FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	QUIC traffic dataset creation and analysis
Student:	Bc. Andrej Lukačovič
Supervisor:	Ing. Karel Hynek, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

Study QUIC protocol, its handshake procedures, and information exchanged during the handshake. Create an ipfixprobe [1] plugin capable of QUIC handshake information extraction from real-world traffic. In cooperation with the thesis supervisor, create a dataset of real QUIC traffic using the implemented ipfixprobe plugin. Analyze the dataset and provide insight about the current state of QUIC in the network.

[1] https://github.com/CESNET/ipfixprobe/

Master's thesis

QUIC TRAFFIC DATASET CREATION AND ANALYSIS

Bc. Andrej Lukačovič

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Karel Hynek, Ph.D.
May 9, 2024

Citation of this thesis: Lukačovič Andrej. *QUIC traffic dataset creation and analysis.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

I would like to express my deepest gratitude to my thesis supervisor, Ing. Karel Hynek, for his expertise, understanding, and patience. Due to our regular consultations, discussions, and meetings, I was able to finish this work. I would also like to thank my parents for giving me the opportunity to study at Czech Technical University. Additionally, big thanks go to my girlfriend, who was my biggest support during the evaluation of this thesis.

Declaration

In Prague on May 9, 2024 ....

Abstract

The primary focus of this thesis is the creation of a QUIC plugin for the IPFIXProbe flow exporting tool, which is capable of decrypting and obtaining various fields from the initial stage of QUIC communication. In a later stage, we also include an analysis that was performed on the dataset created by the plugin. Firstly a simple analysis was conducted on the captured data. Subsequently, we formulated a hypothesis suggesting that the User Agent and Server Name Indication pair can be used as a unique identifier. The dataset created and partially analyzed in this thesis has been published in Data-in-Brief Journal article.

Keywords QUIC, TLS Handshake, User Agent, Server Name Indication, Pattern Matching, Network Traffic Analysis, Cesnet, IPFIXProbe

Abstrakt

Hlavným cieľom tejto práce je vytvorenie QUIC plugin pre nástroj IPFIXProbe exportujúci sieťové toky, ktorý bude schopný obohatiť tieto toky o dešifrované dáta z počiatočnej fázy QUIC komunikácie. V neskoršej časti práce zároveň zahrnieme analýzu ktorá je prevedená na datasete ktorý vznikol na základe vytvoreného pluginu. Analýza sa v princípe delí do dvoch častí. Najskôr popíšeme jednoduché štatistky nazbieraných dát. Následne vyslovíme hypotézu ktorá navrhuje použitie User Agent a Server Name Indication pola ako unikátny identifikátor pre jednotlivé toky. Časť datasetu ktorý vznikol ako výsledok tejto práce je publikovaný ako článok v Data-in-Brief žurnály.

Klíčová slova QUIC, TLS Handshake, User Agent, Server Name Indication, Zhoda vzorov, Analýza sieťovej komunikácie, Cesnet, IPFIXProbe

Abbreviation List

AEAD	Authenticated encryption with associated data
AES	Advanced Encryption Standard
CERT	Certificate
CH	Client Hello
CV	Certificate Verify
DCID	Destination Connection ID
DPI	Deep Packet Inspection
ECB	Electronic codebook
EE	Encrypted Extensions
GCM	Galois/Counter Mode
HMAC	Hash-based message authentication code
HKDF	HMAC-based Extract-and-Expand Key Derivation Function
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IKM	Input keying material
IP	Internet Protocol
IPFIX	Internet Protocol Flow Information Export
ISP	Internet Service Provider
Mac	Macintosh
MAC	Medium Access Control
NAT	Network address translation
Nemea	Network Measurements Analysis
PRK	Pseudorandom key
RFC	Request for Comments
RTT	Round-trip time
SCID	Source Connection ID
SH	Server Hello
SNI	Server Name Indication
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UA	User Agent
XOR	Exclusive or

Introduction

The rapid evolution of the internet has necessitated the development of more efficient communication protocols to meet the growing demand for speed and security. One such advancement is the QUIC protocol, which represents a significant leap forward in reducing latency and improving the overall performance of web applications. Originally developed by Google and later adopted by the Internet Engineering Task Force (IETF) for further refinement, QUIC integrates enhanced features such as multiplexed connections and streamlined connection establishment, setting it apart from its predecessors like TCP and UDP.

QUIC also incorporates TLS in its network stack. In this context, TLS is responsible for establishing secrets, which are then utilized by QUIC to encrypt packets. Since the TLS protocol facilitates the exchange of these secrets, it is necessary for QUIC to include in its initial packets the messages required for the exchange of TLS protocol secrets. One such field is the Server Name Indication. Another field, that is not specific to the TLS protocol, is the User Agent. Although this field is marked as deprecated and appears unused in versions of Chrome higher than 100, it remains available in earlier versions. In network communications, where vast amounts of data are involved, analysis can be challenging due to data overlap and the sheer volume of data can affect the overall analysis. However, by limiting the scope of the data—specifically to only parts of network communication containing some data—we can achieve interesting results because we are working with more unique data.

As the Google Transparency Report states[1], more than 95% of communications transmitted via the QUIC protocol are encrypted. However, in this work, we introduce a QUIC plugin for the monitoring system IPFIXProbe, which can decrypt a portion of this encrypted communication based on publicly known secrets. After decryption, we gain access to all the initial information exchanged between the server and client when establishing a connection.

We believe that, despite this, the data are not particularly sensitive and do not contain any specific application-related information. Based on them, we can conduct a deeper analysis. In the final part of the work, we attempt to present a hypothesis: by combining Server Name Indication and User Agent, we can create a sort of unique identifier. This can then be used to uniquely identify network flows, ultimately aiming to track the activity and location of individual users.

First, in Chapter 1, we will explore the State of the Art of the QUIC Protocol. This chapter contains all the necessary information that will assist us in subsequent chapters. We will build a strong theoretical foundation on the fundamentals of the QUIC protocol, and we will also touch on the fundamentals of TLS, as these are also necessary.

In Chapter 2, we will go step-by-step through the implementation of the IPFIXProbe QUIC Plugin, which is capable of decrypting and extracting fields from the initial packets of the QUIC Protocol. At the end of the chapter, we will also mention the QUIC Dataset, which is later used in the analysis. Moreover, part of the dataset was also published in the Data in Brief journal.

[1] `https://transparencyreport.google.com/https/overview`

The last two chapters are dedicated to the initial analysis of the created dataset and a deeper dive into the analyzed data. Chapter 3 contains an analysis of the captured flows; we examined the contents of the QUIC Protocol initial packets and evaluated the contents of the Server Name Indication and User Agent fields. Chapter 4 goes a bit further; we make a hypothesis that the User Agent and Server Name Indication can be used to uniquely identify flows.

Chapter 1
State of the Art

1.1 QUIC Protocol

QUIC is a universal transport layer protocol, as described by Jim Roskind of Google in his 2012 article [1]. In 2013 [2], it was introduced to the public and the Internet Engineering Task Force[1] (IETF). Despite the name QUIC, which many might consider an acronym or abbreviation, this is not the case. The original proposal [1] by J. Roskind indeed named QUIC as *Quick UDP Internet Connections*, but later IETF documentation does not associate this name with an acronym. In 2015, a proposal was submitted [3] for standardization to the IETF community. In 2016, a research team was formed under the IETF, and the first *Internet-Draft* [4] was issued, which to this day, evolved into 35 versions. Finally, at the time of writing, there is an *Proposed Standard - RFC* called *QUIC: A UDP-Based Multiplexed and Secure Transport* [5]

Today, it is primarily associated with the term HTTP/3. The combination of HTTP/2 and QUIC led to the new designation HTTP/3 [6]. The QUIC protocol also experience adoption in the Web3 space, more precisely, currently one of the fastest blockchain talks about QUIC Protocol in its Networking Stack [2]

In Figure 1.1, we can see a comparison of HTTP/3, which employs QUIC in conjunction with UDP at the transport layer, and HTTP/2, which uses TCP. QUIC has several key advantages that can continue to advance it to the forefront over other transport protocols. These advantages will be described in a later Section. First, however, we need to explain a few basic terms and concepts.

1.1.1 Connection

As stated within the latest *RFC 9000* [5], a QUIC connection is a shared state between a client and a server.

Every connection initiates with a handshake phase, during which the two parties create a shared secret through the QUIC-TLS [8] *Cryptographic Handshake* protocol and decide on the application protocol. This handshake ensures both endpoints are ready to communicate and sets the parameters for the connection.

[1] https://www.ietf.org/
[2] https://www.helius.dev/blog/all-you-need-to-know-about-solana-and-quic

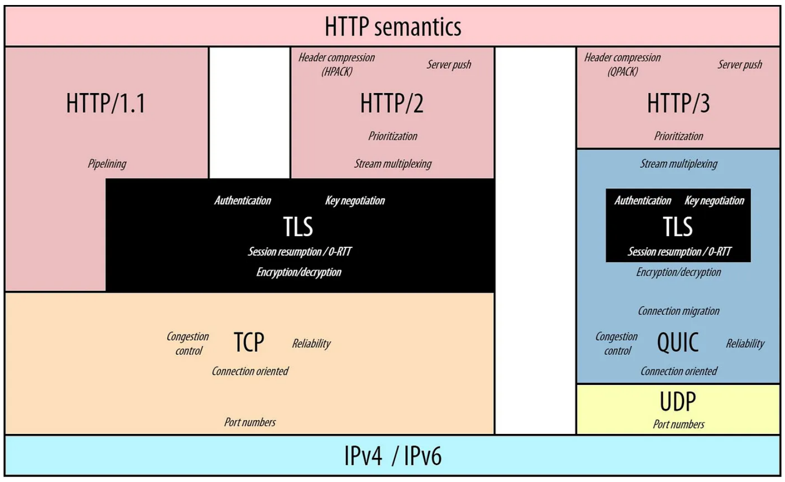

Figure 1.1 HTTP/1 and HTTP/2 stack compared to HTTP/3 stack [7]

1.1.1.1 Connection Identifiers

Each connection has a collection of identifiers, or IDs, each capable of identification the particular connection. These connection IDs are chosen independently by the endpoints; each endpoint picks the connection IDs that its counterpart utilizes. The primary function of a connection IDs is to ensure that changes in addressing at lower protocol layers (UDP, IP) do not cause packets for a QUIC connection to be delivered to the wrong endpoint. These Connection Identifiers are also, among other things, used during the derivation process of Initial Secrets, which are essential for extracting data from the Connection Initialization.

Connection IDs used within the QUIC protocol:

- Source Connection ID (SCID)
- Destination Connection ID (DCID)

1.1.2 Handshake

QUIC utilizes a combination of cryptographic [8] and transport [5] handshake to enhance security and to reduce the overhead involved in establishing a connection. For the transmission of critical cryptographic secrets during the initialization process, QUIC uses CRYPTO frames, which contain information from the TLS 1.3 protocol. The *RFC 9001* [8] describes that the TLS protocol offers two different types of handshakes for better security and efficiency in the connection setup.

- **1-RTT** (Round-trip time): A handshake in which the client (connection initiator) can send data only after completing one full cycle of exchanging the necessary initial information. The server can send data immediately after receiving the first *Initial* packet or even before the final message about the client's identity arrives, see Figure 1.2.
- **0-RTT** (Round-trip time): A handshake in which the client uses information provided by the server in a previous connection to send data right at the initialization of a new connection, see Figure 1.4. This way, the overhead for transmission, i.e., latency, is reduced, but it exposes a risk for so-called *replay attacks.*

1.1.2.1 1-RTT Hanshake

In Figure 1.2, we see the structure of how a typical 1-RTT handshake might look. During this process, it is possible to insert several QUIC packets into a UDP datagram, ensuring that the entire connection initialization can be performed by transmitting only 4 packets. Each line of the process consists of the type of packet being sent from the endpoint, the packet number, the frame located in the sent packet, and brief information about what the frame contains.

```
Client                                                      Server


Initial[0]: Crypto[CH] →


                                     ← (Version Negotiation)
                                                  ← (Retry)
                                Initial[0]: Crypto[SH] ACK[0]
                      Handshake[0]: Crypto[EE, CERT, CV, FIN]
                                    ← 1-RTT[0]: STREAM[data]


Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[0]: STREAM[data], ACK[0] →


                                         Handshake[1]: ACK[0]
           ← 1-RTT[1]: Handshake Done, STREAM[data], ACK[0]
```

Figure 1.2 Example 1-RTT Handshake [5]

1. In the first step, the client sends an *Initial packet.* In parentheses, we see the packet number, followed by a CRYPTO frame that contains the *Client Hello* message. This packet also includes the Destination Connection ID of the endpoint.
2. The server can respond in several ways:
   - *Version Negotiation* - this method of response indicates to the client whether the server supports the version of the QUIC protocol that the client is using. The size of the first packet determines whether the server will send a *Version Negotiation* packet or not.
   - *Retry* - After the server receives the *Client Hello* message, it may request address verification. This request is a special *Retry* packet type containing a *Token* that the server sends to the client. The client must then send the *Token* in every *Initial* packet in that connection.
   - It can proceed to establish the connection by sending the necessary information and parameters to the client, which are listed in RFC 8446 [9]. The TLS protocol takes care of these details, managing the exchange of keys, certificates, etc. Figure 1.3 shows an example of such communication. The QUIC protocol receives information from the TLS protocol, which handles the exchange of keys, certificates, and so on. Subsequently, it secures this information and sends it. At this point, the server can start sending application data to the client, these data are stored in 1-RTT packets and are fully secured.

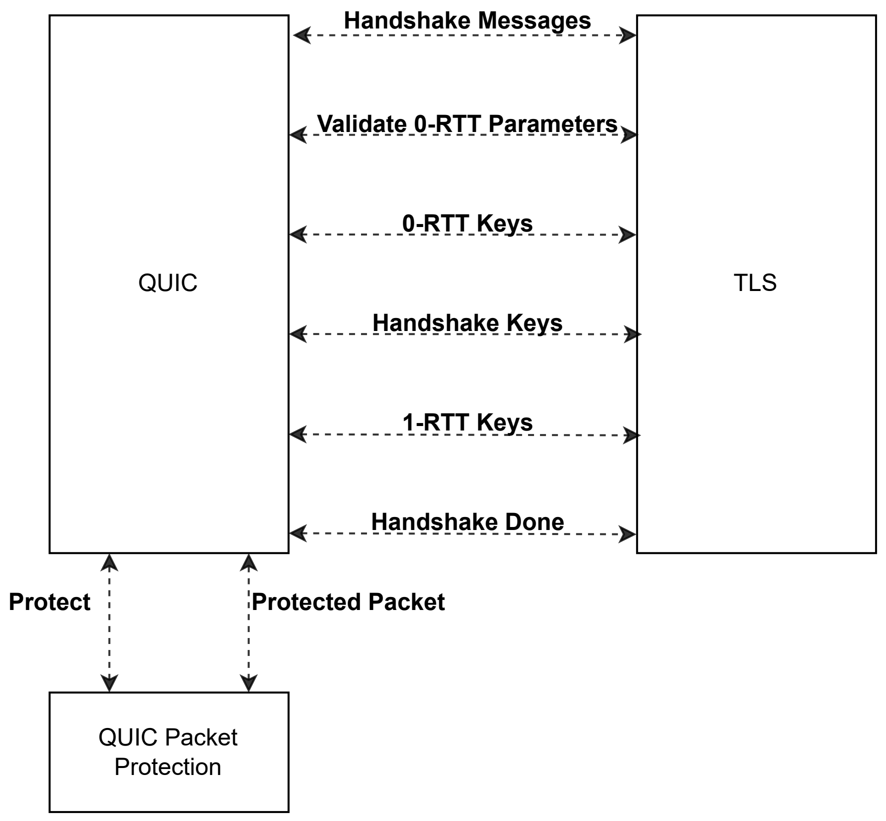

Figure 1.3 QUIC protocol and TLS information exchange [8]

3. The client can respond in several ways depending on the server's response:
   - If the server does not support the version used by the client in the first *Initial* packet, the client can either send a new *Initial* packet of a version that is supported by both endpoints or terminate the connection attempt because they do not support the same version.
   - If the client receives a *Retry* packet, it is necessary to use the *Token* contained in this packet in every subsequent *Initial* packet of that connection. Therefore, the client is expected to resend the *Initial* packet containing the *Token* that the server sent.
   - It can continue to establish the connection by sending the necessary information back to the server. In this case, these are mainly confirmation information for the *Handshake*. Also, at this stage of the connection initialization, the client can send application data stored in 1-RTT packets with strong security.
4. In the final step, the *Handshake* is almost complete. The server sends a confirmation and a message of completion. Likewise, it can send application data, as in point 3. After this response, the initialization is completed, and communication can freely proceed.

1.1.2.2 0-RTT Handshake

Figure 1.4 shows how a 0-RTT *Handshake* can proceed. The most significant difference compared to 1-RTT, which contributes to increased speed or reduced latency, is that the client can send data immediately after sending the *Initial* packet. This fact relies on parameters that were already agreed upon between the client and server in a previous connection. The data sent by the client during the connection initialization are contained in a 0-RTT packet type. The individual steps of establishing a connection are described in more detail in Section 1.1.2.1

```
Client                                                  Server


Initial[0]: Crypto[CH] →
0-RTT[0]: STREAM[data] →


                                      ← (Version Negotiation)
                                                    ← (Retry)
                                   Initial[0]: Crypto[SH] ACK[0]
                         Handshake[0]: Crypto[EE, CERT, CV, FIN]
                               ← 1-RTT[0]: STREAM[data], ACK[0]


Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[1]: STREAM[data], ACK[0] →


                                         Handshake[1]: ACK[0]
            ← 1-RTT[1]: Handshake Done, STREAM[data], ACK[1]
```

Figure 1.4 Example 0-RTT Handshake [5]

1.1.3 Packets

Endpoints using the QUIC protocol communicate within a connection using packets. Packets are processed and then inserted into UDP datagrams. They have a simple structure containing a header and one or more frames. The *RFC 9000* [5] describes that packets are divided based on those with long and short headers.

Packets with a long header are used during connection establishment.

- **Initial**: contain the first messages for connection initialization.
- **0-RTT**: in this context, it refers to a packet type, not a handshake type, carrying application data that are transferred during the initialization of a connection.
- **Retry**: contain a *Token* created by the server.
- **Handshake**: carry information related to the handshake, mainly shared secrets.
- **Version Negotiation**: used only by the server, they contain messages about the agreement on the used version.

Packets with a short header are designed for minimal overhead and are used after a connection is established and 1-RTT keys are available.

- **1-RTT**: in this context, it refers to a packet type, not a handshake type, it is the only type of packet that has a short header, used for data transmission after the connection is established (partially also during establishment, see Figure 1.2).

```
Initial Packet
    # Header
    Header Form (1) = 1,
    Fixed Bit (1) = 1,
    Long Packet Type (2) = 0,
    Reserved Bits (2),                        # Protected
    Packet Number Length (2),                 # Protected
    Version (32),
    Destination Connection ID Length (8),
    Destination Connection ID (0..160),
    Source Connection ID Length (8),
    Source Connection ID (0..160),
    Token Length (i),
    Token (..),
    Length (i),
    Packet Number (8..32),                    # Protected
    # Payload
    Protected Payload (0..24),                # Skipped Part
    Protected Payload (128),                  # Sampled Part
    Protected Payload (..),                   # Remained
```

Figure 1.5 *Initial* Packet with long header format [10]

The short header is implemented primarily to achieve the smallest possible overhead. Thus, after the connection has been initialized and the necessary information for data transmission has been agreed upon, it is no longer necessary to carry a large amount of information in the header see Figure 1.6. For the Long Header format in Figure 1.5, certain parts of the header are masked to maintain integrity and increase security. The method of masking is briefly described in Section 1.1.4.3.

In Table 1.1, we can see how *Long Header* Packet types are identified. A Version Negotiation packet is inherently not version specific. Upon receipt by a client, it will be identified as a Version Negotiation packet based on the Version field having a value of 0.

Table 1.1 Long Packet type fields [5]

Long Packet Type field	Packet Type
0x00	Initial
0x01	0-RTT
0x03	Retry
0x02	Handshake
-	Version Negotiation

1.1.4 Packet protection

In the preceding section, we explained that QUIC Packets are categorized into two groups based on the Header they employ. However, as shown in the *RFC 9000* [5], we can also divide the Packets based on the used Packet Protection.

As with TLS over TCP, QUIC protects packets with keys derived from the TLS handshake, using the AEAD algorithm [11] negotiated by TLS [8]. We will get into a more detailed expla-

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

Figure 1.6 *1-RTT* Packet with short header format [10]

nation of how the Packet Protection for *Initial* packets works and how are the *Initial* secrets derived in the Section 2.5 of Chapter 2.

However, to briefly introduce the logic, QUIC packets have varying protections depending on their type [8]:

- **Version Negotiation** packets have no cryptographic protection.
- **Retry** packets use *AEAD_AES_128_GCM* to provide protection against accidental modification and to limit the entities that can produce a valid Retry.
- **Initial** packets use *AEAD_AES_128_GCM* with keys derived from the Destination Connection ID field of the first Initial packet sent by the client.
- **All other** packets have strong cryptographic protections for confidentiality and integrity, using keys and algorithms negotiated by TLS.

In Figure 1.7, we can see the process of *AEAD_AES_128_GCM* encryption mechanism. An unencrypted packet is first divided into a header and data. Subsequently, an encryption algorithm is applied, in this case *Advanced Encryption Standard with Galois/Counter Mode* (AES-GCM). Several input parameters are needed for encryption:

- The unprotected **Header** is used as *associated data* of AEAD mechanism, this ensures *integrity*, but not *confidentiality* of used input.
- The nonce is generated as the XOR of the **Packet Number** and the **Initialization Vector**.
- In general, the **key** is derived based on the cryptographic level into which the corresponding packet type falls. From Table 1.2, we can see that, in the QUIC Protocol there are multiple levels of cryptography [8]. However, in this work, it is not necessary, to handle different cryptographic levels. For the rest of this work, it is enough to consider that *Initial* Packets fall into the *Initial Secrets* level.

1.1.4.1 Initial secrets

As mentioned above, the main encryption algorithm used is *Authenticated Encryption with Associated Data* from Figure 1.7. The secrets used for encryption in this early stage of communication between the *Client* and *Server* are derived from publicly known data (i.e. from fields of the *Initial* Packet that are not protected with strong cryptographic encryption).

Table 1.2 Different cryptographic levels by Packet Type [8]

Packet Type	Encryption Keys	Packet Number Space
Initial	Initial secrets	Initial
0-RTT Protected	0-RTT	Application data
Handshake	Handshake	Handshake
Retry	Retry	-
Version Negotiation	-	-
Short Header	1-RTT	Application data

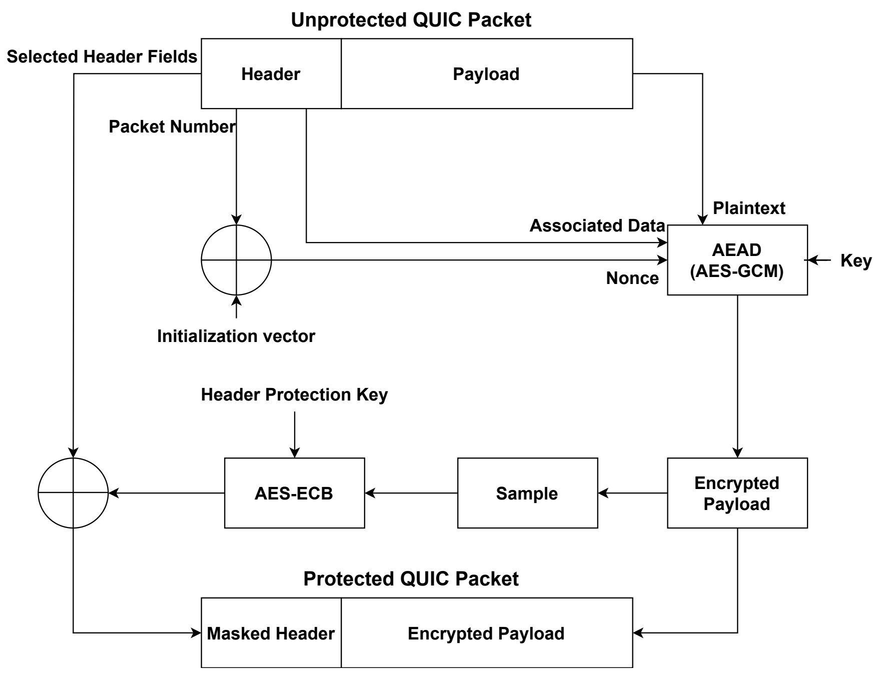

Figure 1.7 Authenticated Encryption with Associated Data [12]

The process of derivation secrets that are used in order to protect *Initial* Packets is as follows (the process is described in more detail in Section 2)

- Based on the *initial_salt*, constant defined within the protocol specification and the *Client Destination Connection ID*, the *initial_secret* is derived.
- Next, the *initial_secret* is used to derive *client_initial_secret* and *server_initial_secret*. Within this work, we focus only on *client_initial_secret* because we aim to revert *Packet Protection* of packets sent by the *Client*.
- Lastly, from the *client_initial_secret* the following secrets are derived:
  - Header Protection secret
  - Initialization Vector
  - Key used for the *AES-GCM*

According to [8], the derivation function for the derivation process of the above secrets is *HMAC-based Extract-and-Expand Key Derivation Function* (HKDF), more precisely in two steps:

- *HMAC-based Key Derivation Function-Extract*

- *HMAC-based Key Derivation Function-Expand-Label*

1.1.4.2 HMAC-based Key Derivation function

For the derivation process of *Initial Secrets*, the *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* [13] is used. This Function consists of two main phases:

- **Extract Phase** is designed to take an input keying material (IKM) and "extract" from it a fixed-length pseudorandom key (PRK). The extraction is typically done using an *Hash-based Message Authentication Code* (HMAC) function. The process involves combining the IKM with a salt using the HMAC.

- **Expand Phase** involves using the PRK as the HMAC key and taking an info string (which is a non-secret value that can be used to bind the derived keys to specific application contexts) and a length parameter (indicating the length of the output keying material required) as inputs. The process can produce a long stream of output key material by iteratively applying the HMAC function.

1.1.4.3 Header Protection

As we may see in Figure 1.5, some parts of the header are marked as *Protected*. This is due to the application of the Header Protection. Parts of QUIC headers, in particular the Packet Number and the Packet Number Length fields, are protected using a Header Protection key that is derived from the Initial Secret.

This derivation process is similar to the derivation process of Initialization Vector and AES-GCM Key, which was described in Section 1.1.4.1. Next, as we can see in Figure 1.7 part of the encrypted Packet Payload is used as an input into the *Advanced Encryption Standard* in *Electronic codebook (ECB)* mode encryption algorithm. Lastly, the output of the encryption algorithm is used in conjunction with certain header fields, applying the XOR operation to create the final protected header fields.

A more detailed description of the derivation process of Header Protection keying material and the application process of this keying material is examined within Section 2.5.3 and Section 2.6 respectively.

1.1.5 Frames

Frames are entities that contain the actual data. This data is not limited to application data; it can also include information necessary for encrypting messages, various connection state properties, and the like. Frames are carried in packets, and their size must be such that it corresponds to the packet size. Therefore, it is not possible for one frame to be divided and then sent in two packets. Simultaneously, multiple frames can be carried in a single packet [5].

In Figure 1.8, we see the classic structure of a QUIC Frame. The attributes include:

- *Frame Type*: indicates the type of frame. An example is provided in Table 1.3.
- *Type-Dependant Fields*: this attribute varies depending on the type of frame being used.

In Table 1.3, we see the types of frames that the QUIC protocol uses, according to the *RFC 9000* [5], the rules for their use are as follows:

- **PADDING:** Without significant use, it is used only for manipulating packet sizes, specifically for their increase.

- **PING:** serves to determine the availability of the communicating party.

```
Frame {
    Frame Type              (),
    Type-Dependant Fields   (...),
}
```

Figure 1.8 Generic Frame Layout

- **ACK:** ACK type frames contain acknowledgments of the receipt and processing of packets.
- **RESET_STREAM:** abrupt termination of stream data transmission.
- **STOP_SENDING:** the endpoint sends STOP_SENDING to inform the other communicating party that the received data will be discarded.
- **CRYPTO:** contains cryptographic handshake information.
- **NEW_TOKEN:** sent by the server to the client when it requires the client to use a new Token in initial packets.
- **STREAM:** contains stream application data.
- **MAX_DATA:** contains information about the maximum amount of data that can be sent in the entire connection.
- **MAX_STREAM_DATA:** contains information about the maximum amount of data that can be transferred in a single stream.
- **MAX_STREAMS:** inform the communicating node about the maximum number of streams that can be opened.
- **DATA_BLOCKED:** contains information that the sender could not send data due to a connection limitation (typically a maximum data limit).
- **STREAM_DATA_BLOCKED:** similar to the previous, but refers to a limitation at the stream data level.
- **STREAMS_BLOCKED:** similar to the previous, but refers to a limitation on the number of open streams.
- **NEW_CONNECTION_ID:** contains information about a new connection identifier for the communicating party.
- **RETIRE_CONNECTION_ID:** information that the communicating node will no longer use the connection identification number that was allocated by the other party.
- **PATH_CHALLENGE:** used in connection migration, tests the reachability of the other party.
- **PATH_RESPONSE:** response to PATH_CHALLENGE.
- **CONNECTION_CLOSE:** information about the closing or ending of the connection.
- **HANDSHAKE_DONE:** an empty frame without content, indicates confirmation from the server side of the handshake.

Table 1.3 Frame Types

Type Value	Frame Type
0x00	PADDING
0x01	PING
0x02-0x03	ACK
0x04	RESET_STREAM
0x05	STOP_SENDING
0x06	CRYPTO
0x07	NEW_TOKEN
0x08-0x0f	STREAM
0x10	MAX_DATA
0x11	MAX_STREAM_DATA
0x12-0x13	MAX_STREAMS
0x14	DATA_BLOCKED
0x15	STREAM_DATA_BLOCKED
0x16-0x17	STREAMS_BLOCKED
0x18	NEW_CONNECTION_ID
0x19	RETIRE_CONNECTION_ID
0x1a	PATH_CHALLENGE
0x1b	PATH_RESPONSE
0x1c-0x1d	CONNECTION_CLOSE
0x1e	HANDSHAKE_DONE

1.1.5.1 CRYPTO Frame

One of the Frame Types that is used in the QUIC Protocol is *CRYPTO* Frame. This frame is used for carrying TLS Handshake Messages. In Figure 1.9, we can see the structure of the CRYPTO Frame. This structure is necessary for correctly obtaining the TLS Handshake data that are transmitted in this type of Frame. In Section 2.8, we will use this structure to put together separated CRYPTO Frames correctly.

```
CRYPTO Frame {
    Type (i) = 0x06,
    Offset (I),
    Length (I),
    Crypto Data (..),
}
```

Figure 1.9 Structure of CRYPTO Frame

The Frame contains the following fields:

- *Offset*: Integer specifying the byte offset for the data in the CRYPTO frame.
- *Length*: Integer specifying the length of the Crypto Data field in the CRYPTO frame.
- *Crypto Data*: The cryptographic message data.

1.1.5.2 Out of Order Frames

Important information to note in terms of QUIC Frames is that the Packet Payload can contain multiple frames of the same type. This can result in a situation where, for example, CRYPTO frames are spread across the Packet Payload, even when the message corresponds to one *Client Hello* message. This means after the Packet Payload is decrypted with the QUIC Plugin described in Chapter 2, we should also consider that the CRYPTO frame containing desired *Client Hello* TLS message is not required to be located at the start of Packet Payload. Rather, the *Client Hello* message can be split into multiple CRYPTO frames, and these frames can be stored in the Payload out-of-order [5].

This feature also called *Chaos Protection* and was introduced in *QUICHE*[3]. The purpose of the feature is to reduce the likelihood of QUIC ossification due to middleboxes.

1.1.6 Stream

A *stream* is an ordered sequence of bytes of data that we want to transfer between end nodes [5]. It primarily concerns application data. Fundamentally, it is divided into two categories: unidirectional and bidirectional. A unidirectional *stream* serves for data transfer in one direction, thus from the initiator to the endpoint. Bidirectional allows for data transfer in both directions. A *stream* is carried in frames. A typical example of a STREAM frame can be seen in Figure 1.10. The first attribute is *stream type*; this value is not uniform because the values of the last 3 bits can change, and they define:

```
STREAM Frame {
    Type        () = 0x08..0x0f,
    Stream ID   (),
    [Offset     ()],
    [Length     ()],
    Stream data (..),
}
```

Figure 1.10 Structure of Stream Frame

- The rightmost bit (0x01), or the *FIN bit*, defines whether the current frame terminates the *stream*. If so, the final length is calculated as the sum of the *Length* variable and the *Offset*.
- The second bit from the right (0x02), or the *LEN bit*, determines whether the *Length* variable is present.
- The third bit from the right (0x04), the last variable, also known as the *OFF bit*, indicates whether the value of the *Offset* variable is present. If this bit is set to 0, meaning the *Offset* value is not present, it implies that this STREAM frame contains the first bytes of the given *stream*.
- *Stream ID*: this attribute contains a unique identifier. It serves to distinguish and assign the received *stream* data. Its use brings the advantage of *Stream multiplexing* (see Section 1.1.7).
- *Offset*: the value of the *Offset* variable indicates a certain position. Imagining the *stream* as a sequence of 0s and 1s, it's clear that this sequence can be too long to fit in a single frame. Therefore, we need to carry information about the index at which the currently transmitted

[3] `https://github.com/google/quiche`

data starts. This information is stored as an integer in the *Offset* variable, which can only take positive values and 0.

- *Length*: is the length of the data carried in the current STREAM frame.

1.1.7 Advantages of the QUIC Protocol

From the features described in this chapter, several advantages of the QUIC protocol emerge [10]:

- **Connection Migration**: Thanks to unique connection identifiers, rapid restoration of communication is possible, for example, in *NAT rebinding* situations [10].
- **TLS over QUIC**: The communication of the TLS protocol through QUIC enhances security and can reduce the latency of connection establishment.
- **Stream Multiplexing**: Each stream contains its unique identifier, which is processed on the server/client side. Thus, if an error occurs in one stream, the others can continue without problems.
- **Congestion Control**: QUIC allows setting data transfer limits, thus controlling congestion in individual connections. This principle can serve as protection when a server (or attacker) sends data faster than the client can process.
- **0-RTT**: Thanks to the 0-RTT handshake, the time to establish a connection is significantly reduced.
- **Encryption and Masking**: Header protection and pseudo-protection of Initial packets increase the complexity of eavesdropping. Even if the encryption algorithm uses visible parameters to derive keys, decryption requires considerable effort compared to the value of the information it provides.

1.2 QUIC-TLS

QUIC carries TLS handshake data in CRYPTO frames, each of which consists of a contiguous block of handshake data identified by an offset and length. Those frames are packaged into QUIC packets and encrypted under the current encryption level. As with TLS over TCP, once TLS handshake data has been delivered to QUIC, it is QUIC's responsibility to deliver it reliably. Each chunk of data that is produced by TLS is associated with the set of keys that TLS is currently using. If QUIC needs to retransmit that data, it must use the same keys even if TLS has already updated to newer keys [8]. In Figure 1.11, we can see how the layers of TLS and QUIC are structured.

1.2.1 Server Name Indication

Server Name Indication extension, which is contained inside the TLS was introduced in the *RFC 3546* [14]. As stated, because the TLS did not offer a method for a client to inform a server with the domain name it is attempting to connect to. It is beneficial for clients to share this information to enable secure connections to servers hosting multiple virtual servers on a single underlying network address.

The client's TLS *Client Hello* inside the CRYPTO Frame might include a Server Name Indication (SNI) extension, through which the client discloses the domain name it plans to connect to, enabling the server to offer a certificate corresponding to that name. When included, SNI details are accessible to unidirectional observers along the client-to-server path. In Figure 1.12 we can see how was the SNI Extension specified within the *RFC 3546* [14]. This structure will be particularly helpful during the extraction of the field in Chapter 2.

```
+--------------+--------------+ +-------------+
|     TLS      |     TLS      | |    QUIC     |
|  Handshake   |    Alerts    | | Applications|
|              |              | |  (h3, etc.) |
+--------------+--------------+-+-------------+
|                                             |
|                QUIC Transport               |
|   (streams, reliability, congestion, etc.)  |
|                                             |
+---------------------------------------------+
|                                             |
|            QUIC Packet Protection           |
|                                             |
+---------------------------------------------+
```

Figure 1.11 QUIC Layers and TLS [8]

```
struct {
    NameType name_type;
    select (name_type) {
        case host_name: HostName;
    } name;
} ServerName;
enum {
    host_name(0), (255)
} NameType;
opaque HostName<1..2^16-1>;
struct {
    ServerName server_name_list<1..2^16-1>
} ServerNameList;
```

Figure 1.12 Structure of Server Name Indication extension [14]

1.2.2 User Agent

As stated in the *RFC 7231* [15], The User-Agent field contains information about the user agent originating the request, which is often used by servers to help identify the scope of reported interoperability problems, to work around or tailor responses to avoid particular user agent limitations, and for analytics regarding browser or operating system use. In Figure 1.13, we can see an example of what the User Agent field can look like. We may notice that the field can contain information about the version of your web browser or operating system in both cases for desktop computers or mobile devices. This field, in terms of QUIC Protocol, has a simple structure, starting with the type, which is used to differentiate between other extensions, followed by the length of the User Agent field, and lastly, the content of the field.

During the Analysis, which is described in Chapters 3 and 4, we noticed that the Chrome versions contained in the User Agent field are always smaller than 100. We think that it is due to the fact that the User Agent field is marked as deprecated[4]. However, for the older devices that use older Chrome versions (9X.X.XXXX.XXX), the field is still present.

[4]`https://www.iana.org/assignments/quic/quic.xhtml`

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36
```

Figure 1.13 Example of the User-Agent field content

1.3 Network monitoring

Among the two most fundamental approaches to monitoring network communication are monitoring at the level of so-called network flows and deep packet analysis.

This work falls into both categories to some extent, as we first need to decipher the contents of *Initial* packets. This will help us obtain more information. These pieces of information are then used within the Network Monitoring tool called *IPFIXProbe* to enrich Network Flows with the extracted information.

1.3.1 Flow monitoring

Monitoring through network flows is becoming increasingly utilized, especially in high-speed networks with a large amount of transferred data. This method works by monitoring at the level of IP flows, thus exporting information from IP headers. As mentioned in RFC 7011 [16], "A flow is a set of packets passing through an observation point in the network during a certain time interval. All packets belonging to a specific flow have a set of common properties." These properties include header parameters, such as IP addresses, used ports, protocols, and others.

One of the protocols that deal with exporting IP flows is the *IP Flow Information Export* (IPFIX) [16]. Its official standard [17] was published in 2008 by the *Internet Engineering Task Force* community. The exported flow data can contain a large amount of information, as we see in [18].

1.3.1.1 IPFIXProbe

One of the well-known tools that is capable of exporting IP flows is IPFIXProbe [5].

This IP flow exporter is an open-source project capable of exporting IPFIX output data extended with various information by extension plugins. This software tool was originally known as flow_meter, a component of the NEMEA [6] system (the development started as a project of bachelor and master theses at the Czech universities in cooperation with CESNET). However, since the IPFIX ecosystem evolved and another open source project IPFIXcol2 became more flexible and efficient, the team has decided to make IPFIXProbe also more independent.

As stated the tool is capable of exporting IPFIX output data extended with various information by extension plugins. Let‘s discuss some of its basic fields.

1.3.2 Deep packet inspection

Deep packet inspection differs from the network flow monitoring method primarily in that it examines the content of packets directly. While flow-level monitoring only examines the content of the IP header, according to the article [19], deep packet inspection can also examine headers of higher layers. Among its advantages and disadvantages are:

- It examines packet data, thereby providing better information about the contents of packets and, thus, better monitoring results.

[5] `https://www.liberouter.org/high-speed-multithreaded-ip-flow-exporter-for-machine-learning/`
[6] `https://nemea.liberouter.org/`

Table 1.4 IPFIXProbe basic fields exported

Output field	**Type**	**Description**
DST_MAC	macaddr	destination MAC address
SRC_MAC	macaddr	source MAC address
DST_IP	ipaddr	destination IP address
SRC_IP	ipaddr	source IP address
BYTES	uint64	number of bytes in data flow (src to dst)
BYTE_REV	uint64	number of bytes in data flow (dst to src)
LINK_BIT_FIELD or ODID	uint64 or uint32	exporter identification
TIME_FIRST	time	first time stamp
TIME_LAST	time	last time stamp
PACKETS	uint32	number of packets in data flow (src to dst)
PACKETS_REV	uint32	number of packets in data flow (dst to src)
DST_PORT	uint16	transport layer destination port
SRC_PORT	uint16	transport layer source port
DIR_BIT_FIELD	uint8	bit field for determining outgoing/incoming traffic
PROTOCOL	uint8	transport protocol
TCP_FLAGS	uint8	TCP protocol flags (src to dst)
TCP_FLAGS_REV	uint8	TCP protocol flags (dst to src)

- It is demanding on performance, thereby reducing processing speed.
- If strong cryptographic encryption is present for the protocol, it is impossible to apply this method.
- If encryption keys are already known, it is very challenging to keep them up to date.
- Examining the contents of packets may infringe on user privacy.

Even though the QUIC protocol, with the underlying TLS, uses strong cryptographic encryption for transmitted packets, for certain limited types of packets, we can employ *Deep Packet Inspection*. As previously mentioned in Sections 1.1.4 and 1.1.4.1, the *Initial* packets are prone to DPI because the encryption used in this part of the *Handshake* can be reversed by an observer. Thus, we can say this approach is also utilized within our work.

Chapter 2

IPFIXProbe QUIC Plugin

To analyze the QUIC protocol and enrich the flows of QUIC communication, we created an IPFIXProbe plugin capable of extracting data from the protocol's *Handshake*.

More precisely, as described within the State of the Art, QUIC communication begins with the *Handshake*. The first packet of the *Handshake* from the Client side is called *Initial* and contains the *Client Hello*. As already mentioned, during this phase, the underlying QUIC-TLS [8] needs to transmit and agree on important data between the Server and Client to proceed with the communication. Among the many parameters transferred within the *Client Hello* are the *Server Name Indication* and the *User Agent*. The QUIC plugin contained within the IPFIXProbe module is capable of extracting these fields.

2.1 Variable-Length Integer Encoding

For plugin implementation purposes, it is worth noting that QUIC Packets and Frames commonly use a variable-length encoding for non-negative integer values. This encoding ensures that smaller integer values need fewer bytes to encode. The QUIC 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 [5].

Table 2.1 Summary of Variable Integer Encoding [5]

2MSB	Length	Usable Bits	Range
00	1	6	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

2.2 Version specific Initial Salt

In Section 1.1.4.1, we described that one of the inputs during the derivation process of the *Initial* Secrets is *initial_salt*. This constant is indeed defined within the QUIC Protocol specification. However, it may be worth mentioning that different QUIC draft versions can specify different values of the *initial_salt*. For example, at the time of writing, the latest *RFC 9000* [8] defines the *initial_salt* with the value of 0x38762cf7f55934b34d179ae6a4c80cadccbb7f0a. Older versions of QUIC Protocol, for example, *draft-ietf-quic-tls-32* [20] uses

0xafbfec289993d24c9e9786f19c6111e04390a899 for *initial_salt*. This fact is later on considered during the derivation process of *Initial* Secrets.

2.3 Initial Packet Filter

Once the packet lands in the detection module, we need to check if the packet contains QUIC Protocol data, additional check if the packet is of type *Initial* with the Long Header needs to be performed. This inspection is required since the detection module can process a huge load of network traffic, so to increase the effectiveness of the plugin, unnecessary packets need to be filtered out from the beginning.

The plugin performs these initial checks:

- Protocol number equals 17 on the Internet Protocol (IP) Layer
- Destination Port equals 443 on the User Datagram Protocol (UDP) Layer
- On the QUIC Protocol Layer, the packet has a Long Header and is of type Initial

Code listing 2.1 QUICParser::quic_initial_checks

```
bool QUICParser::quic_initial_checks(const Packet& pkt)
{
    // Port check and UDP check
    if ( pkt.ip_proto != 17 ||
        pkt.dst_port != 443 ) {
        return false;
    }
    // Initial packet check
    if (!quic_check_initial(pkt.payload[0])){
        return false;
    }
    return true;
}
bool QUICParser::quic_check_initial(uint8_t packet0)
{
    // version 1
    if ((packet0 & 0xF0) == 0xC0) {
        is_version2 = false;
        return true;
    }
    // version 2
    else if ((packet0 & 0xF0) == 0xD0) {
        is_version2 = true;
        return true;
    } else {
        return false;
    }
}
```

2.4 Extract Header Fields

After we are sure that the obtained packet has the necessary type. We can proceed with the data extraction from the Header. As stated in Section 1.1.4 the Header is used as *associated data* input into the AEAD function. Next, *Destination Connection ID* is used as input into the key derivation function, see Section 1.1.4.1. Lastly, *Packet Number* is used as part of the XOR operation in order to derive *Nonce*, the input parameter of the *AES-GCM*, see Figure 1.7.

Code listing 2.2 QUICParser::quic_parse_header

```
bool QUICParser::quic_parse_header(const Packet& pkt)
{
    (...)
    const uint8_t* payload_pointer = pkt.payload;
    (...)
    if (quic_h1->dcid_len != 0) {
        dcid = (payload_pointer + offset);
        offset += quic_h1->dcid_len;
    }
}
```

During the process of extraction fields from the Header, we need to pay attention to the Variable-Length Integer Encoding in Section 2.1. As can be seen in Figure 1.5, the Packet Number Length field inside the Header is marked as *Protected.* To sum up, at this stage, we do not exactly know where the Packet Payload starts because we do not yet know what the length of the Packet Number field within the Header is; thus, we are not even able to extract the Packet Number completely. However, after the Header Protection mechanism is reverted in the next sections, we are going to obtain the Packet Number Length field, which will give us the required information about the Packet Number.

2.5 Initial Secrets Derivation

As may be evident from Section 1.1.4.1, the derivation of initial secrets occurs in several steps.

- The first step involves deriving the *initial_secret*.
- Following the *initial_secret*, the *client_initial_secret* for the client side, respectively *server_initial_secret* for the server side, can be derived.
- In the final step, the final initial secrets, which are used as keying input into the AEAD algorithm, are derived from the previous secrets. These include:
  - The AEAD key and initialization vector.
  - The Header Protection key.

As we can see in Code listing 2.3 in order to derive *initial_secret* we need to use *HKDF-Extract* function which consists of input parameters as:

- Version specific Initial Salt
- Client Destination Connection ID, which is available from the 2.4

Code listing 2.3 pseudo-code of Initial Secret derivation

```
// Initial salt specific for the latest RFC 9001 version
initial_salt = 0x38762cf7f55934b34d179ae6a4c80cadccbb7f0a
initial_secret = HKDF-Extract(initial_salt, client_dst_connection_id)
```

2.5.1 Client Initial Secret

To revert the initial encryption of the *Client Hello* message. We need to obtain encryption keys used by the *Client* to encrypt the *Initial* packet. In Code listing 2.4 we can see pseudo-code how are further *client_initial_secret* or *server_initial_secret* derived from the previously derived common *initial_secret*. To be more precise, the input arguments are as follows:

- Already derived common Initial Secret
- "client_in" or "server_in" string, based on the side.
- Hash.length in this case 32 bytes

Code listing 2.4 pseudo-code of Client Initial Secrets derivation

```
client_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "client␣in", "",
                                          Hash.length)
server_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "server␣in", "",
                                          Hash.length)
```

2.5.2 AES-GCM Initial Secrets

From the derived *client_initial_secret*, we can continue and derive the keying material for the AEAD AES-GCM encryption function. From Section 1.1.4.1 and Figure 1.7 we can conclude that the required keying material is **Initialization Vector** and **Encryption Key**.

The process of derivation is similar to the derivation process of the previous secrets. In Code listing 2.5 we can notice , that *HKDF-Expand-Label* function is used again, with the following arguments

- Previously derived Client Initial Secret
- "quic_key" string for the **AEAD key** and "quic_iv" for the **AEAD Initialization Vector**
- Hash.length of value 16 for the **AEAD key** and 12 for the **AEAD Initialization Vector**

Code listing 2.5 pseudo-code of Key and Initialization Vector derivation

```
key  = HKDF-Expand-Label(client_initial_secret,
                         "quic␣key", "",
                         Hash.length)

iv   = HKDF-Expand-Label(client_initial_secret,
                         "quic␣iv", "",
                         Hash.length)
```

2.5.3 Header Protection secret

As briefly discussed in Section 1.1.4.3 and shown in Figure 1.7, packet Headers also possess minimal security and integrity protection. Parts of the Header, especially the Packet Number and Packet Number Length are protected with the Header Protection algorithm [8]. In order to perform the un-protection algorithm, Header Protection keying material needs to be derived. This keying material is derived in the same way as the Initialization Vector and AEAD Key.

Within the Code listing 2.6, we can see the pseudo-code of the derivation process of the Header Protection key. The inputs are as follows:

- Previously derived Client Initial Secret
- ”quic_hp” string
- Hash.length of value 16

Code listing 2.6 pseudo-code of Header Protection Key derivation

```
hp   = HKDF-Expand-Label(client_initial_secret,
                         "quic␣hp", "",
                         Hash.length)
```

2.6 Reverting Header Protection

The mechanism of the Header Protection is well defined in the official *RFC 9001* [8]. To be able to revert Payload Protection, we need to revert Header Protection first. As already discussed in Section 2.4, Packet Number Length is protected, moreover, Packet Number is a necessary field for Payload protection AES–GCM encryption algorithm as the Packet Number with the conjunction of Initialization vector form Nonce. Unprotected Header is also used as associated_data field for the Authenticated encryption with associated data algorithm.

In the Code listing 2.7, we can see the pseudo-code of the Header Protection application [8]. The mechanism consists of the following steps:

- Encrypt Sample part of the encrypted Packet Payload with the derived Header Protection key.
- Based on the Header Type, within the context of this work, we focus on the Long Header Type as it is used for the *Initial* Packets, mask the first byte of the Header.
- Mask the Packet Number

Code listing 2.7 pseudo-code of Header Protection

```
mask = AES-ECB(hp_key, sample)

pn_length = (packet[0] & 0x03) + 1
if (packet[0] & 0x80) == 0x80:
   # Long header: 4 bits masked
   packet[0] ^= mask[0] & 0x0f
else:
   # Short header: 5 bits masked
   packet[0] ^= mask[0] & 0x1f

# pn_offset is the start of the Packet Number field.
packet[pn_offset:pn_offset+pn_length] ^= mask[1:1+pn_length]
```

It is worth noting that, we discussed within the Section 2.4 that the Packet Number Length is protected so we do not know exactly where the Packet Payload starts as we do not know exactly what is the length of the Packet Number. However, in the official *RFC 9001* [8], allowance was made. This allowance specifies that the Sampled Part will always start at an offset of 4 bytes after the Packet Number starts, this allows the removal of protection by a receiving endpoint.

2.6.1 Sample Encryption

Firstly, to successfully revert the protection of the Header we are going to perform Sample encryption with the Header Protection key derived in Section 2.5.3. This keying material is used as an Input to the *Advanced Encryption Standard* in the *Electronic Codeblock* mode. The Sampled Part is taken as part of the Packet Payload starting on an offset of 4 bytes after the Packet Number. In Code listing 2.8 we can see a simplified process of such encryption.

Code listing 2.8 QUICParser::quic_decrypt_header

```
bool QUICParser::quic_encrypt_sample(uint8_t* plaintext)
{
    // setup context
    EVP_EncryptInit_ex(ctx,
                       EVP_aes_128_ecb(),
                       NULL,
                       initial_secrets.hp,
                       NULL)
    // "plaintext" contains the final output
    (...)
    if (!(EVP_EncryptUpdate(ctx,
                            plaintext,
                            &len,
                            sample,
                            SAMPLE_LENGTH = 16))) {
        return false;
    }
    (...)
}
bool QUICParser::quic_decrypt_header(const Packet& pkt)
{
    (...)
    // Encrypt sample with AES-ECB.
    if (!quic_encrypt_sample(plaintext)) {
        return false;
    }
    memcpy(mask, plaintext, sizeof(mask));
    (...)
}
```

2.6.2 Packet Number Length

After performing Sample Encryption on the Packet Payload Sampled part, we can continue with reverting the protection of Header fields. In this case, we are going to revert protection of the Packet Number Length, this field is crucial as in order to obtain the exact start of the Packet Payload we need to know exactly what is the length of the Packet Number. As already stated, we are going to consider only the Long Header Type so there is no need to perform the check described in Code listing 2.7. In Code listing 2.9, we can see the process of reverting protection of the first byte, the first byte contains Packet Number Length, this fact can be seen in Figure 1.5.

Code listing 2.9 QUICParser::quic_decrypt_header

```
bool QUICParser::quic_decrypt_header(const Packet& pkt)
{
    (...)
    // Long header: 4 bits masked
    first_byte = quic_h1->first_byte ^ (mask[0] & 0x0f);
    pkn_len = (first_byte & 0x03) + 1;
    (...)
}
```

After we successfully revert the protection of the Packet Number Length, we can continue with updating the start of the Packet Payload. To put this in context, as stated in Section 2.4, we were not able to obtain the exact Packet Payload start as Packet Number Length and Packet Number fields are protected with the Header Protection so we had to revert the protection of Packet Number Length first. In Code listing 2.10, we can see that the Packet Payload and the corresponding length of the Payload and Header, respectively, are updated.

Code listing 2.10 QUICParser::quic_decrypt_header

```
bool QUICParser::quic_decrypt_header(const Packet& pkt)
{
    (...)
    // after unprotecting pkn_len, we know exactly pkn length
    // so we can correctly adjust the start of the payload
    payload = payload + pkn_len;
    payload_len = payload_len - pkn_len;
    header_len = payload - pkt.payload;
    (...)
}
```

2.6.3 Packet Number

One of the last steps of reverting the Header Protection is to obtain the Packet Number. In the previous Section 2.6.2, we obtained the Length of the Packet Number. Now we can proceed and obtain the Packet Number, then we will continue and derive the Nonce, which is the input of AES-GCM encryption algorithm in Figure 1.7. In Code listing 2.11, we can see how the Packet Number protection is reverted, and then the Header is updated so that it contains an unprotected Packet Number.

Code listing 2.11 QUICParser::quic_decrypt_header

```
bool QUICParser::quic_decrypt_header(const Packet& pkt)
{
    (...)
    // copy protected packet number into the buffer
    uint8_t full_pkn[4] = {0};
    memcpy(&full_pkn, pkn, pkn_len);
    for (unsigned int i = 0; i < pkn_len; i++) {
        packet_number |=
            (full_pkn[i] ^ mask[1 + I]) <<
            (8 * (pkn_len - 1 - i));
    }
    // update the header so that it contains unprotected
    // packet number
    for (unsigned i = 0; i < pkn_len; i++) {
        header[header_len - 1 - i] =
                (uint8_t)
                (packet_number >> (8 * i));
    }
    (...)
}
```

Lastly, as we currently have the Header completely unprotected, we can proceed with creating the Nonce, which is derived from Packet Number and Initialization Vector, see Figure 1.7. In Code listing 2.12, it can be seen how the Nonce is created based on the Initialization Vector derived in Section 2.6.3, and Packet Number.

Code listing 2.12 QUICParser::quic_decrypt_header

```
bool QUICParser::quic_decrypt_header(const Packet& pkt)
{
    (...)
    // adjust nonce for payload decryption
    // The exclusive OR of the padded packet
    // number and the IV forms the AEAD nonce
    phton64(initial_secrets.iv + sizeof(initial_secrets.iv) - 8,
            pntoh64(
                initial_secrets.iv +
                sizeof(initial_secrets.iv) - 8) ^
                packet_number);
    (...)
}
```

2.7 Reverting Packet Protection

For the Payload Protection *AEAD_AES_128_GCM* mechanism is used [8]. From the previous Sections we successfully obtained the:

- AES-128-GCM Encryption Key in Section 2.5.2
- Associated Data which in this case are all Header fields in unprotected form, reverting Header Protection is described in Section 2.6
- Packet Payload starting point in Section 2.6.2
- Nonce creation which was created after unprotecting the Packet Number in Section 2.6.3

Setup Encryption Context and Authentication Tag

In the first step of Packet Payload encryption, we need to set up the encryption context and the Authentication Tag. As seen in Figure 1.7 unprotected Header is used as an associated data input to the AES-GCM algorithm, this means that the algorithm will produce an Authentication Tag at the end of the encryption which is then used to check the integrity of the associated data, but the associated data field is not encrypted in this stage. In Code listing 2.13 we can see this initial setup process.

Code listing 2.13 QUICParser::quic_decrypt_payload

```
bool QUICParser::quic_decrypt_payload(const Packet& pkt)
{
    (...)
    // setup Authentication Tag as last part of the Payload
    uint8_t atag[16] = {0};
    payload_len -= 16;
    memcpy(&atag, &payload[payload_len], 16);
    (...)
    // setup encryption algorithm
    EVP_DecryptInit_ex( ctx,
                        EVP_aes_128_gcm(),
                        ...)

    // setup lengths of IV and Nonce
    EVP_CIPHER_CTX_ctrl(ctx,
                        EVP_CTRL_AEAD_SET_IVLEN = 9,
                        TLS13_AEAD_NONCE_LENGTH - 12,
                        NULL)

    // setup nonce and key
    EVP_DecryptInit_ex(ctx, NULL, NULL,
                       initial_secrets.key,
                       initial_secrets.iv)
    (...)
    // SET ASSOCIATED DATA (HEADER with unprotected PKN)
    EVP_DecryptUpdate(  ctx,
                        NULL,
                        &len,
                        header,
                        header_len)
    (...)
}
```

Key and Nonce

The next step is to set Nonce and Key. The Initialization vector is derived within the Section 2.5.2, and the Nonce is created in Section 2.6.3, respectively. The corresponding AES-GCM encryption key was also derived in Section 2.5.2. Within the Code listing 2.14 we setup the Encryption Key and the Nonce.

Code listing 2.14 QUICParser::quic_decrypt_payload

```
bool QUICParser::quic_decrypt_payload(const Packet& pkt)
{
    (...)
    // Setup Encryption Key and Nonce
    EVP_DecryptInit_ex( ctx,
                        NULL,
                        NULL,
                        initial_secrets.key,
                        initial_secrets.iv
    (...)
}
```

Finalize Encryption and Check Authentication Tag

Finally, the main and also last step of Packet Payload encryption is to perform and finalize the encryption and check the Authentication Tag. In the Code listing 2.15 we can see how the encryption process, the finalization and the Authentication Tag are performed.

Code listing 2.15 QUICParser::quic_decrypt_payload

```
bool QUICParser::quic_decrypt_payload(const Packet& pkt)
{
    (...)
    // Perform the Encyrption
    EVP_DecryptUpdate(  ctx,
                        decrypted_payload,
                        &len,
                        payload,
                        payload_len)
    // Perform the Authentication Tag check
    EVP_CIPHER_CTX_ctrl(ctx,
                        EVP_CTRL_AEAD_SET_TAG,
                        16,
                        atag)

    // Finalize Encrytpion
    EVP_DecryptFinal_ex(ctx,
                        decrypted_payload + len,
                        &len)
    (...)
}
```

2.8 Assemble Packet Payload

Due to the *Chaos Protection* feature described in the Section 1.1.5.2, we have to make sure that we consider also *CRYPTO* frames that are split across the whole Packet Payload of the *Initial Packets*. As mentioned in the *RFC 9000*[5]. There are only a few Frame types that can occur in the Packet Payload of *Initial* packets. More precisely,

Frame types that can be included in the *Initial* packets:

- **PADDING** with the type value of *0x00*
- **PING** with the type value of *0x01*

- **ACK** with the type value of *0x02-0x03*
- **CRYPTO** with the type value of *0x06*
- **CONNECTION_CLOSE** with the type value of *0x1c-0x1d*

Once we know which Frame types we can expect in the *Initial* Packets, we can simply iterate through the Packet Payload.

Code listing 2.16 QUICParser::quic_reassemble_frames

```
bool QUICParser::quic_reassemble_frames(const Packet& pkt)
{
    (...)
    uint8_t* payload_end = payload + payload_len;
    while (current < payload_end) {
        if (current == CRYPTO){
            // update current position inside
            quic_copy_crypto(payload);
        } else if (current == ACK){
            // update current position inside
            quic_skip_ack(payload);
        } else if (current == CONNECTION_CLOSE1){
            // update current position inside
            quic_skip_connection_close(payload);
        } else if (current == PADDING){
            current++;
        } else if (current == PING){
            current++;
        } else{
            return false;
        }
    (...)
}
```

In Code listing 2.16 listing above, we can see the simplified logic of such iteration. During the iteration, once we find *CRYPTO* Frame, we can proceed with copying *Handshake* data from the CRYPTO Frame. As described in Section 1.1.5.1 CRYPTO Frame contains the field Length and Offset. These two fields tell us how much *Handshake* data we should copy and what is the offset on which this *Handshake* data should be located. In Code listing 2.17 we can see how the described logic is performed, additionally worth noting that, the Frame Offset and Length fields use Variable Length Encoding mentioned in Section 2.1.

Code listing 2.17 QUICParser::quic_copy_crypto

```
bool QUICParser::quic_copy_crypto(uint8_t* start, uint64_t& offset)
{
    (...)
    // located length and offset fields
    uint16_t frame_offset = quic_get_variable_length(start, offset);
    uint16_t frame_length = quic_get_variable_length(start, offset);
    (...)
    // copy Handshake data to the assembled payload
    memcpy(assembled_payload + frame_offset, start + offset, frame_length);
}
```

2.9 Obtain TLS Data

In Section 1.3.1.1, we described the IPFIXProbe and basic flow data that the Plugin exports. Within this Section, we will introduce new flow fields corresponding to the QUIC Protocol and the underlying TLS.

After we successfully reverted Header Protection and Packet Payload Protection in the previous Sections, it is now possible to parse data from the re-assembled CRYPTO Frames contained within the *Initial* Packet. In order to obtain TLS data, IPFIXProbe TLS Plugin was partially used; however, this parser was further expanded to be able also to parse fields such as User Agent (described in Section 1.2.2).

In the Code listing 2.18, we can see the starting process of obtaining TLS data. Firstly, based on the TLS Parse plugin already contained in IPFIXProbe, we try to check if the TLS message contains the required fields. If any of these checks fail, the process is terminated. When all of the checks pass, we proceed with obtaining the TLS data.

Code listing 2.18 QUICParser::quic_parse_tls

```
bool QUICParser::quic_obtain_tls_data()
{
    (...)
    tls_parser.tls_get_server_name(payload, sni, BUFF_SIZE);
    tls_parser.tls_get_quic_user_agent(payload, user_agent, BUFF_SIZE);
    (...)
}
bool QUICParser::quic_parse_tls()
{
    (...)
    if (!tls_parser.tls_check_handshake(payload)) {
        return false;
    }
    if (!tls_parser.tls_skip_random(payload)) {
        return false;
    }
    if (!tls_parser.tls_skip_sessid(payload)) {
        return false;
    }
    (...)
    if (!quic_obtain_tls_data(payload)) {
        return false;
    }
    (...)
}
```

In the Code listing 2.19, we can see what the simplified version of obtaining a User Agent can look like. However, this function is only the last point and is pretty simple as the structure of this extension stored in the QUIC *Initial* Packet is publicly known and can be freely searchable.

Code listing 2.19 QUICParser::get_quic_user_agent

```
void TLSParser::get_quic_user_agent(
        TLSData &data,
        char *buffer)
{
   const uint8_t *quic_transport_params_end =
        data.start +
        quic_transport_params_len +
        sizeof(quic_transport_params_len);
   uint64_t offset = 0;
   uint64_t param  = 0;
   uint64_t length = 0;

   while (data.start + offset < quic_transport_params_end) {
        // variable length fields
        param  =
            quic_get_variable_length((uint8_t *) data.start, offset);
        length =
            quic_get_variable_length((uint8_t *) data.start, offset);
      if (param == TLS_EXT_GOOGLE_USER_AGENT) {
        memcpy(buffer, data.start + offset, length);
        return;
      }
      offset += length;
   }
   return;
}
```

2.10 Finalized Plugin

The output (or the flow) of the IPFIXProbe protocol was enriched by the data contained within the decrypted *Initial* Packets. Table 2.2 shows fields that are extracted from the *Initial* Packets of the QUIC Protocol, in Section 1.3.1.1 we can see Basic fields that IPFIXProbe extracts. The QUIC Plugin output can be optionally included.

Table 2.2 New IPFIXProbe fields introduced by the QUIC Plugin

Output field	**Type**	**Description**
QUIC_SNI	string	Server name Indication
QUIC_USER_AGENT	string	User Agent field
QUIC_VERSION	uint32	Version of used QUIC Protocol

2.10.1 Dataset

Based on the plugin created, and with the help of my supervisor, we created Dataset, which is later on used within the Analysis Chapter 3 and 4. This Dataset contains backbone flows captured with the IPFIXProbe and enriched by QUIC Plugin flow field data. The Dataset contains captured and analyzed data, which span from Week 40 of the year 2022 until Week 20 of the year 2023. There is a small gap in the Dataset, which is in the week 50 of the year 2022. Unfortunately, the IPFIXProbe was not operational during some of the days that week, thus the week is not included in the Dataset. The part of the overall dataset (Week 44 til Week 47) was made publicly available in the journal publication *CESNET-QUIC22: A large one-month QUIC network traffic dataset from backbone lines* [21]

Chapter 3

Analysis

In Chapter 2 we created the IPFIXProbe QUIC plugin capable of extracting encrypted data from *Initial* Packets of the QUIC Protocol. Later on, after the plugin was finalized and deployed, with the help of my thesis supervisor, we created QUIC Dataset, which is described in Section 2.10.1. This Dataset is analyzed in this Chapter.

3.1 State of the Dataset

Firstly we analyzed how data contained within the *Initial* Packets evolve each set of weeks. How many flows contain Cronet-type User Agent, how many flows contain Chrome-type User Agent, how many flows are connected to the particular device types, and more. In Table 3.1 we can see these numbers. Each column represents data from the week span contained within the column label.

Table 3.1 Initial Flow Statistics

	40-43	44-47	48-52	01-04	05-08	09-12	13-16	17-20
Number of flows	307 863	335 916	243 116	216 760	248 364	308 630	242 313	221 306
Chrome User Agents	84.9%	86.7%	84.8%	86%	83.2%	86.1%	83.2%	85.6%
Cronet User Agents	13.7%	11.2%	11.6%	11.1%	13.3%	11.4%	13.6%	11.5%
Android	63 819	71 594	47 269	36 337	47 199	59 386	52 353	55 666
Mac	58 422	77 171	42 535	34 995	35 921	54 271	38 041	30 100
Windows	111 653	107 999	84 398	92 845	117 282	141 081	97 531	85 469
Linux	29 528	39 298	38 756	26 304	14 093	17 468	11 252	13 360

The Table shows the initial statistics of the flows we captured. Firstly we differentiate between the Chrome user Agents and Cronet User Agents. This is because Chrome User Agents can provide more information about the device than the Cronet User Agents, example of both types can be seen in Figure 3.1, so we want to know what is the fraction of these types.

The Table 3.1 also contains numbers representing occurrences of each device type. We can see that the numbers are decreasing over time; we suspect that this is due to the newer version of Chrome, which deprecated the User Agent fields, and the field is no longer present in this version.

```
Cronet related User Agent:
    com.google.android.gms Cronet/99.0.4844.35
Chrome related User Agent:
    Chrome/92.0.4515.107 Windows NT 10.0; Win64; x64
```

Figure 3.1 Comparison of Cronet and Chrome related User Agents

3.2 User Agents

After we collected basic statistics about the Dataset, we continued evaluating the User Agents. We were interested in observing how many User Agents are still present in the *TLS Handshake*, and what these User Agents look like. Thus, this Section contains data about the ten most common User Agents depending on the ecosystem, such as Androids, Macs, iPads and iPhones, Windows, and lastly, Linux.

Android

In Figure 3.2, we can see a pie chart representation of the 10 most common Android User Agents across our entire dataset. This chart clearly shows which Android User Agents are used most frequently. *Chrome/91.0.4472.134 Android 10; JNY-LX1* is the most common, indicating a high prevalence in our data. Based on the Huawei official webpage[1], this model number belongs to the *HUAWEI P40 lite* which we think is expected as the Huawei brand is common in the general public. This is followed by *Chrome/92.0.4515.115 Android 10; Redmi Note 7* and *Chrome/94.0.4606.85 Android 11; Mi A3*, showing a lower but significant usage.

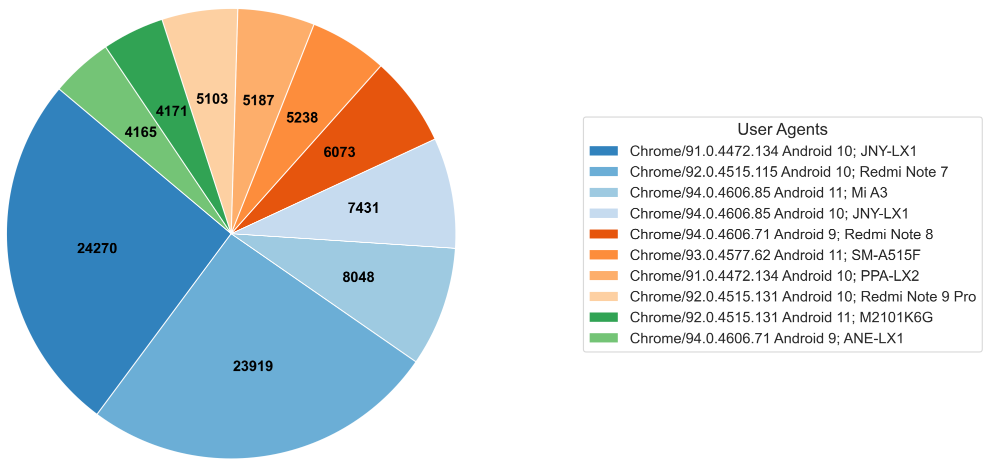

Figure 3.2 most common Android related User Agents

Mac

Next, in Figure 3.3, we can see the distribution of the 10 most common Mac User Agents within our dataset. The chart effectively highlights *Chrome/94.0.4606.71 Intel Mac OS X 10_15_7* as

[1] `https://consumer.huawei.com/cz/support/phones/p40-lite/`

the predominant user agent, clearly showing its widespread use among Mac users. Following closely are *Chrome/91.0.4472.164 Intel Mac OS X 10_15_7* and *Chrome/91.0.4472.114 Intel Mac OS X 10_15_7*, which also represent significant portions. We can clearly see that the device part within the User-Agent stays the same for almost the whole ordering; the only thing that changes is the Chrome version.

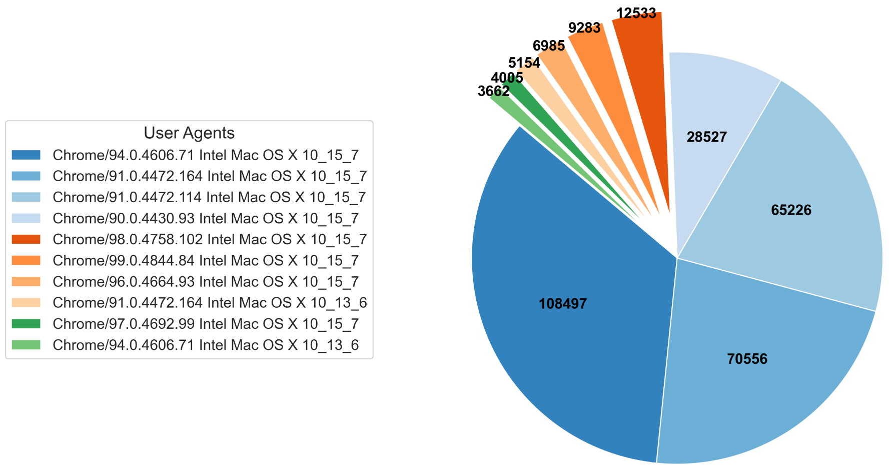

Figure 3.3 most common Mac related User Agents

iPhone and iPad

Continuing with iPhone and iPad devices, in Figure 3.3, we noticed that the variance of Mac-based devices is not high. Moreover, we suspect that the *Intel Mac OS X 10_15_7* User Agent corresponds to the MacBooks. Thus, we tried to evaluate if we would be able to find also portable devices from the Apple Ecosystem. In Figure 3.4, we can see that the Dataset also contains iPads and iPhones. We can also notice that the absolute number of occurrences in our Dataset is relatively low compared to other devices. Even though, I find it interesting, as I thought more people owned iPhones or iPads. However, the lower number of such devices could probably result from two factors: first, not as many people as I initially thought actually own iPhones; second, within this ecosystem, especially for mobile devices, the Safari browser is more popular.

Windows

The situation with Windows devices is pretty much the same as for Macintosh devices. Based on Figure 3.5 we can again conclude that the device part does not provide as much variance as the Chrome version. We can see that the most occurred device across all Dataset connected to Windows is *Windows 10* with the *64-bit architecture*. We think that this User Agent mostly refers to personal desktop computers and servers.

Linux

The last category in the User Agents that we try to observe is Linux-related User Agents. In Figure 3.6 we can see the ten most common Linux User Agents across the Dataset. From first sight, we can observe that the situation is almost the same as for Macintosh and Windows, like the User Agent *Chrome/91.0.4472.77 Linux x86_64* contains always the same device model, but

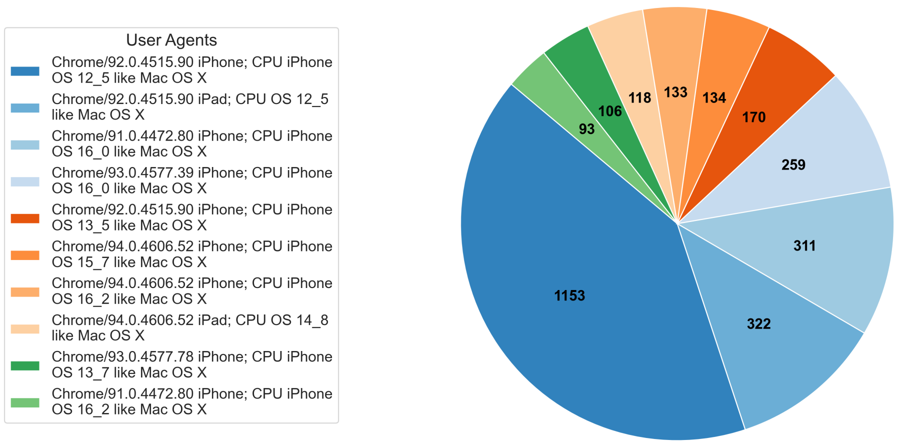

Figure 3.4 most common iPhone and iPad related User Agents

the Chrome version provides variance. On the other hand, it is interesting that for User Agents *built on Debian 10.9. running on Debian 10.13 Chrome/90.0.4430.212 Linux x86_64* and *built on Ubuntu, running on Ubuntu 16.04 Chrome/90.0.4577.82 Linux x86_64* we can see different Linux distributions i.e. Debian and Ubuntu.

3.3 Server Name Indications

Regarding the Server Name Indication field, in Figure 3.7 we can see the ten most accessed Server Names in our Dataset. As expected, we can see that amongst the top common Server Names are services corresponding to Spotify, Google, and Avast. We do not find these numbers particularly interesting right now, however in Chapter 3 4 we will discuss that the Server Name Indication can be helpful in aggregating flows based on unique identifiers.

3.4 QUIC Version

To complete the initial analysis we collected the most common QUIC Version numbers. In Table 3.2 we can observe that the most common specified versions are *0xff00001D* which corresponds to the *draft-ietf-quic-transport-29* versions. As stated in the *draft-ietf-quic-transport-29* [22], version numbers used to identify IETF drafts are created by adding the draft number to 0xff000000. For example, *draft-ietf-quic-transport-13* is identified as 0xff00001D. The second most common version is version *0x00000001* which identifies that the latest QUIC specification is used [5].

Table 3.2 Quic Version occurrences

QUIC Version Field	Number of occurrences
0xff00001D	1268946
0x00000001	855320

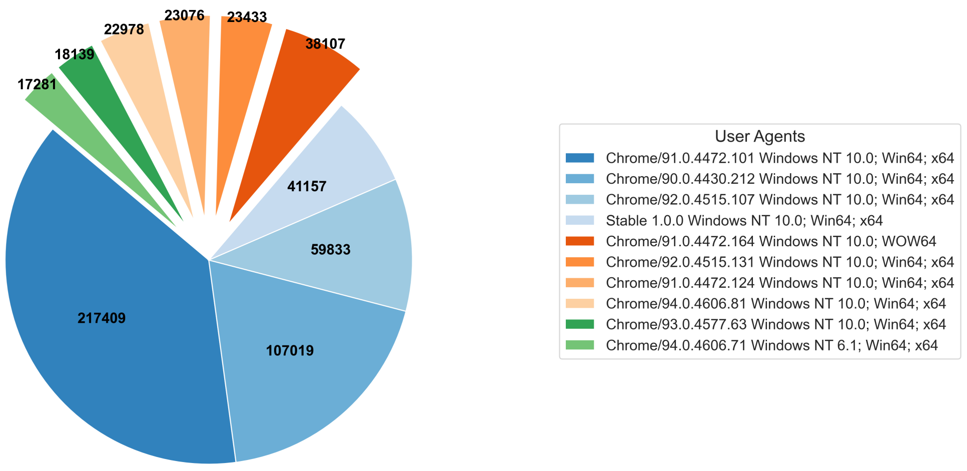

Figure 3.5 most common Windows related User Agents

3.5 Conclusion

In this Chapter, we obtained basic statistics from our Dataset. We were observing differences between Cronet and Chrome, like User Agents. We also collected numbers corresponding to the most common User Agents or devices, based on the Ecosystem they belong to, namely Androids, iPads, iPhones, Macs, Windows, and Linux. We noticed that the model identifiers in the User Agent field are interesting, publicly observable identifiers. In connection with the Server Name Indication, we think that we may be able to aggregate flows in Dataset which can help us to understand more about the User's behavior, activity, or location. This topic is described in more detail within the next Chapter 4.

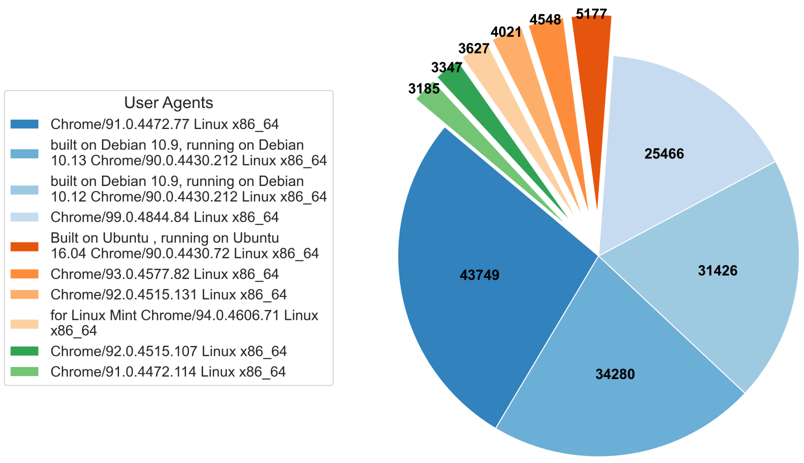

Figure 3.6 most common Linux related User Agents

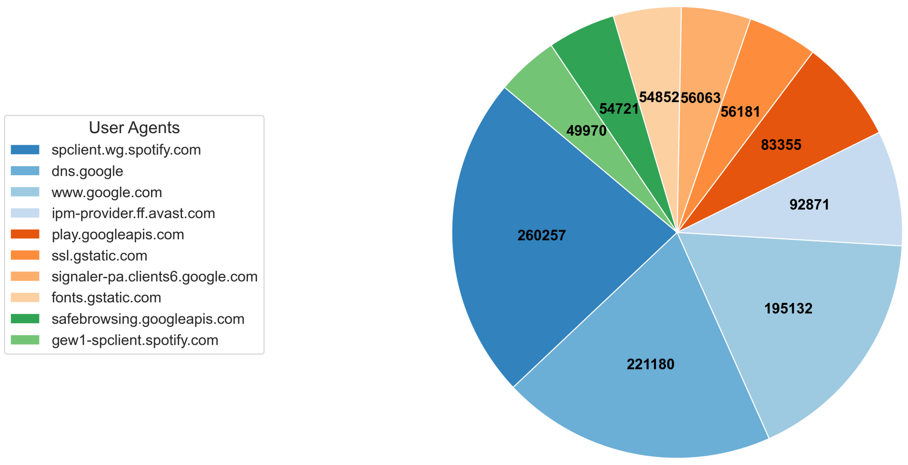

Figure 3.7 most common Server Name Indications

Chapter 4

User Agent Model Numbers

In the previous Chapter 3 we noticed that apart from the Cronet User agents, the remaining User Agents contain information about the device, like model number and occasionally chrome version. Since some model numbers were non-standard and "exotic," we formulated a hypothesis that a persistent Chrome user agent can be used as a persistent device identifier regardless of IP address changes. If true, the device movement can be traced between multiple locations determined by the device IP addresses. In this chapter, we describe all analysis methods and results used for confirmation or rejection of our hypothesis.

4.1 Dataset Preparation

From the raw dataset flow data, we automatically extracted all information necessary for our analysis. All other fields have not been used in this analysis. The used fields are:

Device Location or, more precisely, address of IP address owner (e.g., institution or ISP). We have created a script that was automatically executed by the thesis supervisor to obtain owners of the IP addresses. For that purpose, we use public service *IPinfo*[1], which provided us with the Name of the address owner and its address. The address of the owner served as a location in our analysis.

User Agent was obtained directly from the dataset. It is considered a persistent identifier.

Server Name Indication obtained directly from the dataset. A combination of User Agent and SNI can make the persistent identifier more accurate.

Timestamp used for monitoring changes of Device location in time.

The real user IP addresses have been processed automatically and never inspected by me or the thesis supervisor. By analyzing these data, we did not attempt to link the User Agent with the real user on the network. Moreover, we believe that such linkage is impossible based on the data available in the dataset. The aim of this analysis is to inspect the possibility of possible privacy violation techniques that could be leveraged by high-power entities like oppressive regimes.

4.2 Aggregating flows

The first idea is to aggregate the flows. The Dataset created is basically nothing less than a huge amount of flows captured. We think that it is important to aggregate flows into the smaller

[1] `https://ipinfo.io/`

clusters, and then we may be able to analyze these clusters better. Ideally, it would be best to find such unique keys so that when we aggregate the flows together, each cluster will correspond to one particular device.

4.2.1 User Agent and Location

Firstly we try to aggregate the flows by the User Agent and Location. This idea from first sight may look promising, on the other hand, this aggregation will not help us much because users can end up in different clusters, even though the User Agent is the same, but the location differs.

4.2.2 User Agent as Unique Identifier

The second approach we tried was to aggregate flows based only on User Agents. This approach is not sufficiently unique, for example for stationary devices, there are a lot of flows, with the same User Agent but with different locations. For more portable devices, this approach may be good for older devices, which will result in a smaller cluster of flows, however for newer devices we would obtain a large cluster of data but spread across geolocation. Within the next part, we add an additional flow field, which can increase uniqueness, thus improving aggregation.

4.2.3 User Agent and SNI as Unique Identifier

Finally, the last approach to the aggregation of flows is to aggregate them Using the Server Name Indication along with the User Agent. Within this approach, we follow the hypothesis that users are connecting from the same device, and connect to the same Web Pages repeatedly. For example, I‘m attending the Czech Technical University Faculty of Information Technology in Prague, but I‘m from Slovakia and I go home every third week. We believe that this scenario can result in situations where most of my QUIC network traffic can be captured in Prague. However, during some weekends, my network traffic can also be captured within my hometown. Therefore, based on the approach of aggregating flows based on User Agent and Server Name Indication, we should see such behavior in students traveling across the Czech and Slovak Republic.

4.3 Evaluation Function

At this point we had IPFIX flows aggregated based on the User-Agent and Server Name Indication as described in the previous Section, We started to think that it would be helpful to try to evaluate the aggregated data based on some evaluation function. The function does not need to evaluate every aggregated Server Name Indication and User Agent with a precise number, but it should provide some helpful information. Based on this, we created *Evaluation function*, which can extract the following information from the aggregated data.

- **Number of Empty Days/Weeks**, number of days or weeks for which the User Agent and Server Name Indication aggregated flows have no flow record.
- **Number of unique locations** across all aggregated flow records for the given User Agent and Server Name Indication pair.
- **Number of Unique Cities**
- **Approximation of kilometers traveled**. This number can be calculated based on the location.
- **Aggregated Days**. Flows for the corresponding User Agent and the Server Name Indication are again aggregated based on the part of the week they occurred

- Monday, Tuesday: start of a week
- Wednesday, Thursday: mid of a week
- Friday: a separate part
- Saturday and Sunday: marked as weekend

- **Aggregated Hours**, Approach similar to the point above, however, for hours in the day.
  - Hour 0-3: marked as night
  - Hour 4-7: marked as early morning
  - Hour 8-11: marked as morning
  - Hour 12-13: marked as lunch
  - Hour 14-16: marked as after lunch
  - Hour 17-21: marked as evening
  - Hour 17-21: marked as night
- **Dominant Weekpart**, based on the above aggregated weekdays, we calculated which part of the week is dominant.
- **Dominant Daypart**, based on the above aggregated day hours, we calculated which part of the day is dominant.
- **The Smallest Distance**, which is computed as the smallest distance between two consecutive (meant consecutive in connection to time) locations that are not the same (so that it cannot be 0).
- **The Biggest Distance**, which is the biggest distance between two consecutive locations.
- **Maybe Server**, a field that is simply computed based on the occurrence of the string "DNS" or "Windows NT" in the User Agent.

4.4 Visualization

After we created the *Evaluation Function*, we could easily filter out those devices that contain, for example, a lot of unique Locations or limit the number of Locations to a strict number. This can be particularly helpful in situations when we want to visualize what the dataset looks like or how the network traffic captured for the particular User Agent looks like. This was exactly our next step.

4.4.1 Folium Visualization

The first and most simple approach was to start visualizing locations, clusters, and very limited patterns by using Folium[2]. Folium is a Python-based library that uses Leaflet[3] under the hood. This visualization may be useful as an initial step before trying to analyze data, as it provides simple maps with the dots where the user was or as a final product after the full analysis is performed just to showcase results. We do not believe that this visualization technique was particularly helpful during the analysis process. Therefore, we mention it only as part of the process, but we will not draw any conclusions based on this approach.

[2] `https://github.com/python-visualization/folium`
[3] `https://leafletjs.com/`

4.4.2 Heatmaps

We continued by visualizing using the HeatMap. In Figure 4.1, we can see an example of such HeatMap. On the Y-axis, there is a weekday with the corresponding hour, on the X-axis we can see the city or the location where the corresponding User Agent was. Each of the cells in the HeatMap corresponds to the number of flows at this Location during that time. Another piece of information that this graph provides is the color of the cell, which changes with the number of flows. What we think is also important is that during each period, the corresponding User Agent was in one location or the other but never both at the same time. On one hand side, this cannot ensure that the User Agent corresponds to only one device moving between Kosice and Presov. On the other hand, if the User Agent is at the same time at two different locations, it means that this User Agent probably does not correspond to one user but rather to multiple users. What is also interesting is that even though the User Agent may not be at different locations at the same time, the distance between Locations can also provide additional information or invalidation. If we consider that the Locations are tens or hundreds of kilometers apart and the User Agent was at one location at one time period and the next hour at the second location, it can also be an invalidation that this User Agent corresponds to only one device.

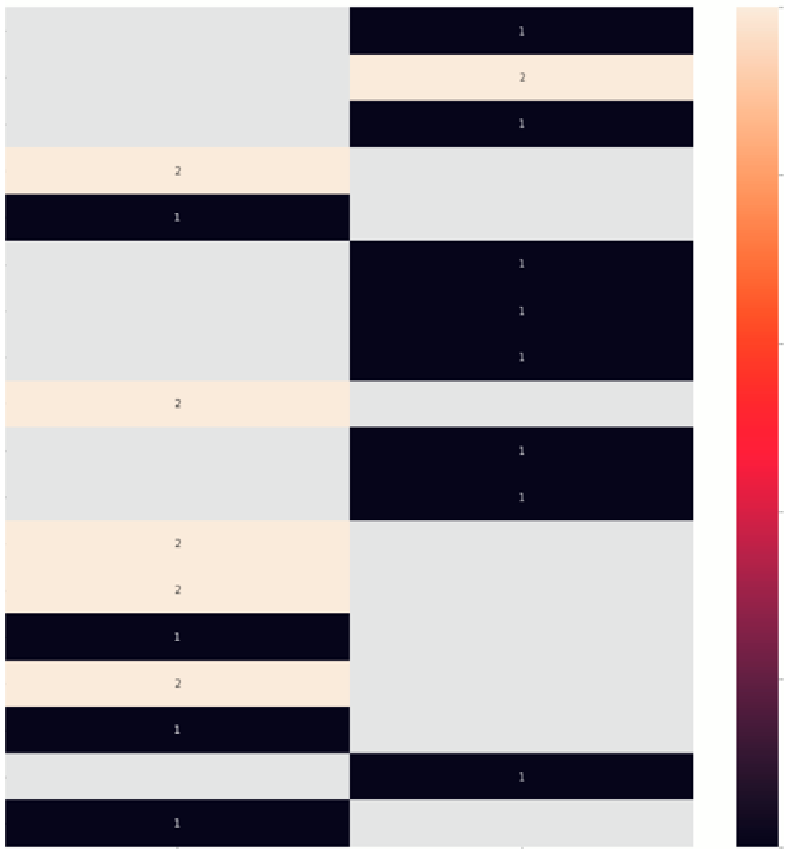

Figure 4.1 Heatmap Visualization, with X-axis as Location and Y-axis as Day-Hour pair

The second great visualization that the HeatMap provides is based on the color. In other words, it can provide the activity of the User Agent. If we look at Figure 4.2 we can see that even though the User Agent is all of the time at only one Location, we can however see the activity of the device connecting to the corresponding Server Name Indication.

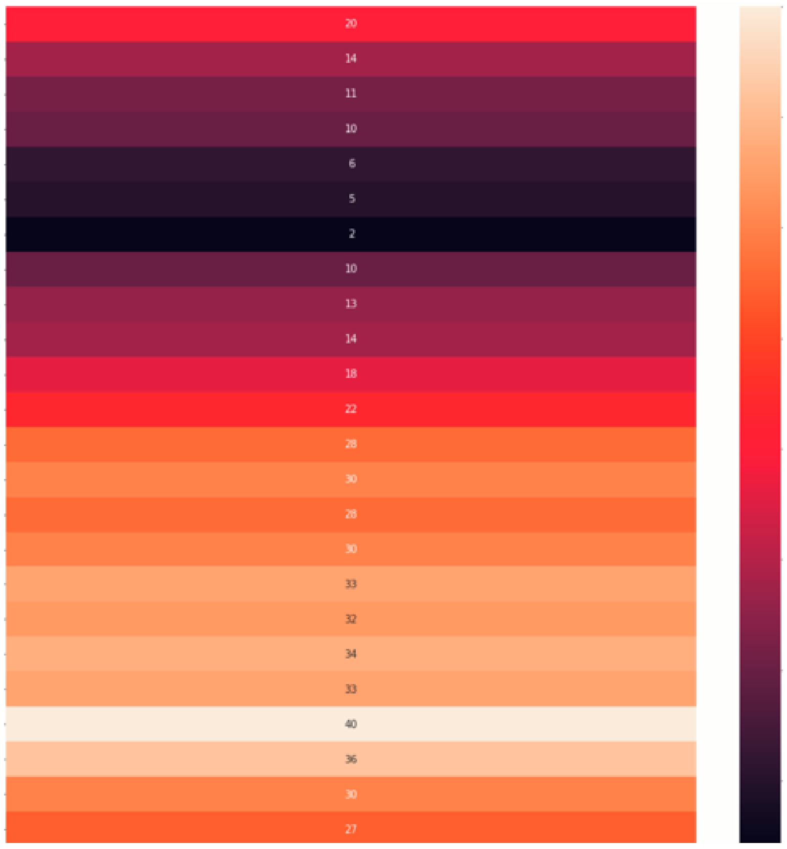

Figure 4.2 Heatmap user activity Visualization, with X-axis as Location and Y-axis as Day-Hour pair

4.4.3 Grid Visualization

The previous visualization technique could help us to find which User Agents can be good potential adepts to spot some activity. However, the technique does not tell anything about the patterns. Our next visualization approach aims to look for patterns. We first start by visualizing using a Grid; then we will continue by trying to obtain additional information from the Grid, leading to potential pattern matching and extraction.

4.4.3.1 Grid Visualization by day

In Figure 4.3, we can see what the Grid looks like. The idea is that the Y-axis corresponds to a day of the week. This means that the grid contains 7 rows. The X-axis is composed of weeks. Each column corresponds to a week. Next, for each cell, the orange color is the default value and it signals that there is no record in our dataset. The dark color corresponds to location, it is not important what is the precise location, the important here are potential patterns. Lastly, the white-colored dot means that there is a collision. This collision can again serve as invalidation; however, as we can see, the User Agent stays in the same place most of the time, so we should probably not take it as invalidation, but rather as noise. What we can notice is that the precision is not great as we have one dot for basically the whole day, in the next Section we are going to split the days for more detailed information.

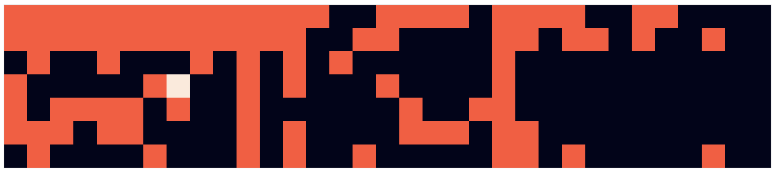

Figure 4.3 Grid visualization with X-axis as week number and Y-axis as week-day. *Orange* = no record, *Black* = record in dataset, *White* = location collision

4.4.3.2 Grid Visualization by day-hour

Based on the previous Grid visualization shown in Figure 4.3, we were wondering if we could increase the data the Grid provides. If we think about it, each cell corresponds to one location in the time. Even though there is often no collision, we can extend this in a way that the day will also be split into smaller pieces. In Figure 4.4 we can see expanded days. The idea is that the Y-axis corresponds again to day but this time with an hour group. Days are split into 4 intervals of 6 hours in length. This means that the grid contains 28 rows. The X-axis is composed again of weeks. Even though the colors changed the idea is still the same as for Figure 4.3.

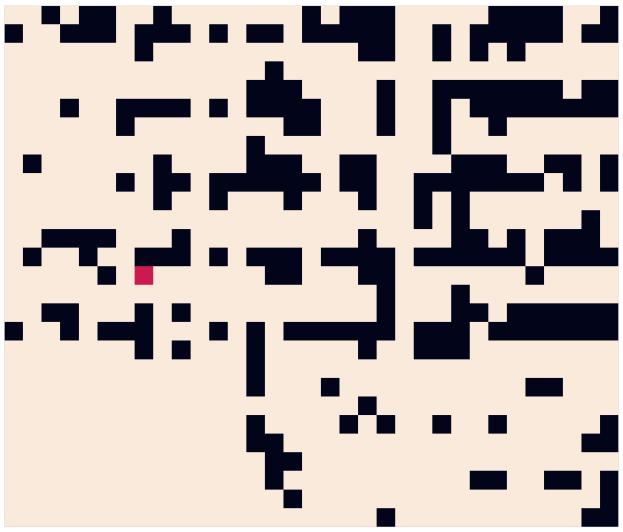

Figure 4.4 Grid visualization with X-axis as week number and Y-axis as weekday with additional hour range. *White* = no record, *Black* = record in dataset, *Red* = location collision

4.4.3.3 Filling Grid Spaces

During the grid visualization, we noticed that we may be able to increase the data contained in the grid synthetically. This means if a User Agent had at a specific location at some time the corresponding cell in the grid is colored by a different color than the default color. Then if the User Agent had the same location at the following cell, but not immediately the next cell but the cell after, in a vertical meaning, we may try to fill the middle space with the surrounding color. We need to be careful because by this, we can introduce noise to the data as we modify the Grid with synthetic inputs, and we may introduce unsubstantiated data. In Figure 4.5 we can see this adjustment in practice compared to Figure 4.4.

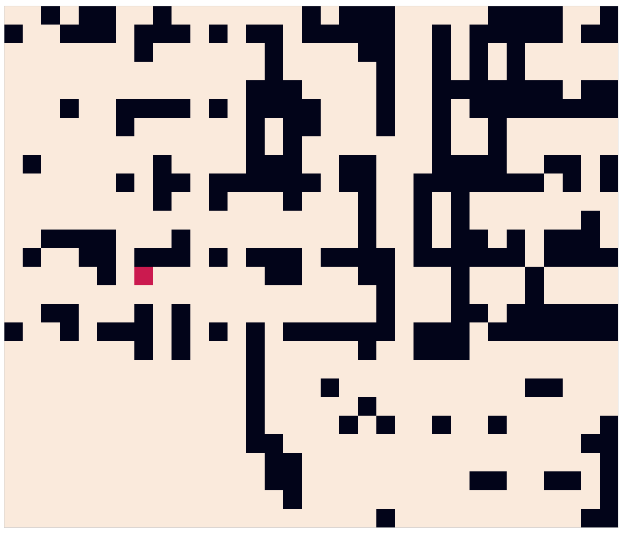

Figure 4.5 Grid visualization with X-axis as week number and Y-axis as weekday with additional hour range and filled spaces. *White* = no record, *Black* = record in dataset, *Red* = location collision

This visualization provides a quality insight into pattern finding. On the other hand, if there are a lot of different Server Name Indications with a lot of different locations it may not work. An example of such a situation can be seen in Figure 4.6, this Grid Map corresponds to the Windows NT User Agent. We can see that to spot patterns for User Agent and Server Name Indication pair movement are hardly possible.

4.5 Finding Patterns Programmatically

Based on the visualization in Figure 4.4 or Figure 4.5, we started to think if it would be possible to find patterns in such a grid programmatically. The patterns, if there are any, do not necessarily need to be spotted by the eye. Thus, we tried to find patterns programmatically in multiple ways. Our approach was primarily based on the correlations of shifts with the original layout and on finding smaller windows in each row with the highest correlations.

4.5.1 Row-wise Autocorrelation with Shift

The first idea is to compute row-wise autocorrelation using shifts. This approach computes which shift has the highest correlation with the original row. As we previously described, each row corresponds to a weekday paired with the hour. Then each column corresponds to a week number. We think that if we shift the whole row by some number it may result in high correlation as we expect that the behavior of users is repetitive. More precisely, if a device was connecting

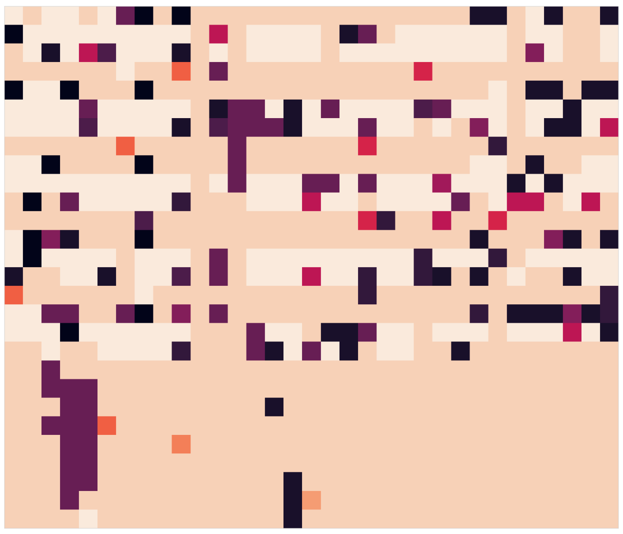

Figure 4.6 Grid visualization example for a lot of records with the correspondence to one device, in this case Windows NT

from some location on Monday morning, then this location may be repetitive, meaning that the device is at the same location each Monday morning or every second Monday morning. This shift should prove it, or at least point it out with a higher correlation number.

In the context of Figure 4.4 the performed shift correlation resulted in the following numbers:

- We will not list all of the rows as it would create an extensive listing.
- The first and the second row from the top have a correlation of 0.24 and 0.43 respectively, with the shift of length 9.
- The fifth row from the top has a correlation of 0.49 with the shift of length 1.
- The eighth row from the top has a correlation of 0.47 with the shift of length 10.
- The sixteenth and seventeenth row from the top correlates 0.47 and 0.37 respectively, with the shift of length 4.

We also performed the same shift correlation on Figure 4.5, which, however, did not provide any significant difference compared to the correlation from Figure 4.4.

4.5.2 Window matching with correlation

Our next approach was to correlate windows of all sizes. This approach is also row-wise, however instead of shifting the whole row and computing correlation with the original row. We started by creating the smallest possible windows to the highest possible ones i.e. row length. These windows are created in such a manner that, for example, for window length 3, we went through the whole row and found all possible windows of length 3. Then we tried to move through the whole row again with this window and compute how much the window correlates with the position, the part where the window was initially found was skipped as this would artificially increase the correlation. This iteration created multiple correlation numbers, from which we then computed the average correlation and median correlation for a particular window. Lastly, based on the average correlation or median correlation we could choose which window matches the best for the whole grind or for only the row.

In the following Figure 4.7 we can see the most correlated windows based on the median correlation 0.13 from the first row from the top for the grid in Figure 4.4 and we got the same result also for the grid in Figure 4.5.

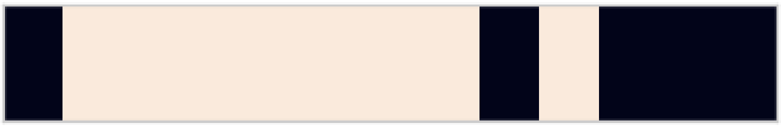

Figure 4.7 The most correlated window for the first row based on the median

Continuing with the second example in Figure 4.8 we got the best window for the fifth row with an average correlation of 0.20 and a median correlation of 0.26. This window is also common for both Figure 4.4 and Figure 4.5.

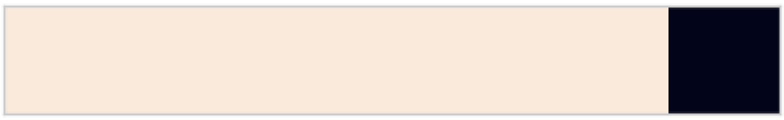

Figure 4.8 The most correlated window for the fifth row based on the median

On the other hand, apart from the previous example, in Figure 4.9 we can see a window that is not common for Figure 4.4 and Figure 4.5, more precisely, this window was only found in the Figure 4.5 which contains filled spaces between corresponding locations in respect to the Y-axis, on the other hand the Figure 4.4 does not contain any window in this row that has median correlation higher than 0.

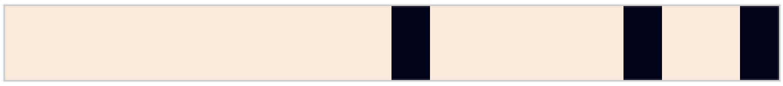

Figure 4.9 The most correlated window for the first row in the grid visualization with filled spaces based on the median

4.6 Discussion

Our final thought for the uniqueness of the User Agent and Server Name Indication pair is that for the older devices that use older versions of Android (in this case, we can talk about Android 6). It may be possible to investigate location changes and behavior of the user during the day, across broader time horizons, and, for example, provide advertisements based on the User's online activity or location. On the other hand, as we said in Section 4.1, we have no options, and also we did not try to map the patterns and behavior to the specific users. We also believe that the analysis of patterns can be extended in more detail. During the pattern matching analysis, we only applied simple strategies to identify patterns. For example, as developments in machine learning and artificial intelligence progress, we believe that more sophisticated techniques from this field can be used to perform better and deeper analysis of patterns. Although the User Agent field is marked as deprecated and is starting to disappear from the *Initial* packets of the QUIC Protocol, it will definitely remain available for some time, especially on older devices. Furthermore, knowledge of this hypothetical tracking opportunity also provides insight into what was possible during times when User Agents were more commonly used. ‘

Conclusion

Within this work, we managed to create the QUIC plugin for the IPFIXProbe flow exporting tool, which can decrypt and extract TLS and QUIC-specific fields from the Initial Packets of the QUIC communication.

Based on the plugin, the Dataset consisting of several weeks was created, moreover, part of the Dataset was made publicly available in the journal publication *CESNET-QUIC22: A large one-month QUIC network traffic dataset from backbone lines* [21].

In the later stages of the thesis, we performed an analysis on the created dataset. The analysis essentially consists of two parts. The first part discusses the initial analysis and insights into the created dataset, focusing specifically on the User Agent and Server Name Indication fields. We evaluated the state of the dataset, identifying the most common User Agents, Server Name Indications, and also included the most common QUIC Versions. The second part of the analysis involves a deeper dive into the User Agent field. Although this field is marked as deprecated, it still appears in the captured flows. The User Agent field not only contains the used Chrome Version but also includes the Model Numbers of devices. Based on this, we formulated a hypothesis that the field, enriched by the Server Name Indication, can provide a unique identifier for a captured flow. We believe that this can open doors to location and activity tracking, especially for older devices that are not so common in the dataset.

The output of our work can be definitively split into two major contributions. Firstly, the QUIC plugin for the IPFIXProbe flow exporter can be freely used on high-speed networks. Although the Initial Packets of the QUIC protocol are marked as encrypted, we, and also other researchers, can now look into these Initial Packets and perform analyses on the fields contained within. This is supported by a recently merged Pull Request to IPFIXProbe[4] by our colleagues from Germany who were interested in extending the QUIC plugin with additional fields as seen in the Pull Request.

Secondly, the proposed hypothesis regarding the User Agent and Server Name Indication as unique identifiers can also be extended, and the analysis can be performed on a larger dataset with more sophisticated pattern-matching techniques. As also shown in Chapter3, there are still plenty of flows that contain the User Agent field, and we believe that it will take some time for this field to completely disappear.

[4]`https://github.com/CESNET/ipfixprobe/pull/194`

Bibliography

1. ROSKIND, Jim. *MULTIPLEXED STREAM TRANSPORT OVER UDP* [Working Draft]. 2012-04. Internet-Draft. Available also from: `https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit`. [cit. 2021-04-25].
2. ROSKIND, Jim. *Experimenting with QUIC*. 2013-06. Online. Available also from: `https://blog.chromium.org/2013/06/experimenting-with-quic.html`. [cit. 2021-04-25].
3. SIMON, Sergio De. *Google Will Propose QUIC As IETF Standard*. 2015-04. Online. Available also from: `https://www.infoq.com/news/2015/04/google-quic-ietf-standard/`. [cit. 2021-04-25].
4. IYENGAR, Janardhan; THOMSON, Martin. *QUIC: A UDP-Based Multiplexed and Secure Transport* [Working Draft]. 2016-11. Internet-Draft, draft-ietf-quic-transport-00. IETF Secretariat. Available also from: `https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-00`. [cit. 2021-04-25].
5. IYENGAR, Jana; THOMSON, Martin. *QUIC: A UDP-Based Multiplexed and Secure Transport* [RFC 9000]. RFC Editor, 2021. Request for Comments, no. 9000. Available from DOI: `10.17487/RFC9000`.
6. CIMPANU, Catalin. *HTTP-over-QUIC to be renamed HTTP/3*. 2018-11. Online. Available also from: `https://www.zdnet.com/article/http-over-quic-to-be-renamed-http3/`. [cit. 2021-04-25].
7. RODRIGUEZ, Luis. The Future of the Internet is Here: QUIC Protocol and HTTP/3. *Medium*. 2024. Available also from: `https://medium.com/@luisrodri/the-future-of-the-internet-is-here-quic-protocol-and-http-3-d7061adf424f`.
8. THOMSON, Martin; TURNER, Sean. *Using TLS to Secure QUIC* [RFC 9001]. RFC Editor, 2021. Request for Comments, no. 9001. Available from DOI: `10.17487/RFC9001`.
9. RESCORLA, Eric. *The Transport Layer Security (TLS) Protocol Version 1.3* [RFC 8446]. RFC Editor, 2018. Request for Comments, no. 8446. Available from DOI: `10.17487/RFC8446`.
10. IYENGAR, Jana; THOMSON, Martin. *QUIC: A UDP-Based Multiplexed and Secure Transport* [Working Draft]. 2021. Internet-Draft, draft-ietf-quic-transport-34. IETF Secretariat. Available also from: `http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-34.txt`. [cit. 2021-04-25].
11. MCGREW, David. *An Interface and Algorithms for Authenticated Encryption* [RFC 5116]. RFC Editor, 2008. Request for Comments, no. 5116. Available from DOI: `10.17487/RFC5116`.

12. GAGLIARDI, Eva; LEVILLAIN, Olivier. Analysis of QUIC Session Establishment and Its Implementations. *Information Security Theory and Practice Lecture Notes in Computer Science*. 2019, pp. 169–184. Available also from: `https://hal.archives-ouvertes.fr/hal-02468596/document`. [cit. 2021-04-25].

13. KRAWCZYK, Dr. Hugo; ERONEN, Pasi. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* [RFC 5869]. RFC Editor, 2010. Request for Comments, no. 5869. Available from DOI: `10.17487/RFC5869`.

14. BLAKE-WILSON, Simon; MIKKELSEN, Jan; NYSTRÖM, Magnus; HOPWOOD, David; WRIGHT, Tim. *Transport Layer Security (TLS) Extensions* [RFC 3546]. RFC Editor, 2003. Request for Comments, no. 3546. Available from DOI: `10.17487/RFC3546`.

15. FIELDING, Roy T.; RESCHKE, Julian. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* [RFC 7231]. RFC Editor, 2014. Request for Comments, no. 7231. Available from DOI: `10.17487/RFC7231`.

16. CLAISE, B.; TRAMMELL, B.; AITKEN, P. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information* [Internet Requests for Comments]. RFC Editor, 2013-09. STD, 77. RFC Editor. ISSN 2070-1721. Available also from: `http://www.rfc-editor.org/rfc/rfc7011.txt`. [cit. 2021-04-25].

17. CLAISE, B. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information* [Internet Requests for Comments]. RFC Editor, 2008-01. RFC, 5101. RFC Editor. ISSN 2070-1721. Available also from: `http://www.rfc-editor.org/rfc/rfc5101.txt`. [cit. 2021-04-25].

18. IANA. *IP Flow Information Export (IPFIX) Entities*. 2007-05. Online. Available also from: `https://www.iana.org/assignments/ipfix/ipfix.xml`. [cit. 2021-04-25].

19. YANG, Feng. The tale of deep packet inspection in China: Mind the gap. In: *2015 3rd International Conference on Information and Communication Technology (ICoICT)*. 2015, pp. 348–351. Available from DOI: `10.1109/ICoICT.2015.7231449`. [cit. 2021-04-25].

20. THOMSON, Martin; TURNER, Sean. *Using TLS to Secure QUIC*. Internet Engineering Task Force, 2020-10. Internet-Draft, draft-ietf-quic-tls-32. Internet Engineering Task Force. Available also from: `https://datatracker.ietf.org/doc/draft-ietf-quic-tls/32/`. Work in Progress.

21. LUXEMBURK, Jan; HYNEK, Karel; ČEJKA, Tomáš; LUKAČOVIČ, Andrej; ŠIŠKA, Pavel. CESNET-QUIC22: A large one-month QUIC network traffic dataset from backbone lines. *Data in Brief*. 2023, vol. 46, p. 108888. ISSN 2352-3409. Available from DOI: `https://doi.org/10.1016/j.dib.2023.108888`.

22. IYENGAR, Jana; THOMSON, Martin. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet Engineering Task Force, 2020-06. Internet-Draft, draft-ietf-quic-transport-29. Internet Engineering Task Force. Available also from: `https://datatracker.ietf.org/doc/draft-ietf-quic-transport/29/`. Work in Progress.

Contents of attached medium

```
dataset ........................ used data that does not include personal information
  quic-dataset.pickle ........................ dataset output of the QUIC Plugin
src
  plugin ........................................ source code of resulting QUIC Plugin
    quic.cpp .......................................... IPFIXProbe plugin header file
    quic.hpp .......................................... IPFIXProbe plugin source file
    quic_parser.cpp ........................... header file for the QUIC parsing logic
    quic_parser.hpp ........................... source file for the QUIC parsing logic
  analysis ............................................ source codes used during analysis
    ipinfo-script.ipynb ...................... source code for obtaining IPInfo data
    quic-analysis.ipynb ...................... source code for performed analysis
  thesis ..................................................... LaTeX thesis source code
text ................................................................... thesis text
  thesis.pdf ................................................... PDF format of thesis
readme.txt ............................................... overview of the contents
```