ethernet datagram, as in figure 4.2. Alternatively, they can parse just the QUIC packet, previously extracted from the UDP payload. All data is extracted with bit shifts and masks and kept in the `info` dictionary.

The parsing then continues with the QUIC header. Two separate functions exist for this purpose, `parse_long_header()` and `parse_short_header()`. The header type can be obtained with a bitmask, and the according function is called, as in code 4.7.

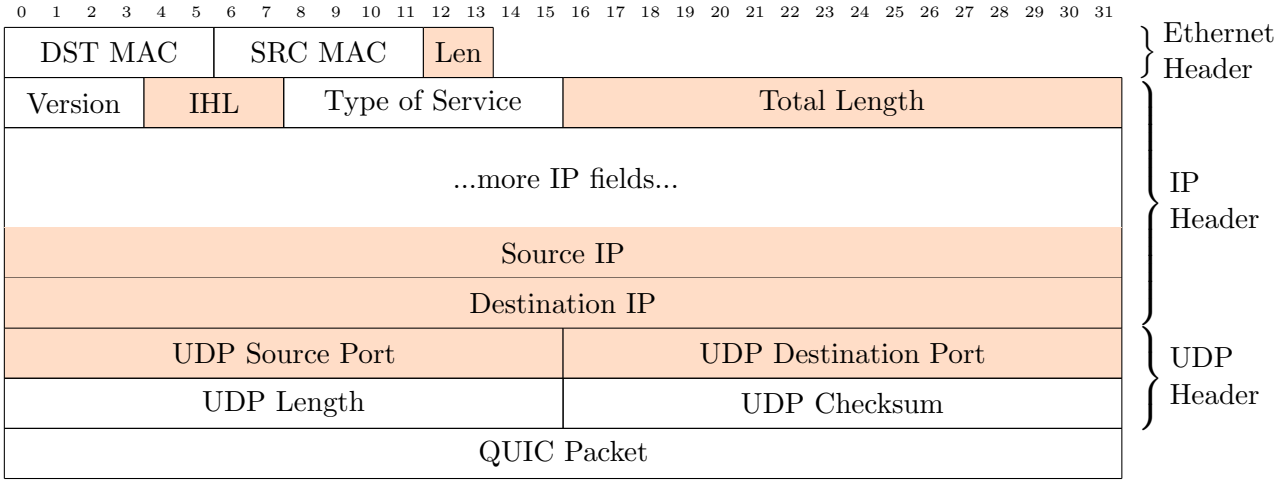


Figure 4.2: A packet, with data being extracted highlighted.

```
if (header_form := (first & 0b10000000) >> 7) == 1:
    quic_header, payload_offset = hparser.parse_long_header(
        first, packet_bytearray, payload_offset)
elif header_form == 0:
    quic_header, payload_offset = hparser.parse_short_header(
        first, packet_bytearray, payload_offset)
else:
    raise ParsingException(f"Invalid header form: {header_form}")
```

Code 4.7: Selection of the parser function.

One critical remark must be made regarding the code snippet. In the whole program, the packet is parsed linearly, with the current position in the packet being represented by the variable `payload_offset`. All functions which operate on the packet and advance the pointer return a tuple (D, O) containing a dictionary with the parsed and a new offset.

In this case, the dictionary D contains all the fields as described in sections 3.4.1 and 3.4.2. In addition, source and destination connection IDs are provided as hexadecimal strings for readability. An integer representation is additionally provided to perform the required comparison to obtain the key as in section 4.3. All fields are again obtained with bit shifts and masks minus the connection ID lengths, which are assumed to be 20 bytes.

Once the header dictionary is returned to the main function, the parser continues parsing the payload, which translates into parsing all the frames contained in it.

4.3.2 Frame parsing

Parsing the frames is trickier than parsing the header and has a higher impact on performance. Users can enable or disable frame parsing with the `PARSE_FRAME` flag.

As said before, the QUIC payload consists of a sequence of frames, each prefixed with a byte-long frame type. Zero-payload payloads are not allowed, and endpoints must fill the payload with `PADDING` frames to reach the minimum packet size. Implementations can also choose to disregard the option to pack multiple frames in the payload and send just one frame.

The frame parser is written recursively. Code 4.8 shows part of the parsing process. The recursive function takes, as usual, the packet `bytearray` and the offset, but also a pointer to a list of frames,

which is updated after each call. The first byte pointed by the offset is inspected[5], and matched against a list of possible frame types. Some frame types, such as `STREAM` and `ACK`, match multiple frame types. `PADDING` frames are separately handled since they can be repeated in the payload.

Once the function to be called is chosen, it is called by reference with the `bytearray` and the offset, returning a frame object and the new offset. The frame object is appended to the list of frames, and the function continues with the next frame.

```
def parse_frames_rec(packet_bytearray: bytearray,
                     payload_offset: int,
                     frames: list,
                     parsing_flags: int = 0) -> list:
    # ...
    frame_type = packet_bytearray[payload_offset]
    frame_type = frame_type & 0b00111111 # Discard two MSB

    if frame_type == PADDING:
        # ...
    elif frame_type in STREAM:
        function_name = parse_stream_frame
    elif frame_type in ACK:
        function_name = parse_ack_frame
    elif frame_type == CONNECTION_CLOSE \
        or frame_type == APPLICATION_CLOSE:
        function_name = parse_closing_frame
    elif frame_type in \
        (STREAM_DATA_BLOCKED, STREAMS_BLOCKED_BIDI, STREAMS_BLOCKED_UNI):
        function_name = parse_streamsblocked_frame
    elif frame_type == NEW_CONNECTION_ID or frame_type == RETIRE_CONNECTION_ID:
        function_name = parse_connid_frame
    else:
        # ...

    if function_name:
        frame, payload_offset = function_name(packet_bytearray, payload_offset)
        # ...
        return parse_frames_rec(
            packet_bytearray, payload_offset, frames, parsing_flags)
    else:
        # ...


def parse_frames(packet_bytearray: bytearray, payload_offset: int) -> list:
    frames = []
    result = parse_frames_rec(packet_bytearray, payload_offset, frames)

    return frames, result
```

Code 4.8: The frame parser. Some pieces of code are omitted for brevity.

Functions called by the frame parser perform sequential operations on a packet depending on the type. Some frame types, like `NEW_CONNECTION_ID`, are very straightforward and require little

[5]The two most significant bits are discarded, as the frame types are encoded using variable integer encoding as in section 3.4.5

operations on the extracted fields. On the other hand, `STREAM` and `ACK` frames are more complicated. Indeed, QUIC acknowledgements use a compact format that allows for the selective acknowledgement of separate ranges of packets (section 3.4.3). Code 4.9 shows a snippet of the implementation.

```
largest_ack, payload_offset = get_varint(packet_bytearray, payload_offset)
ack_delay, payload_offset = get_varint(packet_bytearray, payload_offset)
block_count, payload_offset = get_varint(packet_bytearray, payload_offset)
ack_block, payload_offset = get_varint(packet_bytearray, payload_offset)

if largest_ack < ack_block:
    raise ValueError("largest ack < ack block")

smallest_ack = largest_ack - ack_block
acks = set()
acks.add((smallest_ack, largest_ack))

for _ in range(0, block_count):
    gap, payload_offset = get_varint(packet_bytearray, payload_offset)

    if smallest_ack < gap + 2:
        raise ValueError("smallest_ack < gap + 2")

    largest_ack = (smallest_ack - gap) - 2
    ack_block, payload_offset = get_varint(
        packet_bytearray, payload_offset)

    if largest_ack < ack_block:
        raise ValueError("largest ack < ack block")

    smallest_ack = largest_ack - ack_block

    acks = acks.add((smallest_ack, largest_ack))
```

Code 4.9: A snippet of the code parsing an `ACK` frame.

In this code, four fields are first extracted from the payload: the largest packet being acknowledged, the acknowledgement delay (which is not used in the current implementation), the amount of ACK blocks, and the first ACK block. So first, the first ack range is derived and added to the set of ranges. Then — if present — the subsequent ACK blocks are parsed, adding the relative ranges to the set.

This process adds a layer of complexity to the parsing process. The amount of ACK blocks is unknown until the `block_count` field is parsed, which may depend on several factors. For example, network congestion may cause packets to get dropped, prompting QUIC to send selective acknowledgements and add multiple ACK blocks to the frame.

The same is true for `STREAM` frames. A QUIC connection may open an arbitrary number of streams, and the operation depends on the endpoints, the negotiated parameters, and the type of data sent. While the parsing of `STREAM` frames is more straightforward than `ACK` frames (code 4.10), each new stream opened adds a new entry in the dictionary of streams. This makes the memory footprint of the local state grow.

```
first = packet_bytearray[payload_offset]
payload_offset += 1

stream_id, payload_offset = get_varint(packet_bytearray, payload_offset)

if (first & 0x04) != 0: # Offset
    packet_offset, payload_offset = get_varint(packet_bytearray, payload_offset)
else:
    packet_offset = 0

if (first & 0x02) != 0: # Data length
    frame_len, payload_offset = get_varint(packet_bytearray, payload_offset)
    payload_offset += frame_len
    # ...
else:   # All the remaining bytes
    frame_len = len(packet_bytearray[payload_offset:])
    payload_offset = len(packet_bytearray)
# ...
stream_type = stream_id & 0b11 # Stream type
# ...
return {
    "frame_name": "STREAM",
    "frame_type": f"{first:02x}",
    "stream_id": stream_id,
    "stream_type": stream_type,
    "offset": packet_offset,
    "fin": (first & 0x01) != 0,
    "len": frame_len,
}, payload_offset
```

Code 4.10: A snippet of the code parsing an `STREAM` frame. Bits `0x04`, `0x02` and `0x01` are used as frame flags, hence the multiple frame types mapped to `STREAM` frames.

Once all the frames are parsed, the code deals with a final quirk before returning the parsed content to the main function. As hinted in section 3.3.2, an AEAD tag is appended to QUIC packets as part of the encryption process. However, we assumed that our implementation works on decrypted packets. To simultaneously hold this assumption and trick endpoints into thinking the packet is still encrypted, our client/server implementation of choice (section 5.2) maintains the tag even without performing the encryption steps.

This behaviour causes the addition of 16 bytes of garbage at the end of the packet. The code addresses this by terminating the frame parsing process when it detects that 16 or fewer bytes remain before the end of the packet.

4.4 Data tracking

Once the parsing is complete[6], the program focuses on saving the obtained data to the `local_state` structure. However, the obtained data is not saved as it is, as most information retrieved from the packet is either redundant or irrelevant to the final analysis. Depending on the data type, some transformation steps reduce the amount of data required to be saved, compute some statistics, or

[6]For optimisation purposes, part of the tracking process is completed before the frames are parsed; this does not affect the output.

perform other operations.

Just like the frame parser is optional, so are the various features of the data tracker. Each can be enabled or disabled with `TRACK_*` flags, which are described in appendix A. The various tracking functions are:

- **1RTT packet tracking** (flag `TRACK_PACKET_COUNT`): tracks the highest packet number received in the current connection, the total amount of packets received, and the average length of such packets, calculated with a moving average.
- **RTT measurement** (flag `TRACK_RTT`): tracks the current round trip time and its variance, both calculated with a moving average.
- **Stream tracking** (flag `TRACK_STREAM`): tracks all currently opened streams and the amount of data sent on each of them. Moreover, it keeps track of when packets from each stream were last seen, and when the stream was first opened. Finally, it tracks the average amount of data exchanged on each stream.
- **Acknowledgement tracking** (flag `TRACK_ACKS`): tracks the sets of acknowledged and not-acknowledged packets. Every time a new ack range is added to the set, eventual packets marked as not-acknowledged are removed from the set. Finally, it tracks the latest available acknowledgement delay. Part of this process is shown in code 4.11.
- **Garbage tracking** (flag `TRACK_GARBAGE`): this tracker uses some heuristics to determine if a received packet has been either corrupted or voluntarily sent with an unparsable payload. Furthermore, it tracks `CONNECTION_CLOSE` frames and exposes protocol violation errors sent in them. Finally, it tracks the amount and the time passing between each garbage packet.
- **Connection ID tracking** (flag `TRACK_CONN_ID`): tracks new connection IDs being issued, other connection IDs being retired, and also tracks their corresponding stateless reset token. It also keeps track of each connection ID's sequential number.

```
if flags["TRACK_ACKS"] and frame["frame_type"] in ACK:  # T40
    (ack_delay := frame["ack_delay"])
    (acked := frame["acked"])

    if ack_delay:
        ack_delay = (2 ** ACK_DELAY_EXPONENT) * ack_delay
        ack_delay = ack_delay / 1000  # convert to ms

        if ack_delay > MAX_ACK_DELAY:
            local_state["ack"]["ack_delay_exceeded"] = True

    if acked is not None:
        for rr in acked:
            current_lower, current_upper = local_state["ack"]["acked"]
            lower, upper = rr

            scheduled_for_removal = set()
            for rrm in local_state["ack"]["missing"]:
                if current_lower <= rrm <= current_upper:
                    scheduled_for_removal.add(rrm)

            for rrmd in scheduled_for_removal:
                local_state["ack"]["missing"].remove(rrmd)
```

```
            if lower < current_lower and \
                upper < current_lower:
                local_state["ack"]["missing"].update(
                    set(range(upper, current_lower))
                )
                local_state["ack"]["acked"] = (lower, current_upper)
            # ... further if cases
```

Code 4.11: Updating the ACK tracking data structure.

The tracking part of the program was deliberately written to be as simple and fast as possible. The design choices were made with efficiency in mind, reducing not only the amount of time the program spends calculating the metrics but also minimising its memory footprint in the process. The effort was concentrated on the following points:

- avoid the usage of inefficient data structures, such as lists, instead opting for sets;
- avoid the usage of classes, opting for dictionaries when needed;
- calculate metrics on the fly as the packets are parsed, using a moving average technique, which is described in section 4.4.1;
- keep timestamps as floating point values, as provided by the operating system.

These choices allow for a swift data tracker implementation and a relatively small memory footprint. Nevertheless, some degenerate cases may cause the tracker to increase its memory usage. For example, this may happen during network congestion, as some packets may be dropped and not acknowledged. In extremely bad0 conditions, this may cause a substantial growth of the missing packets set. Other degenerate cases include when endpoints open several streams simultaneously, as each stream has its own set of metrics to be saved, and when endpoints continuously open and retire connection IDs. However, not only are such cases sporadic, but they may be a sign of an incoming attack. Indeed, a case study shown in section 5.1.3 addresses such a scenario.

This approach is far from perfect, and it could be further optimised by using Numpy data types, which are C-compatible and have a fixed size. However, for our purposes, the current design is acceptable, and future improvements should instead focus on rewriting the program in more efficient languages such as C or Rust.

4.4.1 Moving averages

Some of the tracking functions require a moving average to be calculated. Moving averages use a constant — called *smoothing factor* and indicated as α — to obtain an estimate of the average of a series of values. For example, indicating the current average as $\hat{\mu}$ and a new sample as x, it can be summarised with the following formula:

$$\hat{\mu}_i = (1 - \alpha) \cdot \hat{\mu}_{i-1} + \alpha \cdot x$$

This technique is frequently used in statistics and recommended for RTT calculation in section 5.3 on RFC 9002 [44], which documents loss detection and congestion control mechanisms for QUIC. Further information on the management of QUIC flows was taken from another IETF Draft [48]. The former document calculates the average RTT and RTTvar as in code 4.12 (pseudocode). In this case, the smoothing factor α is set to 0.125 for the RTT and 0.25 for the RTTvar.

```
if (handshake confirmed):
    ack_delay = min(ack_delay, max_ack_delay)

adjusted_rtt = latest_rtt

if (latest_rtt >= min_rtt + ack_delay):
    adjusted_rtt = latest_rtt - ack_delay

smoothed_rtt = 7/8 * smoothed_rtt + 1/8 * adjusted_rtt
rttvar_sample = abs(smoothed_rtt - adjusted_rtt)
rttvar = 3/4 * rttvar + 1/4 * rttvar_sample
```

Code 4.12: Pseudo-code for estimating the RTT.

4.5 Alerting

Finally, with the parsing complete and the data tracked and inserted into the corresponding data structures, the program can optionally alert the user about potential issues, either encountered during the analysis of the single packet or derived from the situation of the connection flow.

Our approach to alerting is *query-based*. Users wishing to configure the system must first understand their monitoring needs and then decide which alerts should be enabled and which not. This approach maximises efficiency but potentially leaves parts of the system exposed to some attacks that were not considered previously.

To show the system's capabilities, we implemented four different alerting scenarios. Each one is optional and can be enabled with the `ALERT_*` flags. Section 5.1 shows how they are implemented and to which scenario they cater.

5
Evaluation

This chapter presents several case studies designed to evaluate the system's capabilities. These cases are studied in the literature as possible examples of attacks on the QUIC protocol. Some of them are easier to mitigate and detect, while others consume substantial resources and time to be performed and detected. The aim of this chapter, however, is not to demonstrate the detection's effectiveness but to measure the system's performance when set up to detect a specific set of attacks.

5.1 Case studies

To test the system's capabilities, we devised three different case studies. Each case study focuses on a specific attack scenario on the QUIC protocol. For each case study, we show which features implemented in the system can be used to detect the attack. Indeed, different scenarios require different features to be enabled, ultimately affecting the system's performance.

We stress that this proof-of-concept is not designed to be a feature-complete intrusion detection system. The code provided uses some heuristics to detect attacks in the lightest way possible without any kind of complete payload analysis. However, it is nowhere near complete and would need to be expanded and fine-tuned to be used in a production environment.

5.1.1 Stream Commitment attack

Case study	Stream Commitment [8, 45]

Requirements	`TRACK_PACKET_COUNT`
	`TRACK_STREAMS`
	`TRACK_GARBAGE`

Attack	An adversary opens an unnecessarily large number of streams in a connection. This can be repeated on a large number of connections, like SYN flooding attacks in TCP. For example, opening stream no. `4000000` opens 1 million and 1 client-initiated bidirectional streams.
Consequence	The endpoint's resources are exhausted.
Tradeoffs	Alerting is relatively easy, but it is inherently prone to false positives. Indeed, it revolves around the maximum amount of streams allowed by the endpoint. It is impossible to decide which is the correct threshold for "too many" streams. Also, an attacker could try to guess the threshold and open the maximum amount of allowed streams minus one.

Stream commitment attacks (section 5.1.1) are a type of Denial-of-Service (DoS) attacks on the QUIC protocol. Attackers wishing to deplete the resources of the endpoint open a vast number of streams in a connection, forcing the server to maintain state or each new stream opened.

Servers can easily deflect this attack by restricting the number of streams allowed by the endpoint. This approach, however, may severely limit the endpoint's flexibility, especially for legitimate traffic with a high degree of multiplexing. Moreover, as with other types of DoS attacks, the biggest bottleneck is always the server's computational resources and the amount of memory it uses. An adversary with a sufficiently large amount of memory can still open enough streams to exhaust the server's resources.

To detect this attack sufficiently early, we rely on three features: `TRACK_PACKET_COUNT`, `TRACK_STREAMS`, `TRACK_GARBAGE`. Stream management is fundamental, as it exposes all required stream information, such as the number of streams open and how much data is being sent to them. Packet

counting is required to calculate the average length of received packets. Garbage detection is used to detect if the endpoint sends useless packets, e.g., with empty data. Code 5.1 shows how this is implemented.

```
if flags["ALERT_STREAM_COMMITMENT"]:
    if local_state["streams"] != {}:
        max_stream_id = max(local_state["streams"].keys())

        if max_stream_id > THRESHOLD_MAX_STREAM_ID:
            print_monitor(...)

        for stream in local_state["streams"].keys():
            if local_state["streams"][stream]["smoothed_len"] <
                                        THRESHOLD_STREAM_LEN:
                print_monitor(...)

    if local_state['average_1rtt_length'] < THRESHOLD_PACKET_LEN:
        print_monitor(...)
    if local_state["garbage"]["all_padding"] > THRESHOLD_PADDING:
        print_monitor(...)
```

Code 5.1: Implementation of the stream commitment attack heuristics.

Further heuristics that were tried, but scrapped as they were too inefficient, included calculating "gaps" between the opened streams and alerting if there were too many or too large gaps. Code 5.2 shows how this is implemented.

```
open_stream_ids = sorted(local_state["streams"].keys())
deltas = []
for i in range(len(open_stream_ids) - 1):
    deltas.append(open_stream_ids[i + 1] - open_stream_ids[i])

for item in deltas:
    if item > THRESHOLD_STREAM_DELTA:
        print_monitor(...)
```

Code 5.2: Alerting on large gaps in stream commitment attacks

5.1.2 SlowLoris attack

SlowLoris attacks (section 5.1.2) are not a new type of attack, being described in the literature, used and exploited in real-world attacks for years [88]. Classic SlowLoris attacks opened TCP connections to the target, sending data with a low frequency to keep the connection open. This situation, in turn, resulted in a DoS attack, as, again, the target's memory was exhausted.

In QUIC, SlowLoris attacks can be used similarly to the TCP variant by simply opening several different connections to the target and exploiting the same weaknesses. However, these attacks can be easily mitigated with a proper configuration, e.g., by limiting the maximum amount of incoming connections from a single IP address.

However, a more subtle attack switches to opening several streams rather than several connections and proceeding not to use them and only send data with a low frequency. By combining the two approaches with many adversary endpoints, an attacker can coordinate an attack in which several

Case study	SlowLoris [8, 45, 77, 88]
Requirements	`TRACK_PACKET_COUNT` `TRACK_STREAMS`
Attack	SlowLoris attacks involve opening one or more connections to the target but using them sparingly, sending some bits of data rarely just to keep connections open.
Consequence	Since the end host has to maintain state for each opened connection, SlowLoris attacks consume resources on the target, particularly its memory.
Tradeoffs	SlowLoris attacks usually open only a limited amount of streams in which to send data. We can expect to keep state for a few streams, which in turn would not affect much performance

endpoints open a connection. Each of these connections will open several streams and leave them open, sporadically sending data to prevent timeouts.

The implementation is very similar to the stream commitment case study described in section 5.1.1. Out of the three case studies presented, SlowLoris was found as the most difficult one to balance, as introducing too many checks would slow down the overall performance. We tried to introduce a cross-connection correlation between possible attackers, but this further slowed down the whole system's performance. Code 5.3 shows the retained implementation.

```
if flags["ALERT_SLOWLORIS"]:
    open_stream_ids = sorted(local_state["streams"].keys())
    for stream in open_stream_ids:
        if local_state["streams"][stream]["smoothed_len"] < THRESHOLD_STREAM_LEN:

            print_monitor(...)
        if local_state["streams"][stream]["last_seen"] \
            + THRESHOLD_STREAM_TIME < time.time():
            print_monitor(...)
```

Code 5.3: Implementation of the SlowLoris heuristics.

5.1.3 Stream Fragmentation attack

Case study	Stream Fragmentation [8, 45]
Requirements	`TRACK_STREAMS` `TRACK_ACKS`
Attack	Stream Fragmentation attacks involve an adversary intentionally not sending portions of the data it is supposed to send. For example, adversaries can send stream data with missing portions or fail to acknowledge packets.
Consequence	Stream Fragmentation attacks cause a disproportionate receive buffer memory commitment or the creation of a large and inefficient data structure at the receiver. They can also force senders to store unacknowledged stream data for retransmission.
Tradeoffs	Extremely hard to manage. It requires full parsing of the whole packet and every stream being used but needs additional state to keep track of the fragments being received and not effectively duplicating the end host's stream management. It requires a timeout, in which opened streams that are either dangling or not fully acknowledged are dropped. However, it is prone to false positives.

Stream fragmentation and reassembly attacks (section 5.1.3) are slightly different from the two aforementioned case studies. Instead of consuming resources via *not* utilising streams, stream

fragmentation attacks consume the server's resources by using streams (and acknowledgements) wrong intentionally. Attackers may open one or more streams, apparently legitimately sending data, before interrupting and leaving the request hanging. They may send ACK sporadically or packets with missing portions, intending to consume even more of the server's resources.

While SlowLoris and stream commitment attacks require a "numerical" effort from attackers, i.e., more processing power, stream fragmentation attacks instead employ sophisticated techniques and algorithms on the attacking side to deduce when to stop sending data or acknowledgements correctly. In turn, detecting such attacks becomes more complex on the defending side, as the system now has to keep track of the state of streams and acknowledgements. Moreover, it must also realise when an endpoint is intentionally not sending data and when it derives from a heavily congested network. Moreover, overcommitment strategies [45] render endpoints even more vulnerable to these attacks as they enlarge the server's memory footprint. Finally, when attackers delay the acknowledgement of packets, they increase the amount of stored data as packets are kept in buffers, ready for retransmission.

Code 5.4 shows the implementation.

```
if flags["ALERT_STREAM_FRAGMENTATION"]:
    # Heuristics for packets still missing
    if len(local_state["ack"]["missing"]) > \
            1/10 * local_state["ack"]["acked"][1]:
        print_monitor(...)

    # Checking for big holes in reassembly
    current_hole = 0
    missing_list = sorted(local_state["ack"]["missing"])
    for i in range(len(local_state["ack"]["missing"]) - 1):
        if missing_list[i] + 1 == missing_list[i + 1]:
            current_hole += 1
        else:
            if current_hole > THRESHOLD_ACK_HOLE:
                print_monitor(...)
            current_hole = 0

    # Check stream usage, like SlowLoris
    open_stream_ids = sorted(local_state["streams"].keys())
    for stream in open_stream_ids:
        if local_state["streams"][stream]["smoothed_len"] \
            < THRESHOLD_STREAM_LEN:
            print_monitor(...)
        if local_state["streams"][stream]["last_seen"] \
            + THRESHOLD_STREAM_TIME < time.time():
            print_monitor(...)
```

Code 5.4: Implementation of the stream fragmentation attack heuristics.

We implemented very simple heuristics (several missing acks, significant reassembly gaps, and overall stream usage), which significantly impact performance. We believe these are insufficient to detect stream fragmentation attacks properly, and additional metrics based on the duration of reassembly gaps would be necessary. Doing so, however, would require additional state to keep track of the duration of such gaps.

5.1.4 Further uses

The three aforementioned case studies focus on DoS attacks, but the system's capabilities may also be used to manage other situations. First, connection ID tracking is available but not used in the current implementation. It could be either used to detect stateless reset attacks [45], a type of attack in which adversaries try to terminate a connection by sending guessed Stateless Reset tokens (section 3.3.1). Furthermore, it could be used to detect coordinated SlowLoris attacks as described in section 5.1.2. Finally, other possible attacks that were experimented with but ultimately not implemented include:

- spoofing and amplification attacks, in which an adversary spoofs its source IP and source destination ID and causes it to send large amounts of data to an unwilling host. Moreover, connection migration might be used to amplify the volume of data that an attacker can generate toward a victim;
- blatant violations of the QUIC protocol, with the hope of causing the system to crash;
- connection migration and path validation abuse, in which an adversary repeatedly migrates connections to a different destination IP address;
- attacks on the TLS protocol, which is not covered by the current implementation.

On the other hand, some metrics, such as RTT and average packet length, could be used to detect unusual performance anomalies at the connection level. Therefore, we devised a dummy case study, labelled `ALERT_PERFORMANCE_ANOMALY`, as a testing ground for implementing alerts for:

- high RTT and RTT spikes;
- low or decreased throughput;
- ECN flags being set;
- critical network congestion.

Although in the final implementation, we scrapped these alerts, the case study remains available and employs the flags `TRACK_RTT`, `TRACK_ACKS`, and `TRACK_GARBAGE`. This allows us in section 5.3 to track a theoretical baseline of monitoring such metrics.

5.2 Environment

This section describes the specifications of the environment in which the implementation is deployed and the tests described in section 5.3 are executed.

5.2.1 Traffic generation

To generate QUIC traffic, we used a customised version of Quiche [10], which is a QUIC client/server library written in Rust. Quiche was edited to bypass the encryption methods while keeping the extra 16 bytes required by the AEAD tag. This tricks the protocol into thinking the payload is encrypted. While this step may seem avoidable, removing the AEAD tag step yielded many unintended side effects, which could have been fixed only by making radical modifications to the code. However, with all packets being analysed as decrypt, we were unable to provide case studies focused on the cryptographic aspects of the protocol.

Quiche generates connection IDs of precisely 20 bytes to comply with the assumptions outlined in section 4.1. It also sends Initial requests with a version of `0xafafafaf`. This prompts the server to send a Retry packet, switching the version to `0x00000001` and changing the connection IDs in the process.

The modified library is available at `https://github.com/mfranzil/quiche/`.

At the server side, we generated fake `index.html` files to serve as the QUIC traffic (code 5.5). We generated files of 1, 10 and 100 times a `KiB` and a `MiB` in size[1].

```
head -c 1KiB /dev/urandom > index-1k.html
head -c 10KiB /dev/urandom > index-10k.html
head -c 100KiB /dev/urandom > index-100k.html
...
```

Code 5.5: Generating fake index.html files.

Concurrent flows from the same client were simulated with a simple `for` loop in the command line (code 5.6).

```
for i in $(seq 1 10); do
    ./client-request.sh https://192.168.50.4:4433/index-1m.html --silent &
done
```

Code 5.6: Generating multiple flows at once.

5.2.2 Virtual machines

We performed our tests on a dedicated machine with the following specifications:

- **CPU**: 24 × Intel Xeon X5660 @ 2.80GHz
- **Memory**: 84 GB
- **OS**: Ubuntu 20.04.4 LTS
- **Kernel version**: 5.4.0
- Nested virtualisation enabled

We used Vagrant to create disposable virtual machines generating traffic for the host. The OS and Kernel version of the VM matched the ones of the host but with reduced resources. The virtual machines were created with a Vagrantfile and several scripts, which were all made available in the `vm` directory of the repository. Due to several dependency conflicts, most tools we installed were built from source. These included BCC, OpenSSL, curl and libcurl, and PyShark.

We believe the application should work on any Linux-based operating system as long as it is new enough to support at least a parser mode as specified in section 4.3.

5.2.3 Network topology

The testing network was structured as the following. On the host machine, we installed the Quiche server. On the VM, we installed the Quiche client. We configured a bridged network connection between the two, separate from the NAT network provided by Vagrant and used for connecting to the Internet.

Thus, client requests ran directly from the VM to the host using the bridge. This allowed us to separate the traffic, letting the monitoring program sniff only QUIC packets travelling on the bridged network interface. Moreover, it allowed us to reduce latency to a minimum and obtain more predictable results.

[1] All sizes in this document use binary prefixes, i.e., `1KiB` is 1024 bytes, `1MiB` is `1024KiB`.

For simulating multiple connecting clients, we built an updated version of the official Quiche Docker image, integrating our modifications. When needed, we launched on-the-fly some containers, each connected to the bridge network and running a Quiche client. However, for the purposes of the case studies shown here, we generated traffic only from the Vagrant VM, as it was found to be more reliable and capable of generating a larger amount of data.

Figure 5.1 shows an overview of the architecture used.

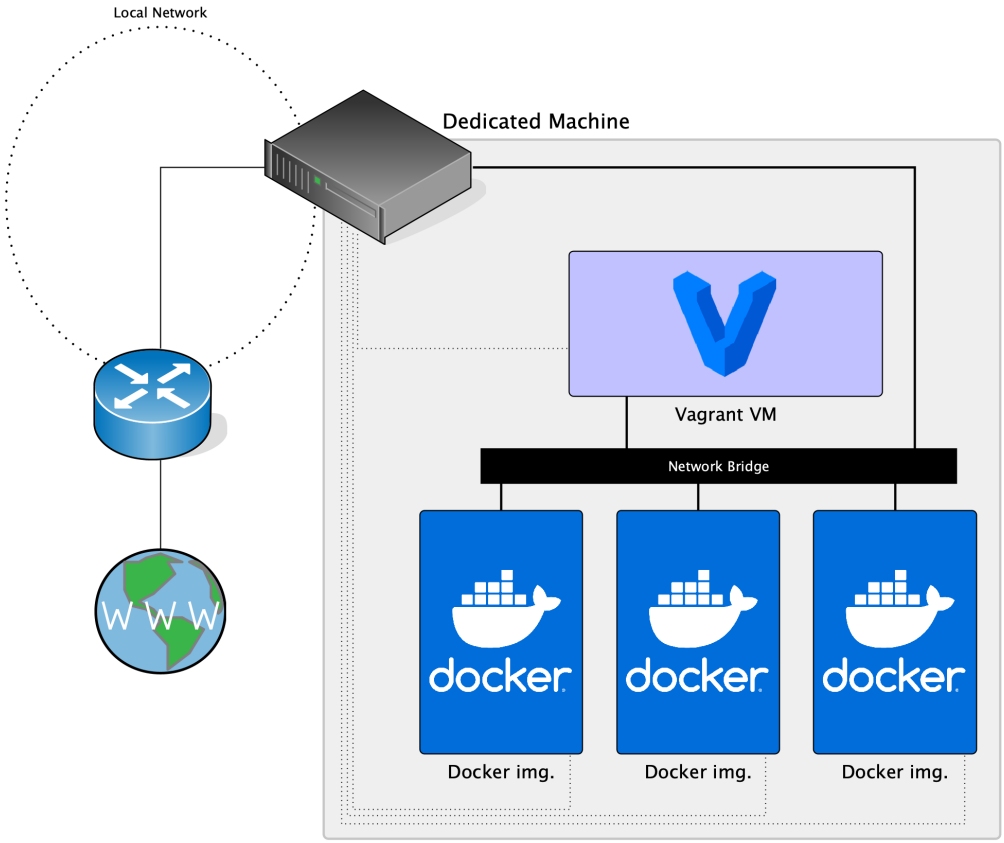


Figure 5.1: The testing network.

5.2.4 Program configuration

The program was equipped with two configuration files:

- the `config.py` file contained most constants for the program; however, it also contained several "hyperparameters", which were not directly controlled from the command line. Instead, they were designed to be tuned by tenants better to match their requirements and the status of the network. Examples of them include:
  - `THRESHOLD_ACK_HOLE`: the size of the largest hole in the acknowledgement set. These types of threshold-based parameters are used in the upcoming case studies to determine when to emit alerts;
  - `SMOOTHED_RTT_FRAC`: the smoothing factor used in the smoothed round-trip time calculation;
  - `DISCARD_CID_WINDOW`: the amount of seconds the system keeps state for an inactive connection flow before discarding it.
- the `flags.py`, which instead controlled which features are enabled or disabled in the program. It also contains code for parsing them from the command line, deriving the necessary dependencies, and enabling them. A full list of implemented flags is available in appendix A.

5.2.5 Test orchestration

With the environment set up, a variety of tests were made. To distinguish tests from each other, we differentiated them based on two parameters:

1. the configuration flags enabled;
2. the flow size, i.e., the total amount of `bytes` the Quiche client received from the host.

In particular, we used the following configuration flag combinations:

- with no flags enabled;
- with exactly one of each of the tracking flags (appendix A.3);
- with all the tracking flags together;
- with exactly one of each of the alert flags (appendix A.4);
- with all the alert flags together.

On the other hand, we used the following flow sizes:

- **mice flows**:
  - 10 KiB
  - 100 KiB
  - 1 MiB
- **regular flows**:
  - 10 MiB (10×1 MiB)
  - 100 MiB (100×1 MiB; 10×10 MiB)
- **elephant flows**:
  - 1 GiB (1000×1 MiB; 100×10 MiB; 10×100 MiB)
  - 10 GiB (1000×10 MiB; 100×100 MiB)

Mice flows were all set up to simulate a single client contacting a server for a small webpage. However, simulating all types of flows with clients contacting servers with a single connection would not have been representative of real-world traffic. Thus, we opted for dividing regular and elephant flows into smaller, concurrent ones. Regular and elephant flows were all divided into 1, 10, 100, or 1000 separate client flows requesting data from 1 MiB to 100 MiB each. Such concurrency was obtained with either multiple Docker containers or with for loops, as previously described.

Once the tests were set up, each run of the program was orchestrated with some Bash scripts to obtain some additional metrics from the experiment. In particular, CPU and memory usage were tracked with a second-precision `top` command. These steps are shown in code 5.7. Finally, a Python script collects, parses, and plots the results using the `matplotlib` library.

```
python3 src/main.py -i br-fc2518b3193b -m SC -f monitor \
    -d auto -x "$FLAGS" -c "$COMMENT" >/dev/null &
sleep 2

PID=$! # Gets PID of the last process
taskset -cp 0 $PID

trap cleanup EXIT

function cleanup() {
```

```
    verify_file_size
    kill $PID
}

function verify_file_size() {
    oldcount=0
    count=$(cat ... | wc -l)

    no_changes=1
    while [[ $no_changes -lt 2 ]]; do
        # Checks every second if the
        # amount of lines changed.
    done

}

top -b -d 1 -p $PID | awk -v FLAGS="$FLAGS" \
    -v COMMENT="$COMMENT" -v OFS="," \
    '$1+0>0 { print FLAGS,COMMENT,$1,$6,$9; fflush() }' >> ...
```

Code 5.7: A snippet of the code automating the tests.

The results are summarised in section 5.3.

5.3 Performance analysis

In this section, we present the results of the performance analysis of our QWIC implementation against the case studies discussed above.

5.3.1 Parsing time

The first graphs are concerned with evaluating how much time the program spends handling incoming packets, regardless of the flow. To measure this, we settled on calculating the *median parsing time*. The median parsing time is calculated in clock cycles using the `hwcounter` Python library and pinning the process to a dedicated CPU core. This minimises interference from other processes and yields more accurate results. The median parsing time is calculated from the start of the call to `parse_packet()` to its end. This includes all parsing, tracking, and alerting steps but excludes time spent waiting for CPU and I/O.

Due to a series of factors, the parsing time can be highly volatile. The overhead introduced by the subset of parsing, tracking, and alerting flags being used is the most significant contributor, and its impact and influence are explored in the following sections. However, within a single run, the parsing time can fluctuate by a significant amount. Reasons for this include:

- the type of header of the packet being parsed. Long headers are usually at the start of the packet. They cause the creation of fresh data structures for new connections and contain more information to be saved. Thus, they cause a disproportionately large parsing time;
- the payload of the packet. For example, acknowledgement frames can be extremely convoluted and could be parsed more slowly;
- the current state of the connection. For instance, as a flow progresses, the saved state (e.g., the open streams, the available connection IDs, the acknowledgements) may become bigger, and the tracking and alerting operations have to handle and process more data.

The results are plotted as boxplots, with the x axis containing various configuration setups that are explained in each graph. On the y axis, boxes extend from the first to the third quartile, i.e., from the 25th to the 75th percentile. Whiskers extend from the box 1.5 times the inter-quartile range (IQR), which is defined as the difference between the 75th and 25th percentile.

Tracking flag evaluation

The following graphs show the performance of the program with the following tracking flags enabled:

- `BASELINE` - no tracking;
- exactly one of `TRACK_PACKET_COUNT`, `TRACK_RTT`, `TRACK_STREAMS`, `TRACK_ACKS`, `TRACK_GARBAGE`, `TRACK_CONN_ID`; each flag is described in section 4.4;
- all of the above together.

Figure 5.2b shows the median parsing time required for a packet when the system receives 100 connections, each requesting a 1MiB file. Figure 5.2a shows the same for a single connection for a 1MiB file.

From figures 5.2a and 5.2b, we can immediately see the different impacts of each configuration flag on the parsing time. The baseline case is the fastest. The `TRACK_RTT` and `TRACK_PACKET_COUNT` flags, which do not analyse the payload, are slightly slower. However, using any of the other four flags significantly slows down the parsing time, with an increase of up to 83%. Enabling all feature flags further puts a strain on the system, with the difference between it and the baseline case nearing 100%.

Furthermore, we can see that the performance of the system remains predictable even as the number of concurrent connections increases. A 100-times increase in the number of connections increases the variance (and thus the quantiles of the boxplots) of each configuration. The clear standout here is `TRACK_ACKS`, whose median does not change, but its variance increases greatly; 50% of packets containing an acknowledgement take anywhere between 90000 and 120000 clock cycles to parse.

Testing with greater amounts of either connections or data did not yield any significant difference in the results.

Alert flag evaluation

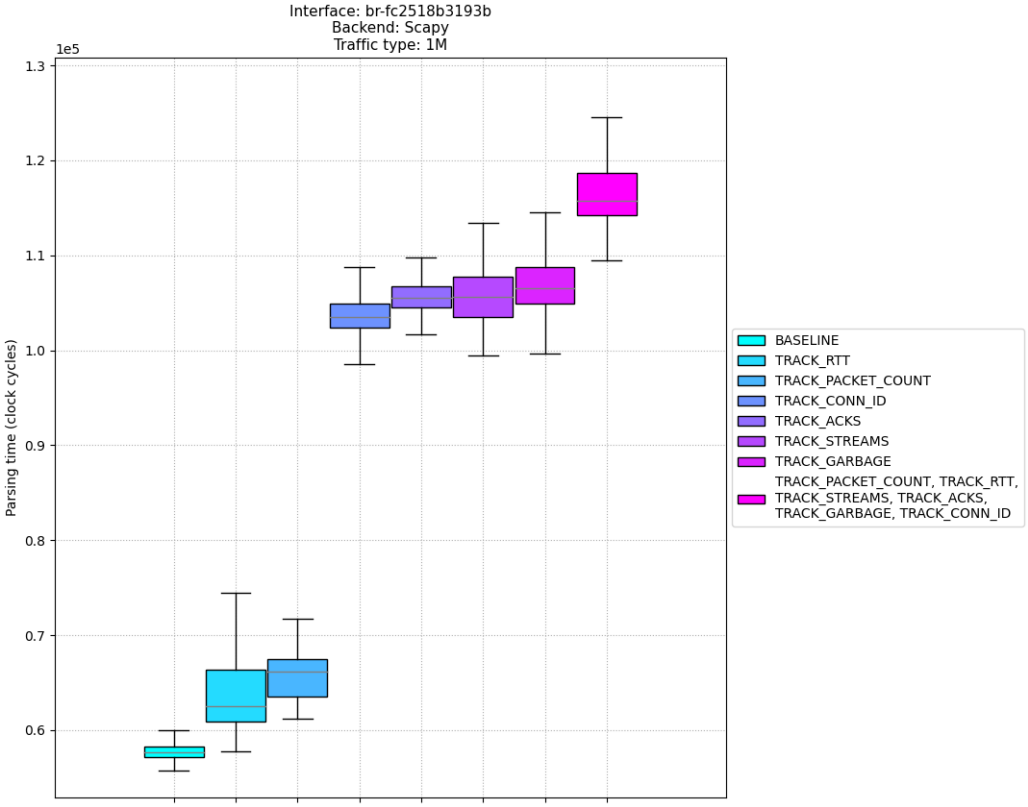
The following graph shows the performance of the program with the following alert flags enabled:

- `ALERT_STREAM_COMMITMENT`, as described in section 5.1.1;
- `ALERT_SLOWLORIS`, as described in section 5.1.2;
- `ALERT_STREAM_FRAGMENTATION`, as described in section 5.1.3;
- `ALERT_PERFORMANCE_ANOMALY`, a dummy case study briefly mentioned in section 5.1.4.

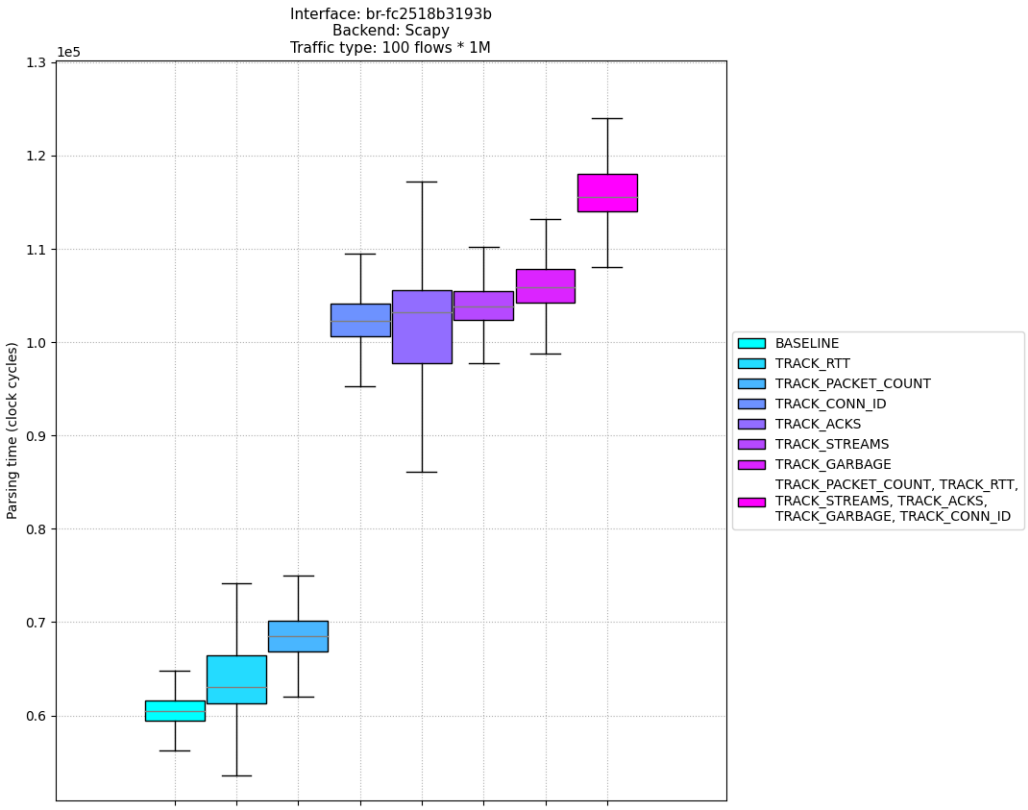
This approach is similar to the one used just above. However, alert flags may not only employ a subset of tracking flags but also add complexity due to the additional code needed for the alert emission. Figure 5.3 shows the results for the above alert flags, tested with 100 connections, each requesting a 1MiB file. Again, tests with a different amount of connections and data did not yield any significant difference in the results.

Baseline results are on par with previous results and are again the fastest, clocking at an average of roughly 60000 clock cycles per packet. `ALERT_PERFORMANCE_ANOMALY` is the least demanding alert flag due to its lightweight reliance on RTT and garbage tracking. However, its variance is also higher, again due to the implementation of the ACK tracking.

On the other hand, the other three case studies (stream commitment, SlowLoris, and stream fragmentation) are more demanding due to their heavy reliance on payload parsing and the addition of more code required for the alert emission. Overall, all three cases require an average of 120000 clock cycles per packet to parse, which is double the baseline. Finally, enabling all four case studies at once



(a) A single connection requesting a 1MiB file.



(b) 100 connections each requesting a 1MiB file.

Figure 5.2: Parsing time of a single packet with varying degrees of traffic and tracking flags enabled.

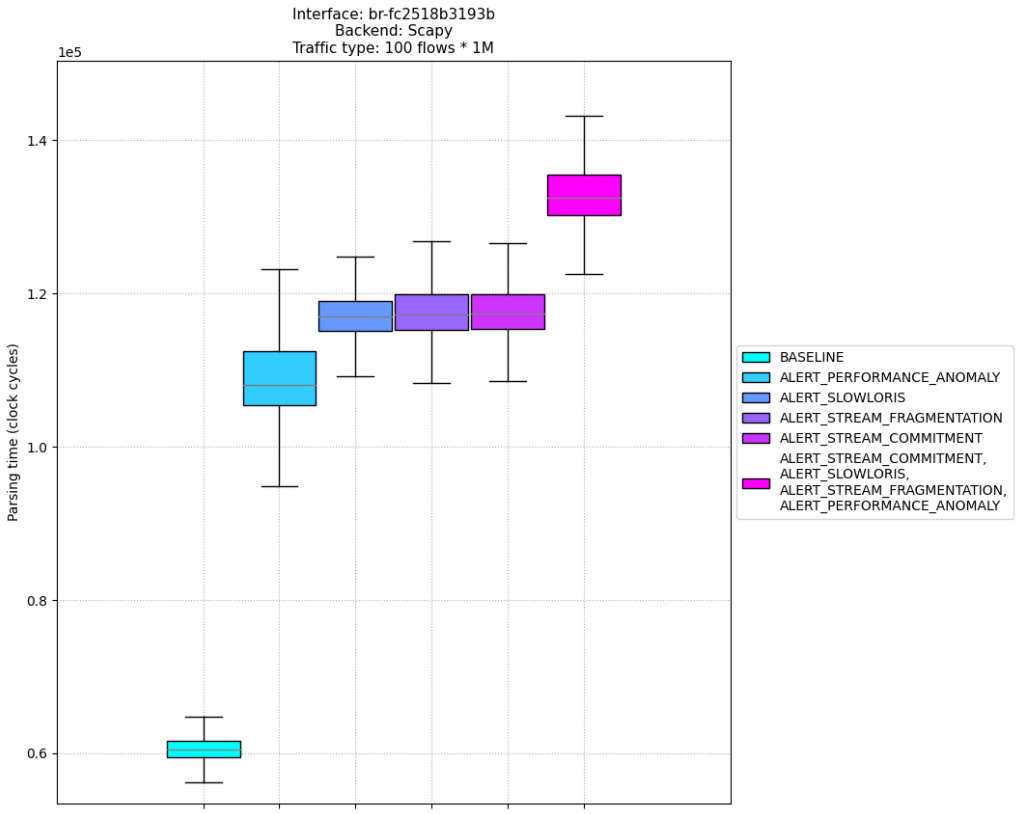


Figure 5.3: Parsing times for 100 connections, each requesting a 1MiB file, with various alert flags enabled.

further increases the required time to 135000 clock cycles per packet, which is 12,5% more than the three case studies and 125% more than the baseline.

In short, with the current feature set, users could enable all four alerts at the expense of a slight performance loss. However, any modification to these alert flags or to the parser itself could radically change the results and would require further analysis.

5.3.2 Memory footprint

Finally, we take a look at the memory footprint of the system.

Theoretical bounds

To begin with, we calculate the theoretical memory footprint of the various configurations of the program. Indeed, as specified in section 4.4, each tracking flag adds some amount of state to the program and thus increases memory usage. On the other hand, alert flags add only a small overhead to the overall usage, mainly in the form of temporary variables and data structures that are later dropped by the garbage collector. We also assume the memory usage of the sniffing and header parsing code to be roughly constant regardless of the active features. The payload parsing code is excluded from this, as it is highly variable depending on the amount of payload data. Thus, we focus our calculations exclusively on the tracking flags.

The following calculations were performed on running code with the `sys.getsizeof()` call, with some exceptions. Indeed, numeric types are dynamic in Python, and their size may grow depending on the size of the held value. From our observations, we settled on using 24 bytes for floating-point timestamps, 36 bytes for integers derived from variable-length integers (section 3.4.5), and 32 bytes for all other integers.

Firstly, we calculate the memory overhead required to keep track of a single connection with no flags set. An empty `local_state` variable holds the source connection ID and the destination ID

(both as strings) and when the last packet was seen (a timestamp). This amounts to $64 + 89 + 89 + 24$ bytes. This does not count the space required for memorising the key in the relative data structure (section 4.3), which amounts to an additional $64 + 64 = 128$ bytes. This overhead is constant and will not change if any additional flag is enabled. This amounts to $266 + 128 = 394$ bytes of overhead for each connection.

Secondly, we calculate the memory overhead of each flag:

- `TRACK_PACKET_COUNT`: three integer metrics (last packet number, total packets sent, average packet size) amount to $32 + 32 + 24 = 88$ bytes;
- `TRACK_RTT`: smoothed RTT, RTT variance, the current spin bit value and a timestamp amount to $24 + 24 + 32 + 24 = 104$ bytes;
- `TRACK_STREAMS`: here, calculations get tricky as an arbitrary number of streams can be open. Given S as the average number of streams opened, we need to track the packet count, the type, its smoothed length, and two timestamps tracking the last packet seen and when it was opened. This amounts to $(32 + 32 + 24 + 32 + 24) \cdot S = 144 \cdot S$ bytes;
- `TRACK_ACKS`: again, the number of pending acknowledgements is not constant, but here the upper bound is given by the number of packets sent. We need to track the currently acknowledged range, a boolean flag, and a set that may hold up to the number of sent packets. This amounts to $32 + 32 + (216 + P \cdot 32) + 32$ bytes $= 312 + P \cdot 32$ bytes, where P is the total number of packets that may be sent in that connection;
- `TRACK_GARBAGE`: again, a collection of integer and timestamp metrics, amounting to $32 + 32 + 32 + 24 + 32 = 152$ bytes;
- `TRACK_CONN_ID`: this depends on the maximum number of connection IDs, which is also not a constant and may grow up to the provided limit (although empirically, it has been observed to be limited to 3 at most, as connection IDs may be retired and replaced during the lifespan of a connection). For each connection C, we need to hold $36 + 32 + 36$ bytes (as the `retire_prior_to` and sequence number fields are variable-length integers). This is a total of $C \cdot 104$ bytes.

Table 5.1 summarises this information.

Tracking flag	Footprint
Baseline (no flag)	394 bytes
`TRACK_PACKET_COUNT`	88 bytes
`TRACK_RTT`	104 bytes
`TRACK_STREAMS`	$144 \cdot S$ bytes
`TRACK_ACKS`	$312 + 32 \cdot P$ bytes
`TRACK_GARBAGE`	152 bytes
`TRACK_CONN_ID`	$104 \cdot C$ bytes

Table 5.1: Memory overhead for a single connection and various tracking flags.

With all the tracking flags enabled, this would amount to a total of:

$$\begin{aligned} &394 + 88 + 104 + 144 \cdot S + 312 + 32 \cdot P + 152 + 104 \cdot C \\ &= 1050 + 144 \cdot S + 32 \cdot P + 104 \cdot C \end{aligned}$$

with $C \geq 1, S \geq 0, P \geq 0$ as defined earlier.

With these results, let us calculate the memory impact of various program configurations in two different scenarios.

From our observations in lossy (but not disastrous) networks, we noticed common values of $P \leq 10$. Moreover, with elephant flows lasting a long time, servers tended to issue at least a new connection ID

($C = 2$) and open several streams at once to facilitate multiplexing ($S \leq 5$). This is our first scenario. Substituting these values in the previous equation, we obtain a theoretical worst-case memory footprint of 2.298 bytes, more than two kibibytes, and almost six times the baseline overhead.

Clearly, the above figure is exaggerated due to network conditions, and estimates can vary greatly. As a second scenario, we take into account a fresh, healthy connection. From our observations, such a connection will open three single streams, have a single connection ID, and will have lost no packet. Thus, $C = 1, S = 3, P = 0$. In this case, the theoretical memory usage drops to 1.586 KiB.

This highlights how much poor network conditions (congestion, loss, ...) impact memory usage due to the missing packets increasing the size of the unacknowledged set. In average network conditions — thus with few to no retransmissions — we expect an average connection to require somewhere between 1.4 and 1.6 KiB of overhead. On the other hand, even worse network conditions can increase the aforementioned overhead even further.

The above results refer to a hypothetical worst-case configuration in which all the tracking flags are enabled. Such a setup never happens in practice: even with all the case study alert flags set, not all features implemented in the program are used. To provide representative figures for our analysis, we calculated the memory footprint for each case study's alert flag and all the case studies enabled together. These, along with the aforementioned worst case, are presented in figures 5.4 and 5.5. Again, there is an additional overhead that is inherent to each case study caused by temporary variables and data structures. For simplicity, we do not take into account this overhead as variables are garbage-collected after a certain amount of time.

The first figure, figure 5.4, uses $C = 2$, $S = 5$, $P = 10$ as before, thus representing a lossy network scenario.

Again, we notice that enabling all the alert flags in poor network conditions increases the memory overhead by almost 400 bytes when compared to the most memory-intensive scenario (stream fragmentation). If this were to scale over 1,000,000 connections, it would amount to an additional 400 megabytes of used memory. This may or may not be acceptable, depending on the capabilities of every single deployment.

On the other hand, figure 5.5 shows the same bar chart, but with $C = 1, S = 3, P = 0$, the setting for an ideal scenario in a healthy network. With this setup, the memory footprint for each case study not only drops significantly but also the differences between each case study become smaller.

Experimental results

In practice, a system will allocate memory differently depending on the specific data it encounters. Moreover, in the code, temporary values may be created and then quickly discarded, but the user has no control over when the garbage collector will run and free them. Therefore, in this section, we perform actual measurements on the Python side to examine the actual memory usage observed while parsing a trace.

First of all, several overheads must be taken into account (in addition to the ones previously calculated). The sniffing part of the program takes an enormous amount of memory, even when no traffic is being reported on the interface. Indeed, from our measurements, this amounted to over 100 MiB. To avoid interferences and possible distortions, we temporarily separated this part of the program into a separate process and performed the measurements on the parser-tracker application only. However, not all overheads can be eliminated. Indeed, all the following measurements still take into account Python's baseline memory usage, including the import of the libraries that we use, the program's code, the packet buffers, and more.

Let us consider the case in which 10.000 concurrent connections are opened in a healthy network environment, each retrieving a page of 1 MiB. From our previous calculations, we expect usage of roughly $394 \cdot 10,000 \approx 3.94$ MiB for the baseline case and $1,482 * 10,000 = 14.82$ MiB for the case where we enable all the alerting flags.

Figure 5.6 plots the memory footprints for the aforementioned scenario.

At the top, the theoretical bounds are shown, calculated using the previously used parameters $C = 1$, $S = 3$, $P = 0$. On the other hand, the bottom picture shows the actual results observed. Due to the aforementioned Python overheads, the figures are higher. However, we can still see a growing trend

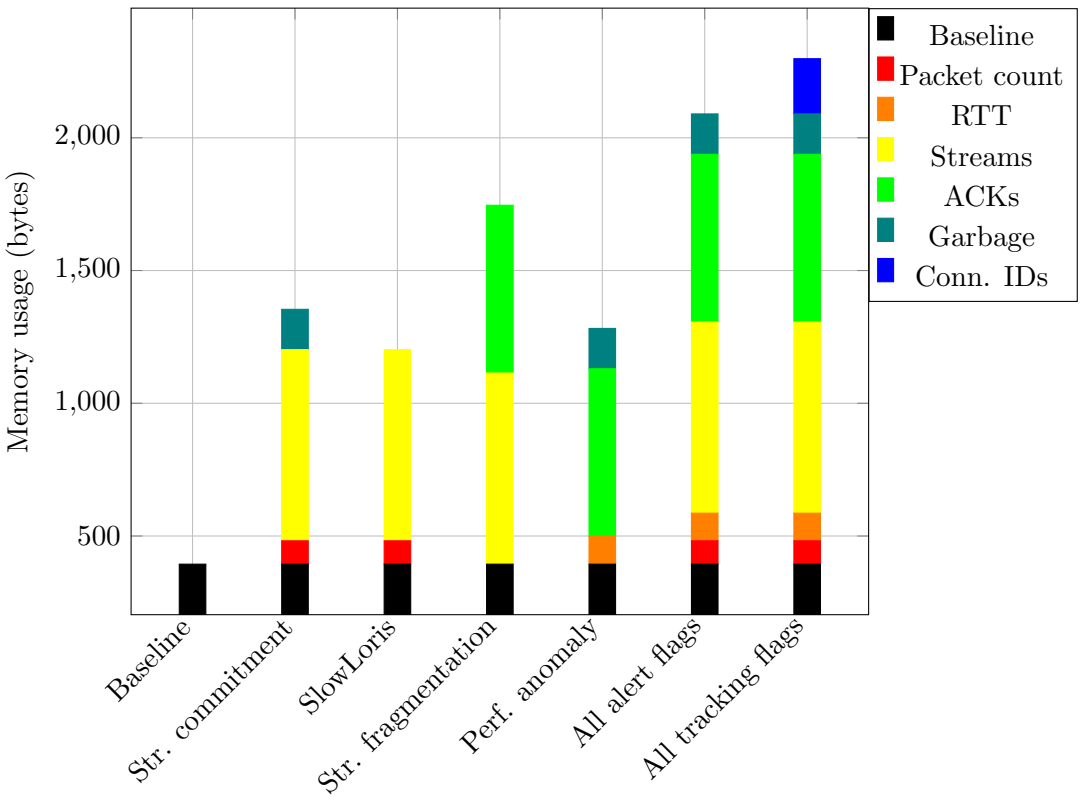


Figure 5.4: Theoretical memory footprints of a single traffic flow, with $C = 2$, $S = 5$, $P = 10$ of a baseline case vs. each case study vs. all case studies enabled together vs all the tracking flags enabled together.

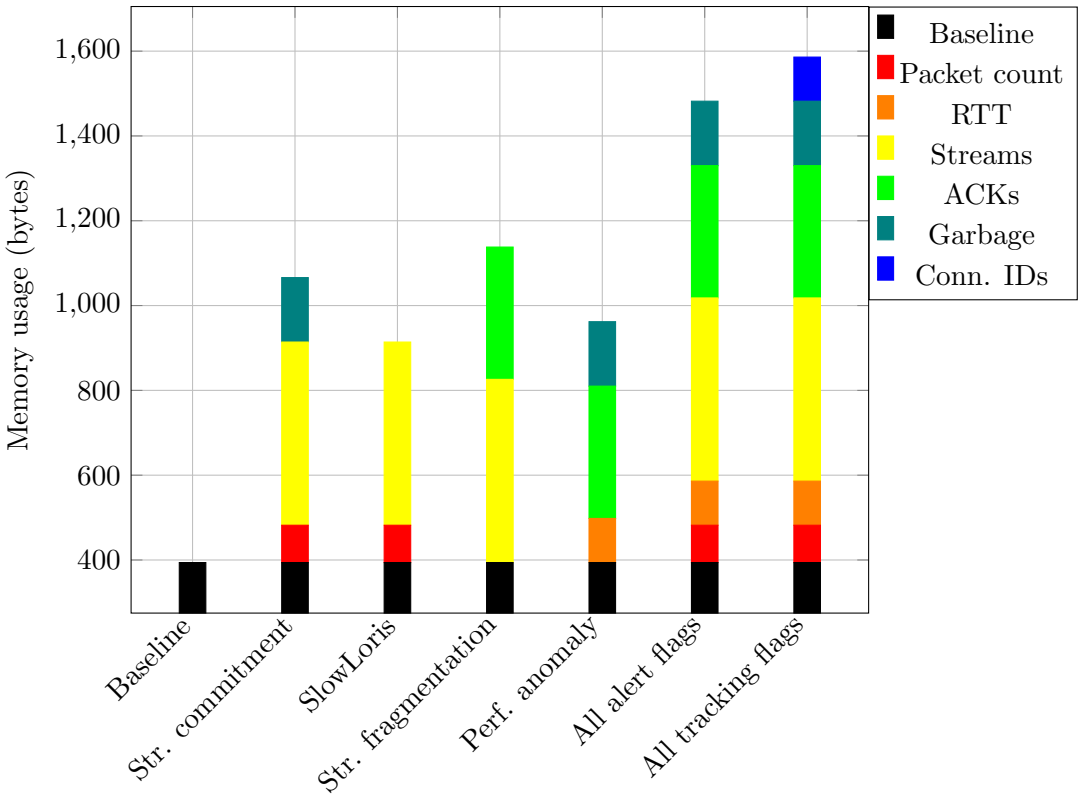


Figure 5.5: Theoretical memory footprints of a single traffic flow, with $C = 1$, $S = 3$, $P = 0$ of a baseline case vs. each case study vs. all case studies enabled together vs all the tracking flags enabled together.

similar to the one in the theoretical memory footprints, with the baseline being the least impactful and all alert flags being the most memory-intensive — up to 50% more than the baseline case.

With all this in mind, one would assume that the calculations performed above would not vary when the system is exposed to increasing levels of traffic, instead only changing depending on the number of actual connections being tracked and the configuration. However, this was not the case.

Figure 5.7 plots the memory usage (again, only for the main part of the program) for all the tests we ran with different amounts of traffic, regardless of the number of tracked connections or the configuration of the system. Each black dot in the scatterplot represents the maximum amount of memory recorded during a single run of the program while being exposed to a certain total amount of traffic.

We can observe that as the total amount of traffic increases, memory usage grows linearly. For minor traffic volumes, this amounts to around 20 MiB but grows to over 80 MiB for elephant flows traffic carrying 10 GiB. In theory, such an increase should not happen, as our previous calculations were agnostic of the traffic volume. We attribute this increase to the number of temporary values created when performing the parsing and processing. Due to the way the program works, some packets may require more computational power than others: they may contain several acknowledgement ranges, contain several frames, and trigger one or more alerts, while others may not. This caused several fluctuations and noise in the memory usage during our live monitoring, but we were nevertheless able to get a rough idea of the maximum usage in each test.

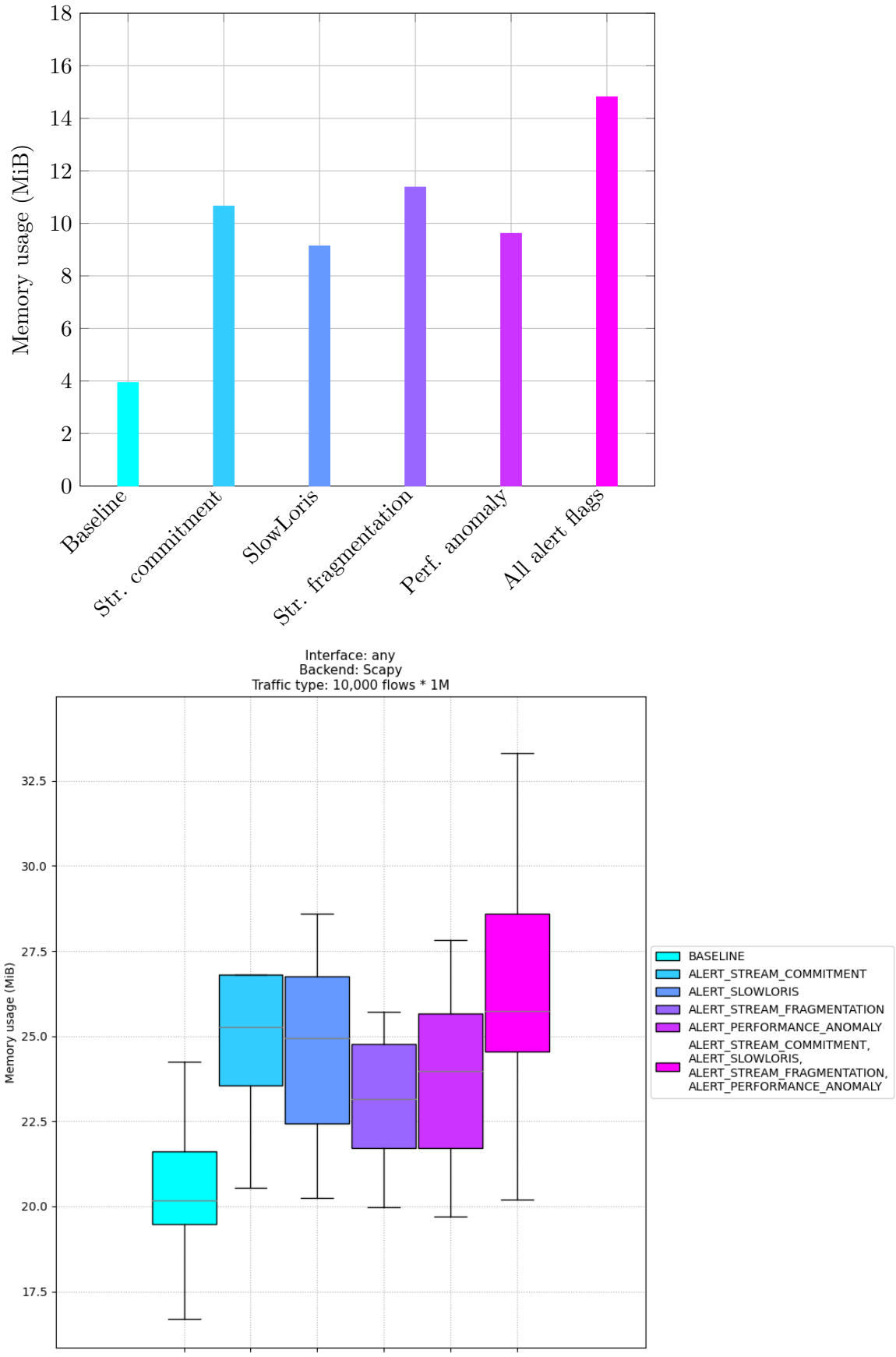


Figure 5.6: Theoretical (top) and actual (below) memory footprints while handling ten concurrent connections, each transferring a 1 MiB file in a healthy network configuration.

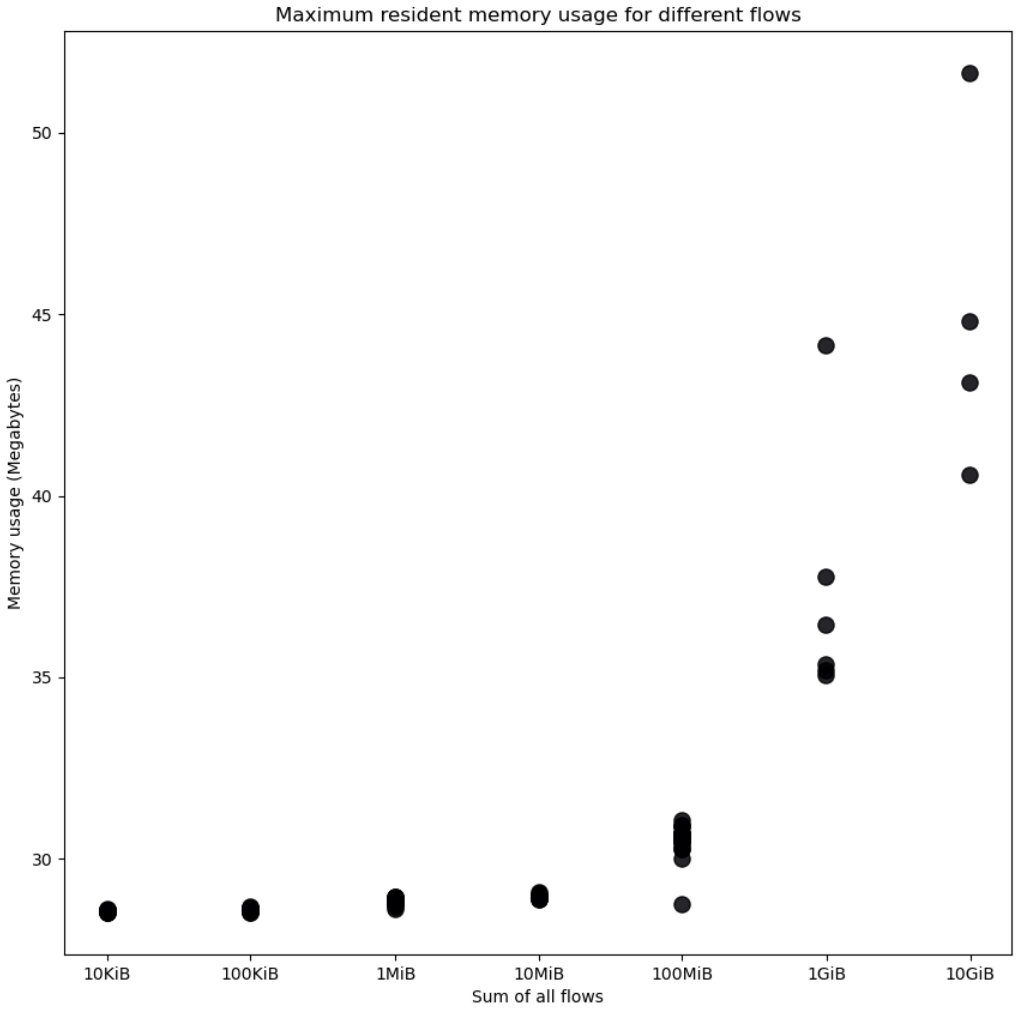


Figure 5.7: Maximum recorded resident memory usage in bytes, by the parsing part only. Each dot represents a single run of the program; the x-axis represents the total amount of traffic that the program was exposed to, in bytes.

6
Conclusions

Internet protocols have radically changed in the past years. Once thought to be irreplaceable and a panacea to any application requirement, TCP is not going anywhere soon and will remain at the foundation of the Internet for many years to come. However, the push that QUIC has received from both the IETF and the main browser vendors is not negligible. With the release of HTTP/3 in early 2022, QUIC is no longer a niche experimental protocol: it now powers 25% of the world HTTP traffic [94], and its popularity will only continue to grow. Therefore, research on efficient, scalable, and functional monitoring solutions is becoming more critical than ever.

Our proposed solution has shown promising results for monitoring QUIC flows, highlighting its efficient parsing, data gathering, and alert emission capabilities. Despite some initial concerns about our approach's accuracy and speed, we collected significant data through our monitoring tool, with overall low memory requirements and stable CPU usage. We, however, believe that the work is far from complete: we identified other potential improvements that, while beyond the scope of this project, would significantly boost the performance of the monitoring module.

First of all, both Quiche and QWIC were running as single-core applications. This meant that the application could not handle the load at a certain threshold and would either enter congestion mode (for Quiche) or analyse packets at a delayed rate (QWIC). We, therefore, decided to limit receiver windows and flow control limits. Our tests recorded bandwidths up to 100 Mbps, which is a reasonable upper limit for a testing scenario but is far from the 10 Gbps and greater speeds observed in data centres.

Secondly, using virtual machines and containers as clients meant that latency could only be simulated, thus lacking the unpredictability and variability of real network conditions. Although we experimented with traffic shapers to further restrict bandwidth and add artificial latency, those limitations did not significantly impact the proposed solution's performance. However, we believe that testing the solution in real networks, with actual clients located far away from the server, would have led to more insights on this aspect. For example, the RTT measurements — which were always predictable in our scenarios — would greatly vary when transferred on lossy or high-latency networks.

Thirdly, some algorithms and heuristics used in the solution are very naïve and could see considerable improvement. For example, the RTT measurement — which uses the algorithm suggested by the QUIC authors themselves — is relatively reliable but suffers from some degenerate cases which hamper its accuracy. Namely, networks with a very high packet loss could cause packet reordering, leading to incorrect measurements. There has been some research in this field [7, 59], which suggests some alternative approaches to measuring RTT. Another part of the program which could see some further optimisations is the acknowledgement tracking, which has been repeatedly optimised yet is still very resource intensive.

Finally, the program was written in pure Python, but this was mainly done for simplicity and speed of prototyping. We believe there could be two different approaches to optimising the performance of the code. At the current state, the program's packet sniffing is language agnostic and relies on either eBPF or `libpcap`. Thus, the program could be rewritten from scratch in a compiled language such as C or Rust while maintaining the `libpcap` sniffing module. However, a very interesting approach would be switching to a pure eBPF implementation, using it both for filtering, parsing, and data extraction. eBPF's high speed and near-zero overhead would make it a viable choice to replace performance-critical parts of the program. Recent developments in eBPF [36, 62, 63] have introduced support for a C frontend and *compile once, run anywhere* capabilities, which would make it possible to compile the program once and deploy it on multiple datacentre machines with little management overhead.

With research on QUIC yet in its relative infancy, we believe that soon, as the HTTP/3 adoption

and the development of QUIC tools increase, a better understanding of the protocol will come to light soon. It will allow new deployment strategies, particularly in data centres, where QUIC is currently lacking. We hope that this, in turn, will further bolster research on monitoring solutions for QUIC flows and will allow the creation of efficient and lightweight solutions for tenants and operators alike.

Bibliography

[1] aiortc, "aioQUIC," aiortc, May 2022. [Online]. Available: https://github.com/aiortc/aioquic

[2] N. Banks, "QUIC Disable Encryption," Internet Engineering Task Force, Internet Draft draft-banks-quic-disable-encryption-00, Aug. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-banks-quic-disable-encryption-00

[3] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," Internet Engineering Task Force, Request for Comments RFC 7540, May 2015. [Online]. Available: https://datatracker.ietf.org/doc/rfc7540

[4] J. Berg, F. Ruffy, K. Nguyen, N. Yang, T. Kim, A. Sivaraman, R. Netravali, and S. Narayana, "Snicket: Query-Driven Distributed Tracing," in *Proc. Twent. ACM Workshop Hot Top. Netw.* Virtual Event United Kingdom: ACM, Nov. 2021, pp. 206–212. [Online]. Available: https://dl.acm.org/doi/10.1145/3484266.3487393

[5] M. Bishop, "HTTP/3 and QUIC: Past, Present, and Future," Jun. 2021. [Online]. Available: https://securityboulevard.com/2021/06/http-3-and-quic-past-present-and-future/

[6] ——, "HTTP/3," Internet Engineering Task Force, Request for Comments RFC 9114, Jun. 2022. [Online]. Available: https://datatracker.ietf.org/doc/rfc9114

[7] F. Bulgarella, "QUIC Performance Measurement," Ph.D. dissertation, Politecnico di Torino, 2019. [Online]. Available: https://webthesis.biblio.polito.it/10904/1/tesi.pdf

[8] E. Chatzoglou, V. Kouliaridis, G. Karopoulos, and G. Kambourakis, "Revisiting QUIC attacks: A comprehensive review on QUIC security and a hands-on study," In Review, Preprint, Jul. 2022. [Online]. Available: https://www.researchsquare.com/article/rs-1676730/v1

[9] Cilium, "BPF and XDP Reference Guide — Cilium 1.11.4 documentation," 2022. [Online]. Available: https://docs.cilium.io/en/stable/bpf/

[10] Cloudflare, "Cloudflare/quiche," Cloudflare, Jul. 2022. [Online]. Available: https://github.com/cloudflare/quiche

[11] A. Concordia, "QUIC protocol from the monitoring perspective," 2020. [Online]. Available: https://www.concordia-h2020.eu/blog-post/quic-protocol-from-the-monitoring-perspective/

[12] CSO Online, "6 ways HTTP/3 benefits security (and 7 serious concerns)," Jun. 2020. [Online]. Available: https://www.csoonline.com/article/3564253/6-ways-http-3-benefits-security-and-7-serious-concerns.html

[13] DDOS, "Over 75% of Facebook's traffic uses QUIC and HTTP/3 • InfoTech News," Oct. 2020. [Online]. Available: https://meterpreter.org/over-75-of-facebooks-traffic-uses-quic-and-http-3/

[14] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, "Pluginizing QUIC," in *Proc. ACM Spec. Interest Group Data Commun.* Beijing China: ACM, Aug. 2019, pp. 59–74. [Online]. Available: https://dl.acm.org/doi/10.1145/3341302.3342078

[15] Q. De Coninck and Piraux, Maxime, "PQUIC," PQUIC, Jul. 2022. [Online]. Available: https://github.com/p-quic/pquic

[16] L. Decker, "QUIC & The Dead: Which of the Most Common IDS/IPS Tools Can Best Identify QUIC Traffic?" Tech. Rep., 2021.

[17] A. Delignat-Lavaud, J. Lallemand, J. Protzenko, and B. Parno, "A Security Model & Verified Implementation of the QUIC Record Layer," 2016.

[18] M. Duke, "QUIC Version 2," Internet Engineering Task Force, Internet Draft draft-ietf-quic-v2-04, Jun. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-v2

[19] ——, "QUIC Will Eat the Internet," Feb. 2022. [Online]. Available: https://www.f5.com/company/blog/quic-will-eat-the-internet

[20] ebpf.io, "What is eBPF? An Introduction and Deep Dive into the eBPF Technology," Mar. 2022. [Online]. Available: https://ebpf.io/what-is-ebpf/

[21] Facebook, "Facebookincubator/mvfst," Meta Incubator, Jul. 2022. [Online]. Available: https://github.com/facebookincubator/mvfst

[22] R. T. Fielding, M. Nottingham, and J. Reschke, "HTTP Caching," Internet Engineering Task Force, Request for Comments RFC 9111, Jun. 2022. [Online]. Available: https://datatracker.ietf.org/doc/rfc9111

[23] ——, "HTTP Semantics," Internet Engineering Task Force, Request for Comments RFC 9110, Jun. 2022. [Online]. Available: https://datatracker.ietf.org/doc/rfc9110

[24] ——, "HTTP/1.1," Internet Engineering Task Force, Request for Comments RFC 9112, Jun. 2022. [Online]. Available: https://datatracker.ietf.org/doc/rfc9112

[25] M. Fleming, "An introduction to the BPF Compiler Collection [LWN.net]," Dec. 2017. [Online]. Available: https://lwn.net/Articles/742082/

[26] ——, "A thorough introduction to eBPF [LWN.net]," Dec. 2017. [Online]. Available: https://lwn.net/Articles/740157/

[27] ——, "Some advanced BCC topics [LWN.net]," Feb. 2018. [Online]. Available: https://lwn.net/Articles/747640/

[28] ——, "Using user-space tracepoints with BPF [LWN.net]," May 2018. [Online]. Available: https://lwn.net/Articles/753601/

[29] T. C. i. S. Francisco, "Google QUIC-ly left privacy behind in its quest for a speedier internet, boffins find," Jan. 2021. [Online]. Available: https://www.theregister.com/2021/01/30/quic_fingerprinting_flaw/

[30] E. Gagliardi and O. Levillain, "Analysis of QUIC Session Establishment and Its Implementations," in *Information Security Theory and Practice*, M. Laurent and T. Giannetsos, Eds. Cham: Springer International Publishing, 2020, vol. 12024, pp. 169–184. [Online]. Available: http://link.springer.com/10.1007/978-3-030-41702-4_11

[31] D. P. García, "The eXpress Data Path - Unweaving the web," 2019, January 10. [Online]. Available: https://blogs.igalia.com/dpino/2019/01/10/the-express-data-path/

[32] K. Y. Gbur and F. Tschorsch, "A QUIC(K) Way Through Your Firewall?" Jul. 2021, comment: 7 pages, 8 figures. [Online]. Available: http://arxiv.org/abs/2107.05939

[33] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data Plane Performance Diagnosis of TCP," in *Proc. Symp. SDN Res.* Santa Clara CA USA: ACM, Apr. 2017, pp. 61–74. [Online]. Available: https://dl.acm.org/doi/10.1145/3050220.3050228

[34] A. Ghedini, "Even faster connection establishment with QUIC 0-RTT resumption," Nov. 2019. [Online]. Available: http://blog.cloudflare.com/even-faster-connection-establishment-with-quic-0-rtt-resumption/

[35] D. Green, "Pyshark," Aug. 2022. [Online]. Available: https://github.com/KimiNewt/pyshark

[36] B. Gregg, "Performance Wins with eBPF: Getting Started (2021)," 2021. [Online]. Available: https://www.slideshare.net/brendangregg/performance-wins-with-ebpf-getting-started-2021

[37] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in *Proc. 2018 Conf. ACM Spec. Interest Group Data Commun.* Budapest Hungary: ACM, Aug. 2018, pp. 357–371. [Online]. Available: https://dl.acm.org/doi/10.1145/3230543.3230555

[38] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, "QUIC (draft 00): A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Internet Draft draft-hamilton-quic-transport-protocol-01, Oct. 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-hamilton-quic-transport-protocol

[39] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast Connectivity Recovery Entirely in the Data Plane," *Proc. 16th USENIX Symp. Networked Syst. Des. Implement. NSDI '19*, p. 17, 2019.

[40] HTTP3 Explained, "Comparison with HTTP/2," 2020. [Online]. Available: https://http3-explained.haxx.se/en/h3/h3-h2

[41] igalia.com, "A brief introduction to XDP and eBPF - Unweaving the web," 2019. [Online]. Available: https://blogs.igalia.com/dpino/2019/01/07/introduction-to-xdp-and-ebpf/

[42] IOVisor, "BPF Compiler Collection (BCC)," IO Visor Project, May 2022. [Online]. Available: https://github.com/iovisor/bcc

[43] J. Iyengar and I. Swett, "QUIC Acknowledgement Frequency," Internet Engineering Task Force, Internet Draft draft-ietf-quic-ack-frequency-01, Oct. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-ack-frequency

[44] ——, "QUIC Loss Detection and Congestion Control," Internet Engineering Task Force, Request for Comments RFC 9002, May 2021. [Online]. Available: https://datatracker.ietf.org/doc/rfc9002

[45] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Request for Comments RFC 9000, May 2021. [Online]. Available: https://datatracker.ietf.org/doc/rfc9000

[46] Jim Roskind, "QUIC: Design Document and Specification Rationale," 2012. [Online]. Available: https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit?usp=embed_facebook

[47] M. Kühlewind and B. Trammell, "Applicability of the QUIC Transport Protocol," Internet Engineering Task Force, Internet Draft draft-ietf-quic-applicability-16, Apr. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-applicability

[48] ——, "Manageability of the QUIC Transport Protocol," Internet Engineering Task Force, Internet Draft draft-ietf-quic-manageability-16, Apr. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-manageability

[49] I. Kunze, C. Sander, K. Wehrle, and J. Rüth, "Tracking the QUIC spin bit on Tofino," in *Proc. 2021 Workshop Evol. Perform. Interoperability QUIC*, ser. EPIQ '21. New York, NY, USA: Association for Computing Machinery, Dec. 2021, pp. 15–21. [Online]. Available: https://doi.org/10.1145/3488660.3493804

[50] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," in *Proc. Conf. ACM Spec. Interest Group Data Commun.* Los Angeles CA USA: ACM, Aug. 2017, pp. 183–196. [Online]. Available: https://dl.acm.org/doi/10.1145/3098822.3098842

[51] Linux contributors, "Tshark(1): Dump/analyze network traffic - Linux man page," 2022. [Online]. Available: https://linux.die.net/man/1/tshark

[52] H. Liu, "Tc/BPF and XDP/BPF," Mar. 2019. [Online]. Available: https://liuhangbin.netlify.com/post/ebpf-and-xdp/

[53] S. Magnani, "eBPF_TrafficAnalyzer," Apr. 2022. [Online]. Available: https://github.com/s41m0n/eBPF_TrafficAnalyzer

[54] ——, "IDCAS - Intrusion Detection and Counter Attack System," Jan. 2022. [Online]. Available: https://github.com/s41m0n/idcas

[55] ——, "IPCAS - Intrusion Prevention and Counter Attack System," Feb. 2022. [Online]. Available: https://github.com/s41m0n/ipcas

[56] R. Marx, W. Lamotte, and P. Quax, "Visualizing QUIC and HTTP/3 with qlog and qvis," in *Proc. SIGCOMM 20 Poster Demo Sess.*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 42–43. [Online]. Available: https://doi.org/10.1145/3405837.3412356

[57] R. Marx, M. Piraux, P. Quax, and W. Lamotte, "Debugging QUIC and HTTP/3 with qlog and qvis," in *Proc. Appl. Netw. Res. Workshop*, ser. ANRW '20. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 58–66. [Online]. Available: https://doi.org/10.1145/3404868.3406663

[58] ——, "Qvis: Tools and visualizations for QUIC and HTTP/3," 2020. [Online]. Available: https://qvis.quictools.info/#/files

[59] S. Massano, "A new One-Way Delay measurement system for QUIC protocol," Ph.D. dissertation, Politecnico di Torino, Jul. 2021.

[60] Microsoft, "Microsoft/msquic," Microsoft, Jul. 2022. [Online]. Available: https://github.com/microsoft/msquic

[61] K. Moriarty, "Network Monitoring is Dead... What Now? TLS, QUIC and Beyond," Jun. 2018. [Online]. Available: https://labs.ripe.net/author/kathleen_moriarty/network-monitoring-is-dead-what-now-tls-quic-and-beyond/

[62] A. Nakryiko, "BPF CO-RE (Compile Once – Run Everywhere)," Feb. 2020. [Online]. Available: https://nakryiko.com/posts/bpf-portability-and-co-re/

[63] ——, "HOWTO: BCC to libbpf conversion · BPF," Feb. 2020. [Online]. Available: https://facebook.github.io/bpf/blog/2020/02/20/bcc-to-libbpf-howto-guide.html

[64] M. Nguyen, C. Timmerer, S. Pham, D. Silhavy, and A. C. Begen, "Take the red pill for H3 and see how deep the rabbit hole goes," in *Proc. 1st Mile-High Video Conf.*, ser. MHV '22. New York, NY, USA: Association for Computing Machinery, Mar. 2022, pp. 7–12. [Online]. Available: https://doi.org/10.1145/3510450.3517302

[65] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," Internet Engineering Task Force, Request for Comments RFC 2616, Jun. 1999. [Online]. Available: https://datatracker.ietf.org/doc/rfc2616

[66] K. Oku and J. Iyengar, "Can QUIC match the computational efficiency of TCP? Our research says yes." Apr. 2020. [Online]. Available: https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency

[67] L. Pardue and D. Belson, "HTTP RFCs have evolved: A Cloudflare view of HTTP usage trends," Jun. 2022. [Online]. Available: http://blog.cloudflare.com/cloudflare-view-http3-usage/

[68] T. Pauly, E. Kinnear, and D. Schinazi, "An Unreliable Datagram Extension to QUIC," Internet Engineering Task Force, Internet Draft draft-ietf-quic-datagram-01, Aug. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-datagram-01

[69] M. Piraux, O. Bonaventure, and A. Masputra, "Tunneling Internet protocols inside QUIC," Internet Engineering Task Force, Internet Draft draft-piraux-quic-tunnel-03, Aug. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-piraux-quic-tunnel-03

[70] J. Postel, "Transmission Control Protocol," Internet Engineering Task Force, Request for Comments RFC 793, Sep. 1981. [Online]. Available: https://datatracker.ietf.org/doc/rfc793

[71] Private Octopus, "Picoquic," Private Octopus, Jul. 2022. [Online]. Available: https://github.com/private-octopus/picoquic

[72] Rakesh Seal, "Looking Into QUIC Packets in your Network — Keysight Blogs," Jul. 2021. [Online]. Available: https://blogs.keysight.com/blogs/tech/nwvs.entry.html/2021/07/17/looking_into_quicpa-pUtF.html

[73] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet Engineering Task Force, Request for Comments RFC 8446, Aug. 2018. [Online]. Available: https://datatracker.ietf.org/doc/rfc8446

[74] J. Roskind, "Experimenting with QUIC," 2013. [Online]. Available: https://blog.chromium.org/2013/06/experimenting-with-quic.html

[75] J. Rüth, K. Wolsing, K. Wehrle, and O. Hohlfeld, "Perceiving QUIC: Do users notice or even care?" in *Proc. 15th Int. Conf. Emerg. Netw. Exp. Technol.*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, Dec. 2019, pp. 144–150. [Online]. Available: https://doi.org/10.1145/3359989.3365416

[76] D. Saif, C.-H. Lung, and A. Matrawy, "An Early Benchmark of Quality of Experience Between HTTP/2 and HTTP/3 using Lighthouse," *ArXiv200401978 Cs*, Oct. 2020. [Online]. Available: http://arxiv.org/abs/2004.01978

[77] A. Saverimoutou, B. Mathieu, and S. Vaton, "Which secure transport protocol for a reliable HTTP/2-based web service: TLS or QUIC?" in *2017 IEEE Symp. Comput. Commun. ISCC*. Heraklion, Greece: IEEE, Jul. 2017, pp. 879–884. [Online]. Available: http://ieeexplore.ieee.org/document/8024637/

[78] SecDev, "Scapy," SecDev, Aug. 2022. [Online]. Available: https://github.com/secdev/scapy

[79] R. Shade and M. Warres, "HTTP/2 Semantics Using The QUIC Transport Protocol," Internet Engineering Task Force, Internet Draft draft-shade-quic-http2-mapping-00, Jul. 2016, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-shade-quic-http2-mapping

[80] U. C. Shah, "Flow-based Analysis of QUIC Protocol," Ph.D. dissertation, Masaryk University, 2018.

[81] A. Srivastava, "Performance Analysis of QUIC Protocol under Network Congestion," Ph.D. dissertation, Worcester Polytechnic Institute, 2017.

[82] E. Sy, C. Burkert, H. Federrath, and M. Fischer, "A QUIC Look at Web Tracking," *Proc. Priv. Enhancing Technol.*, vol. 2019, no. 3, pp. 255–266, Jul. 2019. [Online]. Available: https://www.sciendo.com/article/10.2478/popets-2019-0046

[83] M. Thomson, "QUIC Security Overview," Feb. 2020. [Online]. Available: https://raw.githubusercontent.com/chris-wood/NDSS20-QUIPS/master/slides/quips2020_security_overview.pdf

[84] ——, "Version-Independent Properties of QUIC," Internet Engineering Task Force, Request for Comments RFC 8999, May 2021. [Online]. Available: https://datatracker.ietf.org/doc/rfc8999

[85] ——, "Greasing the QUIC Bit," Internet Engineering Task Force, Internet Draft draft-ietf-quic-bit-grease-03, May 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-bit-grease

[86] M. Thomson and C. Benfield, "HTTP/2," Internet Engineering Task Force, Request for Comments RFC 9113, Jun. 2022. [Online]. Available: https://datatracker.ietf.org/doc/rfc9113

[87] M. Thomson and S. Turner, "Using TLS to Secure QUIC," Internet Engineering Task Force, Request for Comments RFC 9001, May 2021. [Online]. Available: https://datatracker.ietf.org/doc/rfc9001

[88] K. K. N. Tiwari, "Denial of Service attack using Slowloris," *Int. Res. J. Eng. Technol. IRJET*, vol. 07, no. 07, p. 7, 2020.

[89] B. Trammell and M. Kühlewind, "The QUIC Latency Spin Bit," Internet Engineering Task Force, Internet Draft draft-ietf-quic-spin-exp-01, Oct. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-spin-exp-01

[90] ——, "The Wire Image of a Network Protocol," Internet Engineering Task Force, Request for Comments RFC 8546, Apr. 2019. [Online]. Available: https://datatracker.ietf.org/doc/rfc8546

[91] B. Trammell, P. D. Vaere, R. Even, G. Fioccola, T. Fossati, M. Ihlar, A. Morton, and S. Emile, "Adding Explicit Passive Measurability of Two-Way Latency to the QUIC Transport Protocol," Internet Engineering Task Force, Internet Draft draft-trammell-quic-spin-03, May 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-trammell-quic-spin

[92] M. Trevisan, D. Giordano, I. Drago, and A. S. Khatouni, "Measuring HTTP/3: Adoption and Performance," *2021 19th Mediterr. Commun. Comput. Netw. Conf. MedComNet*, pp. 1–8, Jun. 2021. [Online]. Available: http://arxiv.org/abs/2102.12358

[93] J. R. Vacca, Ed., *Computer and Information Security Handbook*, ser. The Morgan Kaufmann Series in Computer Security. Amsterdam ; Boston : Burlington, MA: Elsevier ; Morgan Kaufmann, 2009.

[94] w3techs, "Usage Statistics of HTTP/3 for Websites, July 2022," 2022. [Online]. Available: https://w3techs.com/technologies/details/ce-http3

[95] Wikipedia contributors, "Deep packet inspection," *Wikipedia*, Aug. 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Deep_packet_inspection&oldid=1102354735

[96] ——, "Head-of-line blocking," *Wikipedia*, Apr. 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Head-of-line_blocking&oldid=1083849253

[97] ——, "HTTP/3," *Wikipedia*, Jul. 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=HTTP/3&oldid=1097755240

[98] ——, "QUIC," *Wikipedia*, Jul. 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=QUIC&oldid=1097684976

[99] ——, "Transmission Control Protocol," *Wikipedia*, Jul. 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Transmission_Control_Protocol&oldid=1097267723

[100] C. Wiltz, "How Facebook is bringing QUIC to billions," Oct. 2020. [Online]. Available: https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/

[101] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, "Making QUIC Quicker With NIC Offload," in *Proc. Workshop Evol. Perform. Interoperability QUIC*. Virtual Event USA: ACM, Aug. 2020, pp. 21–27. [Online]. Available: https://dl.acm.org/doi/10.1145/3405796.3405827

[102] A. Yu, "Benchmarking QUIC," Jul. 2020. [Online]. Available: https://medium.com/@the.real.yushuf/benchmarking-quic-1fd043e944c7

[103] A. Yu and T. A. Benson, "Dissecting Performance of Production QUIC," in *Proc. Web Conf. 2021*. Ljubljana Slovenia: ACM, Apr. 2021, pp. 1157–1168. [Online]. Available: https://dl.acm.org/doi/10.1145/3442381.3450103

[104] P. Zhan, L. Wang, and Y. Tang, "Website fingerprinting on early QUIC traffic," *Computer Networks*, vol. 200, p. 108538, Dec. 2021, comment: This work has been accepted by Elsevier Computer Networks for publication. [Online]. Available: http://arxiv.org/abs/2101.11871

Attachment A Configuration flags

The following sections provide a brief description of the configuration flags available in the application. Figure A.1 shows the dependencies between the configuration options.

A.1 Miscellaneous flags

Flags in table A.1 are used to control the output of the application and do not affect the parsing of the packets. Printing flags are mutually exclusive, avoiding the printing of redundant or overlapping information.

Misc. flags	Requirements	Description
`SAVE_ADDITIONAL_INFO`	-	An additional data structure that saves parsed information about each packet.
`PRINT_INFO`	`SAVE_ADDITIONAL_INFO`	Prints information about the current packet.
`PRINT_FRAMES`	`SAVE_ADDITIONAL_INFO`	Prints information about the frames received in the current packet.
`PRINT_TIMES`	-	Prints the parsing time for each packet and its timestamp.
`PRINT_LOCAL_STATE`	-	Prints the currently saved state for the connection flow of the current packet.

Table A.1: Flags used for other purposes.

A.2 Parsing flags

Flags in table A.2 are used to control which parts of the packet are parsed or not as in section 4.3. By default, the header is parsed. Users can choose whether or not to parse the payload.

Parsing flags	Requirements	Description
-	-	Full parsing of the packet header.
`PARSE_FRAME`	-	Parses the packet payload recursively.

Table A.2: Flags used in the parsing of packets and frames.

A.3 Tracking flags

Flags in table A.3 are used to control what information is saved or not as in section 4.4. Each one of them depends on one or more parsing flags, depending on where the information is stored.

Tracking flags	Requirements	Description
`TRACK_PACKET_COUNT`	-	Tracks the packets in the shared 1-RTT/0-RTT space. This includes saving the number of packets per connection, the last time a packet was received, the average payload length, and more.
`TRACK_RTT`	-	Tracks the spin bit. This allows the monitoring of the round trip time (RTT) and its variance (RTTvar). The samples are updated with a moving average.
`TRACK_STREAMS`	`PARSE_FRAME`	Tracks `STREAM` frames. This includes their count, type, the average amount of data, the last time the frame was used, and the time since the stream was opened.
`TRACK_ACKS`	`PARSE_FRAME`	Tracks `ACK` frames, keeping track of which packets were acknowledged and which not.
`TRACK_GARBAGE`	`PARSE_FRAME`	Tracks how many corrupted or unparsable packets are received.
`TRACK_CONN_ID`	`PARSE_FRAME`	Tracks new connection IDs and stateless reset tokens for each new ID.

Table A.3: Flags used for tracking of data.

A.4 Alert flags

Flags in table A.4 are used in the application to enable alerts. Each flag is associated with a certain case study as outlined in chapter 5. Each case study requires one or more tracking flags to be able to generate alerts.

Alert flags	Requirements	Description
`ALERT_STREAM_COMMITMENT`	`TRACK_PACKET_COUNT`, `TRACK_STREAMS`, `TRACK_GARBAGE`	Alerts on opened streams with suspiciously high numbers, or many streams with low avg data.
`ALERT_SLOWLORIS`	`TRACK_PACKET_COUNT`, `TRACK_STREAMS`	Alerts on connections open but rarely used, or streams with few to no data.
`ALERT_STREAM_FRAGMENTATION`	`TRACK_STREAMS`, `TRACK_ACKS`	Alerts on attempts by attackers to send incomplete stream frames or acknowledgements.
`ALERT_PERFORMANCE_ANOMALY`	`TRACK_RTT`, `TRACK_ACKS`, `TRACK_GARBAGE`	Alerts on various performance-related metrics.

Table A.4: Flags used for alerting.

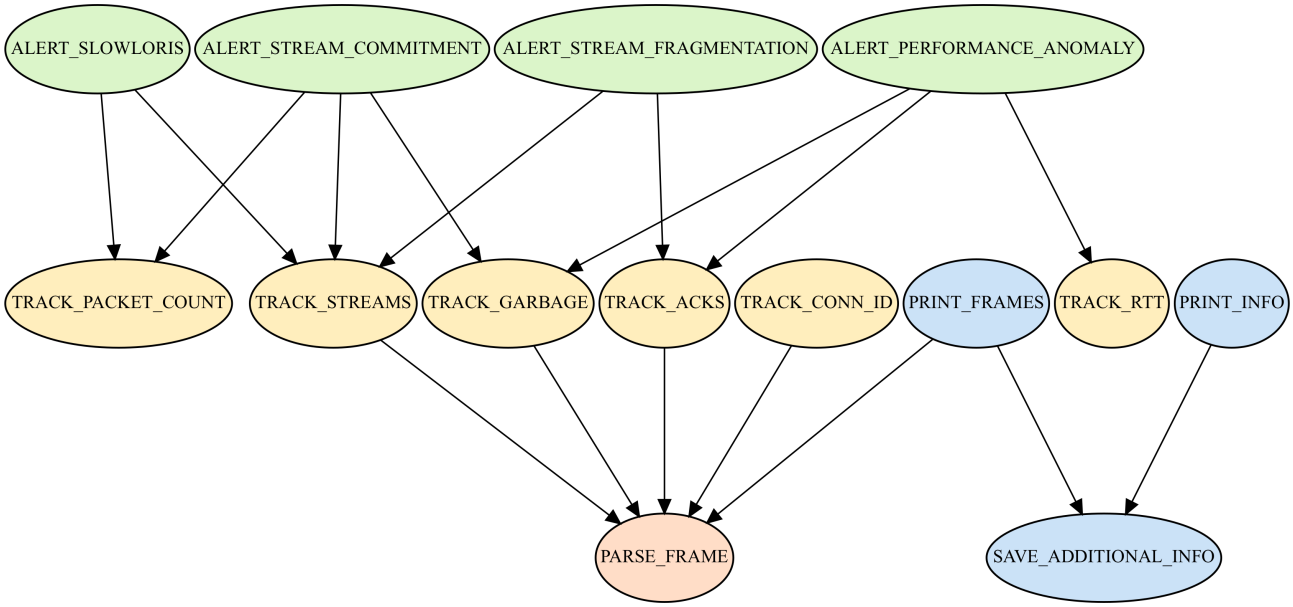


Figure A.1: The dependencies of the configuration flags.