Analysis of QUIC session establishment and its implementations

Eva Gagliardi, Olivier Levillain

Analysis of QUIC Session Establishment and its Implementations

Eva Gagliardi[1] and Olivier Levillain[2]

[1] French Ministry of the Armies,
[2] Télécom SudParis, Institut Polytechnique de Paris

Abstract. In the recent years, the major web companies have been working to improve the user experience and to secure the communications between their users and the services they provide. QUIC is such an initiative, and it is currently being designed by the IETF. In a nutshell, QUIC originally intended to merge features from TCP/SCTP, TLS 1.3 and HTTP/2 into one big protocol. The current specification proposes a more modular definition, where each feature (transport, cryptography, application, packet reemission) are defined in separate internet drafts.
We studied the QUIC internet drafts related to the transport and cryptographic layers, from version 18 to version 23, and focused on the connection establishment with existing implementations.
We propose a first implementation of QUIC connection establishment using Scapy, which allowed us to forge a critical opinion of the current specification, with a special focus on the induced difficulties in the implementation. With our simple stack, we also tested the behaviour of the existing implementations with regards to security-related constraints (explicit or implicit) from the internet drafts. This gives us an interesting view of the state of QUIC implementations.

Keywords: QUIC · Secure communications · Protocol implementation.

1 Introduction

In the recent years, the major web companies have been working to improve the user experience and to secure the communications between their users and the services they provide. One of this effort was QUIC, proposed by Google in 2012. Another change in parallel was the standardization of TLS 1.3[3], which both achieves better performance, with a faster session establishment, and better security, since only up-to-date and secure primitives were kept in this new version of the protocol.

However, even with TLS 1.3 and HTTP/2, the TLS/HTTP combination is still considered a bottleneck by some actors. So the development of QUIC went on, and Google proposed their protocol to the IETF for a standardization. A working group was formed and since 2016, 23 draft versions of the protocol have been discussed. The original protocol has since been renamed gQUIC (for Google

[3] Actually, TLS 1.3 borrowed several ideas from the initial QUIC design.

QUIC). In the remainder of this article, QUIC refers to the IETF version of the protocol, which differs significantly from gQUIC. Indeed, the IETF version offers a more modular protocol than the original proposal. QUIC design relies on the following architecture:

- A transport layer is defined in the `quic-transport` internet draft [4] over UDP. This way, QUIC avoids the delay induced by the TCP three-way handshake, but obviously has to handle packet loss and reordering.
- During the session establishment, cryptographic parameters and keys are negotiated using TLS 1.3 Handshake message. The way QUIC embeds and interacts with TLS is described in the `quic-tls` internet draft [11].
- On top of the transport layer, a new version of the HTTP protocol is being proposed, HTTP/3, which will be designed for QUIC [2].

The working group also wrote several peripheral internet drafts to specify generic properties for QUIC [7, 10] or to give details on specific features [3, 6]. In this article, we focus on the establishment phase described by `quic-transport` and `quic-tls` drafts.

Section 2 describes the QUIC protocol and section 3 details the protection mechanism used to encrypt QUIC packets. For our study, we implemented parts of the protocol with Scapy; section 4 presents the challenges we had to face to interact with existing QUIC stacks. Using our tool, we ran some tests to study the behaviour of public servers with regards to security-related constraints (explicit or implicit) from the internet drafts; section 5 describes the test bench while section 6 contains the obtained results. Related work is presented in section 7 before our conclusion.

2 QUIC in a Nutshell

The message flow of a typical QUIC connection is given in Figure 1. First, the client sends an Initial packet, which includes a TLS 1.3 ClientHello. If the enclosed (QUIC and TLS) parameters are acceptable for the server, it answers with an Initial packet (including the TLS ServerHello). This message is followed by a Handshake packet including the rest of the TLS server messages (in particular the messages related to server authentication). The handshake ends with a message from the client. Then, application data can be exchanged using so-called 1-RTT packets. The three phases, corresponding to different packet types (Initial, Handshake, 1-RTT) correspond to the three cryptographic epochs used in TLS 1.3 (cleartext messages, protection using Handshake secrets, protection using Traffic secrets), with the notable exception that Initial packets are actually encrypted using publicly-available data (we explore this in section 3.1).

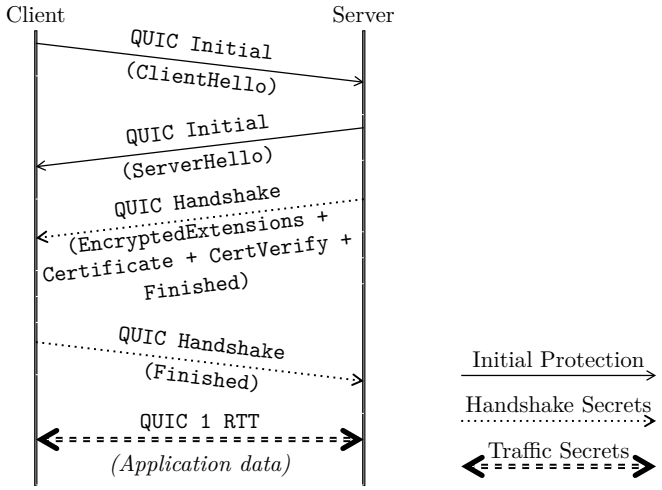

Fig. 1. A typical QUIC connection. The TLS 1.3 messages encapsulated in CRYPTO frames are given in parentheses. ACK and Padding frames have been left out for clarity.

2.1 QUIC Main Goals and Features

The QUIC protocol aims at providing an efficient and secure channel for application data. The efficiency properties include:

- **Low-latency session establishment**. As shown in Figure 1, a typical connection allows the client to send application data to the server after only 1-RTT[4], whereas TLS 1.2 usually requires 3 (including the TCP handshake) and TLS 1.3 typically requires 2. Moreover, when connecting to a known server, a client can benefit from TLS 1.3 0-RTT feature to send application data in its first datagram (whereas TLS 1.3 still requires the RTT induced by the TCP handshake).
- **Stream multiplexing within a shared connection**. 1-RTT packets (as well as 0-RTT packets) include application data which are associated with streams. From the QUIC point of view, these streams are independant and can be multiplexed in QUIC packets using the client and server policies. This feature (also present in HTTP/2) solves the so-called Head of Line blocking issue from HTTP/1.1 pipelining where you must wait for the end of a request to emit the next one.
- **Low bandwidth usage**. The message design in QUIC was made to limit the bandwidth usage of the signaling and transport structures. For example, the draft uses several variable-length fields to limit their sizes. It also defines a padding scheme without any overhead (in case padding is not used).

[4] The session establishment latency is usually measured in RTTs (Round-Trip Time), that is the time required for the client to send a request and get an answer.

The security properties rely on:

- **State-of-the-art cryptographic primitives**. This point is granted by the use of TLS 1.3, which was designed to clean up the cryptographic zoo accumulated for more than 20 years and only uses up-to-date and robust schemes.
- **Privacy-oriented measures**. QUIC offers a padding feature to avoid traffic analysis, and most of QUIC packet contents are encrypted and integrity-protected. However, as discussed in section 3.1, even if Initial packets are encrypted, this mechanism offers no protection in typical attacker models.
- **Countermeasures against denial-of-service attacks**. Since QUIC uses UDP, it is essential not to enable or encourage amplification attacks where an attacker would send a small packet to a server with a forged source IP address, expecting a much larger answer to be sent to the victim. To this aim, before the session has been established, there are constraints on the size of the data the server can send. Moreover, QUIC allows the server to validate the client address before the session establishment (via the so-called Retry mechanism).

Another goal for the IETF working group is for QUIC to be compatible with the internet. In particular, the working group has to face so-called middleboxes, network devices that may intercept or block traffic at different places of the internet[5]. This goal led to the definition of several QUIC invariants [10], which should be taken into account by middleboxes. It also led to encrypting as much as possible, including Initial packets, to make a QUIC packet as hard as possible to grasp for a piece of equipment unaware of a particular version of QUIC.

3 QUIC Packet Protection

Almost every QUIC packet follows the steps described in Figure 2 to encrypt both the payload and parts of the header. Moreover, since the header is fed as Associated Data to the AEAD (Authenticated Encryption with Associated Data) algorithms, both header and payload are integrity-protected.

To protect a packet, the header is first isolated from the payload. Then, the payload is encrypted using the negotiated AEAD. It takes as input the plaintext payload, a key derived from the key exchange, and a nonce (which comes from the XOR of the packet number from the header with an IV also derived from the key exchange).

Then, part of the payload is sampled and used as input to an encryption algorithm (in typical setups, the sample is 16 bit long and is encrypted with AES-ECB). The resulting ciphertext is used to mask (with a XOR) several fields of the header.

This convoluted procedure aims at protecting several fields in the header, such as the Packet Number.

[5] These middleboxes were a real problem during the definition of TLS 1.3 and the TLS working group actually decided to include optional dummy messages in the message flow to accommodate them.

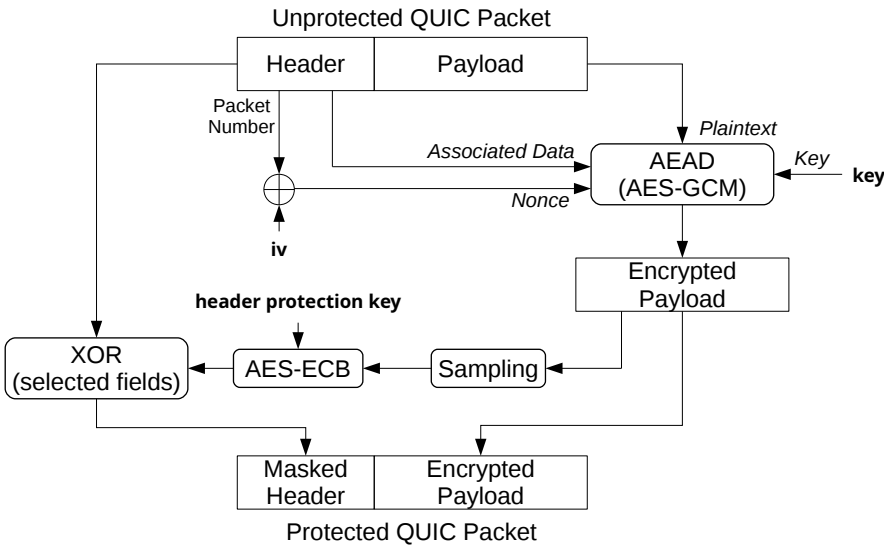

Fig. 2. QUIC packet protection mechanism. The inputs are the packet to protect, the `key` and the `iv` used to encrypt the payload, and the `header protection key`.

3.1 The Special Case of Initial Packets

There is however an egg-and-chicken problem with Initial packets, since they are supposed to be protected, but they contain the key exchange messages which should provide the keying material.

Actually, Initial packets must be protected, but the used parameters are defined by the RFC and one field from the client Initial packet:

- the AEAD used to protect the payload is `AEAD_AES_128_GCM`;
- the Initial secret (from which the key, the IV and the header protection key are derived), is derived from the so-called salt, a constant defined in the specification for a given version of the protocol, and the Destination Connection ID (DCID) embedded in the client Initial packet.

This DCID is actually only sent in the first packet, since each endpoint is responsible for the definition of its own Connection ID (which can be void). Thus, a server would typically answer with an Initial message with a freshly generated Source Connection ID and the DCID chosen by the client (in the Source Connection ID field of the first packet).

It must be clearly stated that this mechanism offers absolutely no protection from an attacker able to observe the first packet sent by the client. The draft indeed states that "[t]his provides protection against off-path attackers and robustness against QUIC version unaware middleboxes, but not against on-path

attackers." The part about robustness refers to the idea that middleboxes unaware of a given QUIC version will not know the corresponding salt and will not be able to inspect the packet. We strongly believe that this is a naive reasonning, and that middleboxes will nevertheless try and decrypt and inspect the packet, which will most certainly lead to reject the packet or report an incident in typical cases. From our point of view, protecting initial packets is a useless mechanism that provides no security in practice.

3.2 Header Protection keys

The `hp` key, used to encrypt selected fields from the header, is generated from the Initial secret, and "is used for the duration of the connection, with the value not changing after a key update." Thus, if an attacker is able to observe the client first packet, she can easily remove the header protection for the whole connection. Since the header protection includes a somewhat great complexity, for a very small benefit, we wonder whether the trade-off is well balanced.

Moreover, the specification is unclear on how to protect the header when a Chacha20- or an AES-256-based ciphersuite is selected during the handshake. Indeed, the initial (and only) header protection key is supposed to be 16-byte long. Yet, when using Chacha20 or AES-256, a 256-bit key (32 bytes) is expected. How should we reconcile this?

4 Implementation of the Initial Exchange

To better assess the reality of the message protection scheme, we implemented a portion of the QUIC protocol in Scapy, a Python framework used to dissect and forge packets for various network protocols [1]. Appendix A presents excerpts of our implementation.

What struck us during this work was the complexity of the mechanism, especially for the client initial packet. Indeed, protecting a packet corresponds to the following sequence (step 5 is only required for the first Initial packet):

1. build[6] the header from its fields;
2. build the payload from its fields;
3. pad the payload so the packet size is long enough;
4. report the payload length in the header to take the padding into account;
5. derive secrets from the version and the DCID;
6. derive the nonce from the IV (derived during the previous step) and the Packet Number (from the header);
7. encrypt the payload;
8. extract the sample;
9. encrypt the header.

[6] We use the term *build* to describe the production of a byte string from the abstract structure manipulated by the rest of the application. It is the reverse operation of the binary parsing, and is sometimes called unparsing, dumping, or serializing.

The corresponding actions to unprotect a received packet are the following (step 2 is only needed to handle the client initial packet):

1. parse the first fields of the header;
2. derive secrets from the version and the DCID;
3. extract the sample from the payload, assuming the Packet Number Length is 4 (more on this later);
4. decrypt the Packet Number Length;
5. infer the real offset/length of the Packet Number field and of the payload;
6. decrypt the Packet Number;
7. derive the nonce from the IV and the Packet Number;
8. decrypt the payload.

Even if these description are very detailed and even if some of our difficulties might be related to the way Scapy works, we strongly believe the sequence is inherently complex. Focusing on the protection procedure, it mixes classical building steps (steps 1 and 2), cryptographic operations (steps 5, 6, 7 and 9), but also raw manipulations of the binary packet (steps 3, 4 and 8[7]). Such manipulations are highly undesirable from a software engineering point of view, especially when they are intertwined with cryptographic or parsing/building steps.

Moreover, the manipulation steps are really hard to get right. For example, updating the payload length in the header requires identifying the offset of this specific field (which is not fixed) and encoding the new length using a variable length field: the precise length of the packet may be different after this update!

Another example of the complexity induced by the specification: since the Packet Number Length is encrypted, there is no way for the receiver to establish where the payload actually starts. This is why the sample required to encrypt the header is not computed from the start of the payload, but from what would be the first byte of the payload, assuming the Packet Number Length is 4 (this means a shift of 0 to 3 bytes).

Overall, the QUIC design forces developers to write so-called shotgun parsers, that is parsers which mix several kind of operations (parsing, input-validating code, processing code) [9], whereas a cleaner design would lead to a simpler and more straightforward implementation.

5 Test Description

To better understand the emerging QUIC ecosystem, we then looked at the existing implementations in the wild, as listed on the QUIC Working Group wiki[8]. During our study, which spanned over several months and followed drafts 18 to 23, we contacted around 20 public servers, corresponding to 16 different implementations. To investigate several configurations further, we also installed several implementations locally.

[7] As a matter of fact, since header encryption (step 9) is not a straightforward XOR on a clearly delimited message, this could also be considered as a raw manipulation.

[8] https://github.com/quicwg/base-drafts/wiki/Implementations

Table 1 describe the implementations we considered and their availability in October 2019. Out of the 16 public servers, 10 were available and up to date after the draft-23 publication.

Implem.	Test server	Comments
aioquic	quic.aiortc.org:443	**OK (draft-23)**
ats	quic.ogre.com:4443	**OK (draft-23)**
f5	204.134.187.194:4433	No answer (latest draft: -22)
lsquic	http3-test.litespeedtech.com:4433	No complete Handshake
mozquic	mozquic.ducksong.com:4433	No answer (latest draft: -12)
msquic	quic.westus.cloudapp.azure.com:4433	No complete Handshake
mvfst	fb.mvfst.net:4433	**OK (draft-23)**
ngtcp2	nghttp2.org:4433	**OK (draft-23)**
ngx_quic	cloudflare-quic.com:443	**OK (draft-23)**
Pandora	pandora.cm.in.tum.de:4433	**OK (draft-23)**
picoquic	test.privateoctopus.com:4433	**OK (draft-23)**
quant	quant.eggert.org:4433	**OK (draft-23)**
quiche	quic.tech:4433	**OK (draft-23)**
QUICker	quicker.edm.uhasselt.be:4433	No answer (latest draft: -20)
quicly	quic.examp1e.net:4433	No complete Handshake
Quinn	ralith.com:4433	**OK (draft-23)**

Table 1. List of the servers we probed during our study and their status in October 2019 when facing a draft-23 Client Initial packet. Two servers never answered to our stimuli during the whole study (`mozquic` and `QUICker`), which might be explained by the fact that their development seems to be on hold. For the results described in this article, we will only consider the 10 servers we could connect to properly during our latest tests (after draft-23 publication).

Indeed, one major difficulty we faced during our tests was that public servers would randomly go down and stop answering to our stimuli. The problem was especially visible each time a new draft was published.

To test the behaviour of these implementations, we sent different stimuli. The baseline was a valid QUIC Client Initial Packet corresponding to the latest version[9]. Then, we sent variations around this first stimulus:

- packets with a future version of the protocol, some of them being partly incompatible with the current wire format;
- packets not respecting the constraints on Client Initial Packet length;
- packets missing mandatory information (QUIC transport parameters, TLS Application-Layer Protocol Negotiation extension);
- packets containing forbidden frame types;
- packets with mangled CRYPTO frames.

[9] To be precise, we actually sent several valid stimuli, to accommodate with minor quirks with the ALPN extension, as described in section 6.3.

6 Results

For this section, we chose to use the latest results, which correspond to the 23rd version of the drafts, published in September 2019. As explained in the previous section, due to the unavailability of several servers, we could only scan 10 implementations in a reliable way before the submission.

Moreover, it is important to keep in mind that the tested implementations, as well as the specifications, are still works in progress, and that the results presented here are only a snapshot of a fast-evolving ecosystem. Our goal is thus *not* to blame a given QUIC stack for possible deviations with regards to the draft (or its spirit, in case of implicit constraints), but to draw the attention on possible issues, which are the consequence of a complex protocol.

6.1 Version Negotiation

The QUIC specification aims at describing a robust protocol able to survive future changes of the concrete representation of messages on the wire. This is why the beginning of a QUIC packet is defined in a document called "QUIC Invariants" [10]: the long header should always look like the definition in Figure 3.

It is important to notice in particular that the payload length is *not* part of this definition. Thus, a QUIC packet advertising a new version should be able to redefine how the packet length is specified. This is why we sent three different stimuli to the test servers: a standard valid draft23-compatible Initial packet, a similar packet advertising a yet-to-be-defined version, and a similar packet advertising the same future version but with the current `Length` field set to a huge value. Since the length should not be parsed for unknown versions, we expect compliant implementations to answer the first stimulus with a valid handshake (an Initial packet followed by Handshake packets) and the two other stimuli with a Version Negotiation message, asking the client to re-emit its packet using a version of the protocol supported by the server.

The majority of the contacted servers actually behaved this way, but we also witnessed one implementation (see Table 2, `ngtcp2` implementation) that answered correctly with a Version Negotiation message when our stimulus contained a *correct* length, while timing out when the length was *incorrect*. This is a violation of the invariants as described in the specifications.

As a side note, it is interesting that we discover this behaviour by accident after a change in the draft describing the invariants when draft-22 was published. Indeed, in July 2019, the working group decided to change the way Connection ID length was sent on the wire[10]. Since we studied both pre-draft-22 and draft-22 implementations at the time, we triggered the incorrect behaviour with recent versions choking an on old stimulus (or the other way around).

[10] We let the reader reflect on the introduction of a *change* in a document describing the protocol *invariants*. Even though this was a bit unsettling, let us recall that this change was a simplification in the design and that QUIC documents are still drafts.

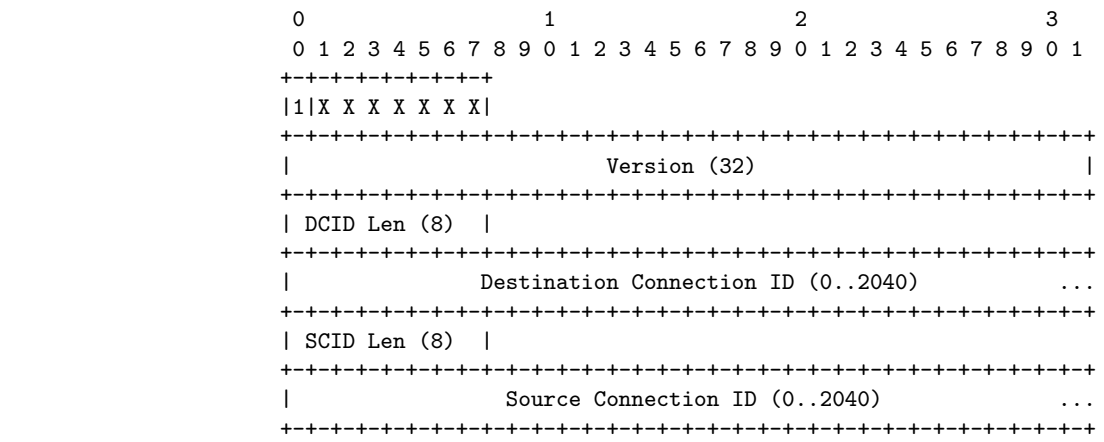

Fig. 3. Description of the first fields of any (long-header) QUIC packet, as defined in the "QUIC Invariants" Internet draft [10].

Implem.	Reaction to a Future version with	
	correct length	incorrect length
Expected	Version Negotiation	Version Negotiation
`aioquic`	Version Negotiation	Version Negotiation
`ats`	Version Negotiation	Version Negotiation
`mvfst`	Version Negotiation	Version Negotiation
`ngtcp2`	Version Negotiation	**Time Out**
`ngx_quic`	Version Negotiation	Version Negotiation
`Pandora`	Version Negotiation	Version Negotiation
`picoquic`	Version Negotiation	Version Negotiation
`quant`	Version Negotiation	Version Negotiation
`quiche`	Version Negotiation	Version Negotiation
`Quinn`	Version Negotiation	Version Negotiation

Table 2. Reaction of the servers selected in the previous section to an initial packets advertising a future version of the protocol. The first one presents a correct length field (with regards to the current specification) while the second presents a bigger length. The Time Out in the second column corresponds to a server waiting for what it interprets as missing bytes.

6.2 Client Initial Packet Length

Since QUIC uses UDP, it is inherently subject to reflection attacks, where an attacker sends a packet with a forged source address, leading the server to answer to the victim. In some cases, the attacker can trigger a huge amount of data using a small packet. These so-called amplification attacks may lead to denial of service situations.

To avoid such attacks, QUIC specifies that a client should send at least a 1,200-byte long initial packet, and that a server should never answer with more than three times the amount of data the client initially sent. Moreover, a server should ignored a client Initial packet which is too small. The combination of these constraints allows the server to send up to 3,600 bytes in its first flight, which is considered sufficient.

To check how servers behaved regarding these constraints, we sent small stimuli, and observed the reaction of the public servers. Several implementations actually answered our invalid packet, as shown in Table 3. The exact implementations that were affected did vary across time, but we also always observed that the server answer was capped at three times the size of the client Initial packet, which at least limited the amplification impact, as planned.

6.3 Missing Parameters

Scattered across the specifications, several parameters of the client Initial packet are described as mandatory. In particular, the TLS 1.3 ClientHello must contain an extension dedicated to QUIC to define the initial values of several transport parameters (e.g. to define the maximum size of the exchanged packets) and the ALPN extension (which defines the nature of the protocol encapsulated in QUIC).

We found out that several implementations seemed to accept a stimulus missing these elements, and in the case of ALPN, we even found implementations that only answered when the extension was missing. The situation might not be a problem after all, since we only looked at the first messages of the connections, and what seemed to be a valid connection might then be shut down by the server when handling the application layer.

Yet, we believe errors should be triggered as soon as possible, both to avoid useless resource usage and to make debugging easier. Indeed, several implementations return an empty error packet when some parameters are missing (or do not correspond to the expected values), and the only way to understand what is happening is to have access to the server logs, or to compare the behaviour of a given server with different stimuli.

6.4 Frame Mangling

Another venue we investigated was to send forbidden frames to the servers. The specification indicates that the only frames that should be sent in an Initial packet are crypto frames (which embed TLS messages), acknowledgement (ACK)

Implem.	Reaction to a small Initial packet
Expected	Time Out
aioquic	Time Out
ats	**Handshake (886 bytes)**
mvfst	Time Out
ngtcp2	Time Out
ngx_quic	Time Out
Pandora	Time Out
picoquic	Time Out
quant	Time Out
quiche	**Handshake (896 bytes)**
Quinn	Time Out

Table 3. Reaction of the selected servers to a small initial packet (300 bytes). Even if several implementations answer with the beginning of a Handshake, they respect the constraint not to send more than three times the amount of data initially received.

Implem.	Ping frame	Split Crypto	Overlapping Crypto frames	
			consistent	inconsistent
Expected	Error	Error	Error	Error
aioquic	**Handshake**	**Handshake**	**Handshake**	**Handshake**
ats	**Handshake**	**Handshake**	Error	Error
mvfst	Error	**Handshake**	**Handshake**	**Handshake**
ngtcp2	**Handshake**	**Handshake**	**Handshake**	**Handshake**
ngx_quic	Error	**Handshake**	**Handshake**	**Handshake**
Pandora	Error	Time Out	Error	Error
picoquic	Time Out	**Handshake**	**Handshake**	**Handshake**
quant	Error	Error	Error	Error
quiche	Error	**Handshake**	**Handshake**	**Handshake**
Quinn	Error	**Handshake**	**Handshake**	**Handshake**

Table 4. Reaction of the selected servers to a initial packets containing strange frames. Behaviours in bold are unexpected ones.

frames, Connection Close frames (which signal errors) and padding frames. However, we observed that several servers would accept a Ping frame enclosed in the first client Initial packet[11]. Again, we would expect the servers to be stricter with the messages they accept.

We also tried to split the TLS ClientHello across two Crypto frames, which should be rejected by implementations, since the specification states that "[t]he first packet sent by a client always includes a CRYPTO frame that contains the entirety of the first cryptographic handshake message.". Most of the implementations nevertheless answered our stimulus.

Finally, we sent packets with two overlapping Crypto frames (bytes 0 to 149, followed by bytes 50 to the end), first where both fragments would contain the same content, and then with a glitch introduced in the first fragment[12]. We thus observed that most of the servers tolerated overlapping frames, including when they were inconsistent. There is no obvious way to directly exploit this behaviour, but we found this a bit unsettling, and would advocate a stricter set of rules in the implementations.

Table 4 summarises these experiments on frame mangling.

7 Related Work

QUIC is a relatively new protocol, and most of the literature related to QUIC is about gQUIC. For example, Jager et al. showed how to exploit the Bleichenbacher attack against RSA Encryption to forge a signature and bypass server authentication in Google QUIC [5].

More recently, McMillan and Zuck presented a modeling of QUIC to test the state machines of existing implementations [8]. We believe our approaches are complementary since we propose a (partial) concrete test bench, whereas they validate implementations at a more abstract level. Their work showed in particular the existence of ambiguities in the specification, which our measurements seem to confirm, when we look at the diversity of behaviours in the existing implementations.

An online tool, QUIC Tracker[13], describes a test suite regarding QUIC features, and shows the reaction of existing implementations. Yet, QUIC Tracker seems to only look at features whereas we believe measuring the conformance to specific constraints from the specification would be of great help.

8 Conclusion and Perspectives

QUIC is a relatively recent protocols aiming at improving the efficiency and security of the web. As of today, it is still a work in progress, which reflects on

[11] As a side note, it appears that placing the Ping frame after instead of before the Crypto frame gets the stimulus accepted by one more server.

[12] We also tried with the glitch on the second fragment, but we mostly obtained errors from the servers.

[13] https://quic-tracker.info.ucl.ac.be

the stability and robustness of the implementations. In our work, we focused on the initial negotiation phase of the protocol, and how to implement it.

We assessed the complexity in practice of the QUIC protection mechanisms by writing a Scapy implementation. We learned that QUIC is a complex beast and we believe it would be useful to simplify several aspects of the specification which are not justified in our mind. We already discussed with the IETF Working Group of several aspects of our findings and plan to continue this interaction.

We proposed a first framework to send stimuli to servers and observe their behaviour during the session establishment. Obviously, it would be useful to pursue this effort and propose more elaborate scenarios to test other features, e.g. address migration or 0-RTT data exchanges.

In the end, QUIC is a very complex protocol, and this complexity will certainly lead to implementation bugs. Indeed, the current situation is far from perfect, since most of the studied implementations do not conform to the specification on several aspects, and some of these aspects could be the first step towards a complex attack.

References

1. Biondi, P., the Scapy community: Scapy. http://www.secdev.org/projects/scapy/ (2003-2016), http://www.secdev.org/projects/scapy/
2. Bishop, M.: Hypertext Transfer Protocol Version 3 (HTTP/3). Internet-Draft draft-ietf-quic-http-23, Internet Engineering Task Force (Sep 2019), https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-23, Work in Progress
3. Iyengar, J., Swett, I.: QUIC Loss Detection and Congestion Control. Internet-Draft draft-ietf-quic-recovery-23, Internet Engineering Task Force (Sep 2019), https://datatracker.ietf.org/doc/html/draft-ietf-quic-recovery-23, Work in Progress
4. Iyengar, J., Thomson, M.: QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-23, Internet Engineering Task Force (Sep 2019), https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-23, Work in Progress
5. Jager, T., Schwenk, J., Somorovsky, J.: On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 v1.5 Encryption. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 1185–1196 (2015)
6. Krasic, C.B., Bishop, M., Frindell, A.: QPACK: Header Compression for HTTP/3. Internet-Draft draft-ietf-quic-qpack-10, Internet Engineering Task Force (Sep 2019), https://datatracker.ietf.org/doc/html/draft-ietf-quic-qpack-10, work in Progress
7. Kühlewind, M., Trammell, B.: Applicability of the QUIC Transport Protocol. Internet-Draft draft-ietf-quic-applicability-05, Internet Engineering Task Force (Jul 2019), https://datatracker.ietf.org/doc/html/draft-ietf-quic-applicability-05, work in Progress
8. McMillan, K.L., Zuck, L.D.: Formal specification and testing of QUIC. In: Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019. pp. 227–240 (2019)

9. Momot, F., Bratus, S., Hallberg, S.M., Patterson, M.L.: The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. In: IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016. pp. 45–52 (2016)
10. Thomson, M.: Version-Independent Properties of QUIC. Internet-Draft draft-ietf-quic-invariants-07, Internet Engineering Task Force (Sep 2019), https://datatracker.ietf.org/doc/html/draft-ietf-quic-invariants-07, Work in Progress
11. Thomson, M., Turner, S.: Using TLS to Secure QUIC. Internet-Draft draft-ietf-quic-tls-23, Internet Engineering Task Force (Sep 2019), https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-23, Work in Progress

A Scapy Implementation

The description of a QUIC packet in Scapy can be done as shown in the following extract:

```
class QUIC(Packet):
    fields_desc = [
        # Flags
        BitEnumField("header_type", 1, 1, {0: "short", 1: "long"}),
        BitEnumField("fixed_bit", 1, 1, {0: "error", 1: "1"}),
        BitEnumField("type", 0, 2, {0: "initial", 1: "0-RTT",
                                    2: "handshake", 3: "retry"} ),
        BitField("reserved", 0, 2),
        BitFieldLenField("PNL", None, 2, length_of="PN",
                         adjust=lambda pkt,x:x-1),

        # Version
        XIntField("version", 0x0),

        # Connection IDs (DCID/SCID)
        BitFieldLenField("DCIL", None, 8, length_of="DCID"),
        StrLenField("DCID", b'', length_from=lambda pkt:pkt.DCIL),

        BitFieldLenField("SCIL", None, 8, length_of="SCID"),
        StrLenField("SCID", b'', length_from=lambda pkt:pkt.SCIL),

        # Token (only when type is initial)
        ConditionalField(QuicVarLenField("token_length", None,
                         length_of="token"),
                         lambda pkt: pkt.version != 0 and pkt.type == 0),
        ConditionalField(StrLenField("token", b'',
                         length_from = lambda pkt:pkt.token_length),
                         lambda pkt: pkt.version != 0 and pkt.type == 0),
        # Length (only when type is 0-RTT or initial)
        ConditionalField(QuicVarLenField("length", None),
                         lambda pkt: pkt.version != 0 and pkt.type != 3),

        # Packet Number (only when type is 0-RTT or initial)
        ConditionalField(StrLenField("PN", b'\x00',
                         length_from = lambda pkt:pkt.PNL+1),
                         lambda pkt: pkt.version != 0 and pkt.type != 3),
    ]
```

Of course, since most fields are only valid for specific types of QUIC packets, we need to determine the presence most of the fields by the presence of certain values before in the packet.

To apply packet protection, we had to write dedicated functions. The following excerpt shows a simplified version of the protection function, which takes a QUIC packet as input and produces the byte string that can be sent on the wire.

```
def protect(material, packet):
    (key, iv, hp) = material
    header = packet.copy()
    header.payload = Raw()
    payload = packet[1]

    # Compute nonce
    nonce = int.from_bytes(iv, byteorder='big') ^
            int.from_bytes(header.PN, byteorder='big')
    nonce = nonce.to_bytes(12, byteorder = 'big')

    # Encrypt the payload
    encryptor = Cipher(algorithms.AES(key), modes.GCM(nonce),
                       backend=default_backend()).encryptor()
    encryptor.authenticate_additional_data(raw(header))
    encrypted_payload = encryptor.update(raw(payload)) +
                        encryptor.finalize() + encryptor.tag

    # Extract the sample
    PNL = header.PNL + 1
    sample_start = 4 - PNL    # The receiver will assume PNL is 4
    sample = encrypted_payload[sample_start:sample_start + 16]

    # Compute the mask
    encryptor = Cipher(algorithms.AES(hp), modes.ECB(),
                       backend=default_backend()).encryptor()
    mask = encryptor.update(sample) + encryptor.finalize()

    # Encrypt the flags and the PN
    encrypted_header = bytearray(raw(header))
    encrypted_header[0] ^= (mask[0] & 0x0f)
    for i in range(PNL):
        encrypted_header[-PNL + i] ^= mask[i+1]
    encrypted_header = bytes(encrypted_header)

    return encrypted_header + encrypted_payload
```

Our implementation could be improved by the addition of a Scapy automaton to handle the QUIC state machine and its transitions. However, we must keep in mind that our goal was to send possibly non-conformant stimuli to servers, so we might want not to follow the expected state machine all the time in our future work.