Revisiting QUIC attacks: A comprehensive review on QUIC security and a hands-on study

Preprint · July 2022

DOI: 10.21203/rs.3.rs-1676730/v1

4 authors:

Efstratios Chatzoglou
University of the Aegean
31 PUBLICATIONS **471** CITATIONS
SEE PROFILE

Vasileios Kouliaridis
European Commission
24 PUBLICATIONS **543** CITATIONS
SEE PROFILE

Georgios Karopoulos
European Commission
52 PUBLICATIONS **708** CITATIONS
SEE PROFILE

Georgios Kambourakis
University of the Aegean
215 PUBLICATIONS **7,863** CITATIONS
SEE PROFILE

Revisiting QUIC attacks: A comprehensive review on QUIC security and a hands-on study

Efstratios Chatzoglou (efchatzoglou@gmail.com)
University of the Aegean: Panepistemio Aigaiou https://orcid.org/0000-0001-6507-5052

Vasileios Kouliaridis
Aegean University: Panepistemio Aigaiou

Georgios Karopoulos
European Commission Joint Research Centre

Georgios Kambourakis
Aegean University: Panepistemio Aigaiou

Research Article

Keywords: QUIC, TLS 1.3, HTTP/3, Network Security, Vulnerabilities, Fuzz testing

Posted Date: July 1st, 2022

DOI: https://doi.org/10.21203/rs.3.rs-1676730/v1

Revisiting QUIC attacks: A comprehensive review on QUIC security and a hands-on study

Efstratios Chatzoglou[1], **Vasileios Kouliaridis**[1], **Georgios Karopoulos**[2], **Georgios Kambourakis**[a,2]

[1]Department of Information & Communication Systems Engineering, University of the Aegean, Karlovasi 83200, Samos, Greece; efchatzoglou@gmail.com (E.C.); bkouliaridis@aegean.gr (V.K.)
[2]European Commission, Joint Research Centre, Ispra 21027, Italy; georgios.karopoulos@ec.europa.eu (G.K.);

Received: date / Accepted: date

Abstract Built on top of UDP, the recently standardized QUIC protocol primarily aims to gradually replace the TCP plus TLS plus HTTP/2 model. For instance, HTTP/3 is designed to exploit QUIC's features, including reduced connection establishment time, multiplexing without head of line blocking, always-encrypted end-to-end security, and others. This work serves two key objectives. Initially, it offers the first to our knowledge full-fledged review on QUIC security as seen through the lens of the relevant literature so far. Second and more importantly, through extensive fuzz testing, we conduct a hands-on security evaluation against the six most popular QUIC-enabled production-grade servers. This assessment identified several effective and practical zero-day vulnerabilities, which, if exploited, can quickly overwhelm the server resources. This finding is a clear indication that the fragmented production-level implementations of this contemporary protocol are not yet mature enough. Overall, the work at hand provides the first wholemeal appraisal of QUIC security from both a literature review and empirical standpoint, and it is therefore foreseen to serve as a reference for future research in this timely area.

Keywords QUIC · TLS 1.3 · HTTP/3 · Network Security · Vulnerabilities · Fuzz testing.

1 Introduction

The primary goal of the QUIC transport layer protocol is to improve performance, since although the HTTP plus TCP plus TLS model served the web well for about 15 years, it became apparent that it could no longer sustain the emerging bandwidth-demanding and latency-sensitive web applications. For instance, one of the major issues of HTTP/1.1 is HTTP Head-of-Line (HoL) blocking, which causes increased response delays when a HTTP request can delay all the subsequent requests within the same TCP connection.

To improve HTTP performance, back in 2009, Google announced *SPDY*, which became the basis for the HTTP/2 specification in 2015 [1]. HTTP/2 solved the HTTP HoL blocking and introduced an assortment of other improvements as well, including header compression, multiplexing, and server push. However, HoL blocking is still possible in HTTP/2, but at the TCP level this time, where a lost or behindhand TCP packet can delay all the subsequent ones. To respond to this issue, Google introduced the from-the-ground-up designed QUIC [2] in 2012 (referred to as gQUIC in the following), which is a fully encrypted, multiplexed, and low-latency transport protocol over UDP, targeting mainly at improving the performance of HTTPS traffic. On May 2021, IETF standardized QUIC in RFC 9000 [3] (referred to as IETF QUIC or simply QUIC in the following), and also specified in RFC 9001 [4] the way TLS 1.3 will act as a security component of QUIC. Notably, HTTP/3 [5] connections are realized only over QUIC.

In Oct. 2020, Facebook announced that "*more than 75% of [their] Internet traffic uses QUIC and HTTP/3*" [6]. At the time of writing, several QUIC implementations are already in use, some using gQUIC [7], while others IETF QUIC. Recent measurements on the top 10 million (10M) websites showed that gQUIC is still in use by approximately 8% of the websites [8], whereas HTTP/3 over IETF QUIC is currently adopted by about 25% of the websites globally [9].

[a]Correspondence: georgios.kampourakis@ec.europa.eu, gkamb@aegean.gr

The increasing popularity and adoption of QUIC by major technology companies, Internet content providers, and other stakeholders, renders it an alluring target for threat actors. However, by examining the relevant literature, thus far, no work offers a holistic survey on QUIC security. Additionally, to our knowledge, the literature lacks of a work concentrating on the security level provided by mainstream QUIC server implementations.

The work at hand aims at addressing both the aforementioned literature gaps. Specifically, as a first contribution, we provide a full-fledged survey on QUIC security by both covering works on QUIC security evaluation and on specific attacks spanning five different categories, namely cryptographic, handshake, fuzzing, transport-layer, and privacy. No less important, a meticulous security assessment is conducted through fuzz testing against the currently six commonest production-grade QUIC-enabled servers. The results of this latter effort are quite interesting, revealing on the one hand a number of zero-day vulnerabilities, which can be easily exploited by opponents to implement Denial of Service (DoS) assaults, and on the other, suggest that QUIC implementations are not yet mature enough, and more investigation is needed to identify latent vulnerabilities.

The remainder of the paper is organized as follows. The next section provides preliminaries on gQUIC and IETF QUIC protocols. Section 3 details past studies on gQUIC and QUIC security evaluation, while Section 4 elaborates on attacks targeting gQUIC or QUIC. Section 5 offers our methodology for empirically assessing six popular QUIC-enabled servers in terms of security, and presents the results. Section 6 chews over some challenges and future directions regarding QUIC security. The last section concludes.

2 Preliminaries

The differences among HTTP v1.1 (RFC 7230), HTTP/2 (RFC 7540) and HTTP/3 (at the time of writing in draft version 34) and their positioning in the network protocol stack is depicted in Figure 1. The first difference is that QUIC encompasses the security (TLS) and part of the transport (TCP) layers found in previous HTTP versions, running on top of UDP for performance reasons. Moreover, QUIC is situated on the user level, making updates easier, in contrast to earlier transport protocols built in the kernel, where modifications could create interoperability and compatibility issues. A functional difference between the two versions of QUIC is that gQUIC transports HTTP/2 frames whereas IETF QUIC is general purpose, transporting different application protocols such as HTTP/3, DNS, and SIP.

In terms of security, a major change is that QUIC provides built-in security using TLS, whereas in HTTP/2 the use of TLS is optional; also, QUIC requires at least TLS v1.3, whereas HTTP/2 supports TLS v1.2 as well. Regarding the two QUIC versions, gQUIC uses the outdated "QUIC crypto" [10] for transport security while IETF QUIC uses TLS v1.3.

According to the security analysis of QUIC presented in RFC 9000 [3], which is based on the threat model defined in [11], the security of the handshake procedure depends on the TLS v1.3 handshake, while additionally providing some protection against handshake Denial of Service (DoS) attacks. Moreover, QUIC is potentially vulnerable to Zero round Trip Time resumption (0-RTT) amplification assaults, where the attacker uses a legitimately acquired address validation token to send congestion data to a victim that currently has the attacker's old IP address, that is, the one used to acquire the token. QUIC can also be vulnerable to optimistic ACK attacks, where the opponent sends acknowledgements for packets that it has not received; this could cause a congestion controller to allow endpoints to transmit in rates that are higher than the actual network throughput. In addition, QUIC could also suffer from request forgery attacks, where the attacker controls the requests that a peer sends to a victim, exploiting potential peer's authorizations. Moreover, QUIC could also be vulnerable to DoS attacks known as *Slowloris* [12], where an attacker sends regularly the minimum amount of data necessary to hold open connections with the attacked system in an attempt to deplete its resources. Because of the low traffic generated, the latter type of attack is very stealthy compared to (volumetric) flooding ones.

Other DoS attack types manipulate the streaming operation of QUIC mounting: (a) stream fragmentation, where the attacker might choose not to send all data so that the receiving end commit resources while waiting for the missing data, or (b) stream commitment when the attacker opens a very large number of streams to deplete the other end's resources, similarly to TCP SYN flooding attacks. On the other hand, a receiving end can attack the sender by not acknowledging the received traffic, so that the sender stores the relevant data for retransmission. Currently, no mechanisms against version downgrade attacks exists in QUIC, which means that QUIC could be vulnerable to this attack unless in future version such a mechanism is defined. Moreover, privacy attacks are also possible such as traffic analysis based on the length or timing of QUIC packets; and while the so-called *PADDING* frame is a mechanism to

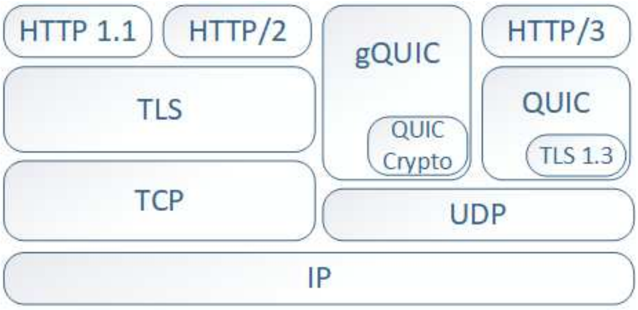

Fig. 1 Positioning of HTTP, gQUIC and QUIC

alter the real length of a packet, active research in the field has shown that defeating traffic analysis in QUIC is quite challenging.

The interest towards QUIC is demonstrated by the numerous open source implementations of the QUIC protocol. A summary of the most popular QUIC libraries in GitHub is presented in Table 1. The libraries shown in the table are open source, have a popularity of at least 100 stars in GitHub, as of May 19, 2022, and support both server and client functionality. The QUIC libraries presented here are implemented in diverse languages, making it possible for developers to find one that matches their experience and existing infrastructure. Currently, most implementations are based on *OpenSSL* as TLS library (*BoringSSL* and *QuicTLS* are forks of OpenSSL, whereas *fizz* and *picotls* can use OpenSSL as crypto engine), offering an increased level of assurance compared to less reliable solutions such as gQUIC's native crypto. Another observation from the table is that many major companies invest on QUIC by either implementing their own or supporting these libraries; this shows a real interest towards the adoption of QUIC in the near future. On the other hand, multiple implementations are more difficult to be examined for possible security issues. For instance, it is unknown if a vulnerability in *OpenSSL* will affect other TLS libraries such as *BabaSSL*. This means that each implementation should be examined separately for assessing its security level.

3 Past studies on QUIC security evaluation

In [30], the authors performed a formal analysis of the QACCE security model proposed in [31] with the use of the *ProVerif* automated security tool, stating that the security proofs of [31] were incorrect. Nevertheless, both works on the protocol analyzer level should be considered outdated since the latest version of QUIC has implemented different protocol changes, such as the IP spoofing protection.

The work in [32] compared the security and availability properties of TLS 1.3 and QUIC when layered with their underlying transport protocols. Namely, the authors examined three schemes in a formalized way: TLS 1.3 over TCP Fast Open, gQUIC over UDP, and QUIC over UDP that uses TLS 1.3 key exchange; note that only the two latter schemes are pertinent to this work. Regarding QUIC over UDP, the authors pinpoint that the protocol does not offer 0-RTT key forward secrecy and 0-RTT data anti-replay. They also refer to the work in [31] which has provided proofs that gQUIC is secure against IP spoofing based on the Authenticated Encryption with Associated Data (AEAD) security – a scheme that preserves the integrity of both the cipher text and authenticated data. Therefore, since source address tokens are validated in both full and resumption sessions, the conclusions in [31] can loosely translated in that gQUIC achieves IP-spoofing prevention.

On the other hand, they concluded that the key exchange header and payload integrity feature is not satisfied in gQUIC because the protocol's first-round key exchange messages, say, *ServerReject* as well as any invalid *ClientHello* are not fully authenticated. The authors also mention that the work in [31] detailed an assortment of attacks on gQUIC availability through key exchange packet manipulations. Moreover, they stated that gQUIC is secure regarding the secure channel header integrity property. This is because the protocol does not permit "header only packets to be sent in the secure channel phases, and the entire protocol header is taken as the associated data authenticated by the underlying encryption scheme." A last observation regarding gQUIC was that it does not meet the reset authentication property because its reset packet, namely *PublicReset*, is not authenticated. Regarding QUIC (their examination was based on a draft version of the RFC) they note that, as with gQUIC, the protocol does not provide 0-RTT key forward secrecy and 0-RTT data anti-replay and key exchange header and payload integrity. Nevertheless, QUIC fulfils the reset authentication property because of its stateless reset mechanism; the one end creates a 128-bit reset token using its static key along with a random 64-bit QUIC Connection ID (CID) as input. This token is transmitted to the other end via the secure channel.

The authors in [33] provide a formal analysis of the security of the handshake protocol of QUIC, i.e., the TLS handshake, using symbolic model checking. The analysis was performed with the help of two state-of-the-art tools for symbolic model checking: *ProVerif* and *Verifpal*. The analysis identified a protocol design flaw by revealing that some security properties are vio-

Table 1 The most common QUIC libraries shorted by popularity (GitHub stars). The parenthesis designates the number of unique values in each column.

Name (18)	**Language (7)**	**Popularity**	**TLS library (9)**	**Usage**
quic-go [13]	GO	6.7K	crypto/tls (GO based)	Algernon, Caddy, NextDNS, etc.
quiche [14]	Rust	6.1K	BoringSSL	Cloudflare, curl
nghttp2 [15]	C/C++	4K	OpenSSL	–
msquic [16]	C/C++	2.8K	OpenSSL	Windows 10/11, Windows Server 2022
quinn [17]	Rust	2.1K	OpenSSL	–
reactor-netty [18]	Java	1.9K	OpenSSL	VMWare
neqo [19]	Rust	1.4K	OpenSSL	Mozilla
mvfst [20]	C++ (folly)	1.1K	fizz	Facebook, Proxygen
xquic [21]	C	1.1K	BoringSSL/BabaSSL	Alibaba
lsquic [7]	C	999	BoringSSL	LiteSpeed, OpenLiteSpeed
aioquic [22]	Python	924	OpenSSL	hypercorn, httpx
ngtcp2 [23]	C	769	OpenSSL	–
s2n-quic [24]	Rust	594	s2n-tls	Amazon
quicly [25]	C	501	OpenSSL	H2O
picoquic [26]	C	329	Picotls	–
kwik [27]	Java	144	luminis.tls	–
quiche [28]	C++	179	OpenSSL	Google
nginx-quic [29]	C	–	BoringSSL/QuicTLS	NGINX

lated; this vulnerability allows an attacker to complete a handshake as a legitimate client with the server.

The authors in [34] present a generalized model for robustness in cryptographic channels that run on top of non-reliable transport protocols like UDP. Given that ciphertexts in such channels are not arriving in sequence, the underlying protocols need to rely on other mechanisms, typically a sliding-window technique where ciphertexts can be decrypted correctly provided that they are not received too far of each other. In this respect, typically, UDP-based protocols include a packet number in each packet to deal with packet drops and re-orderings. Note, however, that packet numbers may raise privacy risks; through traffic analysis, an eavesdropper can correlate packets and sessions. The authors focused on two protocols namely QUIC and DLTS. For QUIC, they examine the short packet format employed for sending application data; recall that QUIC packets comprise a header and a payload, and the latter is encrypted through an AEAD scheme. QUIC uses a dynamic sliding window (set dynamically on the sender side) with an anti-replay window. Based on their analysis, the authors concluded that QUIC attains robust confidentiality and integrity by receiving ciphertexts within a dynamic sliding window and with a window-based replay protection. Specifically, § 6.6 of RFC 9001 states that "In addition to counting packets sent, endpoints must count the number of received packets that fail authentication during the lifetime of a connection". In case the total number of such packets surpasses the integrity limit for the selected AEAD, "the endpoint must immediately close the connection [...]". For instance, that limit for the AES_GCM AEAD is 2^{52} invalid packets.

The authors in [35] examined the security of the QUIC record layer (in draft v30). They modeled QUIC packet and header encryption – parts of QUIC packet headers are encrypted to mitigate the privacy risk as explained in the previous paragraph – and proposed a security definition for authenticated encryption with "semi-implicit" encryption nonces. Recall that QUIC generates the nonce by combining the packet protection Initialization Vector (IV) with the packet number field, and as detailed in § 5.4 of RFC 9001, the packet number is secured through the header protection key. The authors demonstrated that QUIC employs "an instance of a generic construction parameterized by a standard AEAD-secure scheme and a PRF-secure cipher". They formalized and proven the soundness of this construction in terms of security and elaborated on restrictions of nonce confidentiality, "due to the malleability of short headers and the ability to choose the number of least significant bits included in the packet counter". They suggested amendments that explicate the proof and augment sturdiness in presence of strong adversaries.

While not introducing any attack, the work in [36] can be considered directly related to QUIC. This is because the authors provided an analysis of the TLS 1.3 handshake (authenticated key exchange) protocol from a cryptographic viewpoint; their analysis is confined to RFC 8446, not 9001, which describes the way TLS acts as a security component of QUIC. Specifically, the authors examined both the full 1-RTT handshake and the pre-shared key based resumption handshake 0-RTT modes as defined in RFC 8446. Their analysis follows the provable security paradigm, which provides rigorous, reductionist proofs that a cryptographic scheme fulfils a security objective. They concluded that "the TLS 1.3 handshake follows sound cryptographic design principles and establishes session keys with their desired security properties under standard cryptographic assumptions." And as the authors correctly pinpoint, this is also thanks to the proactively transparent standardization process the IETF TLS working group has followed.

4 Attacks targeting QUIC or gQUIC

This section only includes QUIC-related attacks. TLS-oriented attacks such as the TLS-Poison [37] and HSTS bypassing [38] ones are not specific to QUIC but apply to any TLS version, so they are considered out of scope of this work. However, attacks against the TLS 1.3 handshake are pertinent given that TLS acts as a security component of QUIC; basically, the two protocols cooperate. Specifically, QUIC uses the keys derived from the TLS handshake for providing confidentiality and integrity to the exchanged packets through its transport layer. Put simply, based on RFC 9001, QUIC exploits the *handshake* and *alert* components of the *Content* layer of TLS. The various attacks are split into five categories, namely cryptographic, handshake, fuzzing, transport-layer, and privacy. In all the subsections, the discussed works are given in a chronological order.

4.1 Cryptographic attacks

In Crypto 1998, Bleichenbacher presented a seminal attack on RSA-based PKCS#1 v1.5 encryption [39]. Essentially, he assumed the existence of an "oracle" that allowed an attacker to distinguish "valid" from "invalid" PKCS#1 v1.5 padded ciphertexts. Such an oracle may in practice be given by a TLS server, which responds with appropriate error messages, or allows otherwise to tell whether a given ciphertext has a "valid" padding or not; for instance, this can be done by observing the timing behavior of the server when processing the ciphertext. Bleichenbacher showed that such an oracle is sufficient to decrypt PKCS#1 v1.5 ciphertexts.

In 2015, the authors in [40] elaborated on encrypted key transport with RSA-PKCS#1 v1.5 key exchange method as used by TLS, and demonstrated that it was still effective. Specifically, they presented two attacks against TLS v1.3 and gQUIC, respectively. The first, requires a very quick "Bleichenbacher-oracle" to generate the TLS *CertificateVerify* message before the client releases the connection, which in turn reduces the real impact of the attack. The second, was more practical in terms of time as it can be effective in situations where forging a signature with the aid of a "Bleichenbacher-oracle" may require a great deal of time. This is because the aggressor can pre-compute the signature well in advance the client initiates a connection. They considered two scenarios either with "perfect" or "imperfect" oracles.

At the time this attack was proposed, having an "imperfect" oracle was considered to be impractical for a real-world scenario. Moreover, having a "perfect" oracle weakness to a server, it should be considered highly unrealistic. Given that an Ubuntu 14.04 with 2.2 GHz dual core CPU was used to execute the queries, modern hardware is sure to perform much faster. On the other hand, the browser's connection timeout timer may have been reduced due to the higher network speeds. For example, Firefox has now a default connection timeout (*network.http.connection-timeout*) of 90 sec (at that time was 600 sec), while Chrome has still the same default value of 30 sec. Additionally, as NIST suggests, starting from 2020, the RSA key length should be at least 2048 bits [41], i.e., increasing by far the complexity and the resources at the attacker's side; in the context of [40], a 2048 bits key needed 2,120 queries towards the server, requiring 66 sec in total. RFC 8446, appendix § E.8 titled "Attacks on Static RSA" highlights this issue: TLS 1.3 does not use RSA key transport, therefore it is only indirectly susceptible to Bleichenbacher-type attacks. That is, "if TLS 1.3 servers also support static RSA in the context of previous versions of TLS, then it may be possible to impersonate the server for TLS 1.3 connections". This situation can be avoided by simply disabling support for static RSA across all versions of TLS.

The work in [42] introduced an attack dubbed DROWN (Decrypting RSA using Obsolete and Weakened eNcryption). Precisely, DROWN was presented as a novel attack that allowed a perpetrator to break a passively collected RSA key for any TLS server if this RSA key was also used for SSLv2; therefore DROWN is characterized as cross-protocol; the assault uses a server supporting SSLv2 as an oracle to decrypt TLS sessions. In its generic version, which comprises a more powerful variation of the Bleichenbacher RSA padding-oracle assault, the attacker has to passively observe around 1K TLS sessions with RSA key exchange, make 40K TLS v2 connections, and perform 2^{50} symmetric encryption operations. Moreover, they pinpointed that gQUIC is vulnerable to a Man-in-the-middle (MitM) variant of the so-called *DROWN* attack, enabling the opponent to impersonate a server for as long as they wish. According to the authors, this requires 2^{17} SSLv2 connections and 2^{58} offline work. As a basic requirement, the attack can be performed against servers which expose their RSA public keys through SSLv2. The assault capitalizes on the "server config" message received by a gQUIC client when attempting a connection to the server. This message includes the server's Diffie-Hellman public value, it has an expiration date, and it is signed by the server's private key. In this respect, for launching a MitM attack against the client, the attacker needs to produce a (forged) valid "config message"; for counterfeiting the signature they need to find a valid, PKCS conformant SSLv2 ciphertext. The authors pinpoint that QUIC discovery can be done over plain HTTP, meaning that the server can be impersonated by the attacker and falsely indicate support of QUIC. Therefore, the next time the client connects to the server, it will attempt to connect using QUIC, allowing the attacker to present the forged "server config" message and execute the assault. Naturally, this attack does not affect IETF-QUIC as its cryptographic handshake is mandatorily based on TLS 1.3 rather than gQUIC crypto.

The authors in [43] conducted a formal analysis of TLS v1.3 revision 10 through *Tamarin Prover*, a security protocol verification tool. Their analysis, considering both unilateral and mutual authentication, demonstrated that revision 10 fulfils the objectives of authenticated key exchange. Further, by extending their model, they proposed a client impersonation attack based on the delayed authentication process; presumably this is the post-handshake authentication feature defined in § 4.6.2 of RFC 8446, and it is permitted only if the client has sent the *post_handshake_auth* extension. The identified attack requires certificate-based client authentication in session resumption using a PSK, as defined in § 2.2 of RFC 8446 – a PSK can be created in a previous connection and subsequently used to establish a new connection. Specifically, the attacker must be in a MitM position with the client and the server, and therefore is able to establish valid PSKs with both ends. Assuming a session resumption process the requires post-handshake authentication, the authors noticed that the attacker is able to produce the correct *CertificateVerify* message because the Finished messages are not included in the session hash, i.e., the Transcript-Hash as defined in § 4.4.1 of RFC 8446. After that, the attacker possesses the session keys and is able to impersonate the client to the server. On the other hand, § 4.4 of RFC 8446 states that the *Handshake Context* for post-handshake authentication includes the server's Finished message. That is, with reference to § 4.4.1 of the RFC, the signature provided in the *CertificateVerify* message is produced over the value *Transcript-Hash(Handshake Context, Certificate)*. Therefore, as the authors acknowledged, the identified attack is not considered feasible starting from revision 11 of the RFC.

The authors in [44] elaborated on the nonce reuse/misuse resistance authentication encryption schemes for modern TLS and QUIC based web servers. Specifically, the problem of nonce-based authentication encryption is the repetition of nonce in two different messages; it is well-known that any AEAD relies on no reuse of the same pair of key and nonce. The authors argue that the nonce problem is avoided by using a nonce reuse/misuse resistance authentication scheme like the AES_GCM_SIV. The latter, defined in RFC 8452, is a mode of operation for AES, which comprises a nonce misuse-resistant AEAD. Based on their analysis, the AES_GCM_SIV is a superior scheme for achieving better security bounds performing large messages in TLS cipher suites and QUIC-based web servers, which also fulfils the NIST security bound of 2^{-32}. With reference to the relevant RFCs, the § 5.3 of RFC 8446 states that "Each AEAD algorithm will specify a range of possible lengths for the per-record nonce, from N_MIN bytes to N_MAX bytes of input [as given in RFC 5116]. The length of the TLS per-record nonce (*iv_length*) is set to the larger of 8 bytes and N_MIN for the AEAD algorithm. An AEAD algorithm where N_MAX is less than 8 bytes must not be used with TLS." Put simply, TLS 1.3 uses the same key for multiple records, but constructs the per-record nonce by XORing the accordingly padded 64-bit sequence number (is maintained separately for reading and writing records) with either the static *client_write_iv* or *server_write_iv*; the latter is a 96-bit value obtained through key generation. On the other hand, RFC 9000 states in § 12.3 that "The packet number is an integer in the range 0 to 2^{62}-1. This number is used in determining the cryptographic nonce for packet protection. [...] A QUIC endpoint must not reuse a packet number within the same packet number space in one connection." If the packet number for sending reaches 2^{62}-1, the sender must silently terminate the connection.

Recall that based on § 2.2 of RFC 8446, TLS 1.3 supports the use of a PSK that was either established in the context of a previous handshake ("session resumption or resuming with a PSK") or distributed out-of-band. Note that the way an out-of-band PSK is generated and distributed is basically considered external to TLS; nevertheless, the Internet-Draft titled "Importing External PSKs for TLS" [45] provides a PSK importer interface to mitigate against possible security issues. The authors in [46] pinpointed that the problem with this PSK-based, and therefore implicit, authentication mode leaves room to a "(misuse) vulnerability, assuming that a node can act as both a client and a server: the sender of an authentic message could also be, under an attack scenario, the receiver itself." The attack, dubbed "Selfie", is a type of reflection attack where the opponent acts as MitM, say, between Alice and Bob. Alice initiates session S1 with Bob by sending him the *ClientHello* message with the *pre_shared_key* extension. The attacker captures the message and reflects it back to Alice in another session S2, where Alice acts as the server and Bob as the client. In S2, Alice replies to Bob with *ServerHello* and *server Finished* messages. The attacker again captures both messages, and in the context of S1, reflects them back to Alice. These messages will be authenticated by Alice, who then is convinced she is talking to Bob. As a result, in S1, Alice will reply with a *client Finished* message, which is again echoed back to Alice in the context of S2. This procedure will result in Alice opening a "Selfie session with herself", meaning that any piece of data she sends in S1 will be echoed back to her through S2. Guidelines for addressing the Selfie attack are given in Appendix B of Internet-Draft [45].

4.2 Handshake attacks

The authors in [31] contributed the first study that focused on gQUIC security. They managed to identify three different attacks in which the gQUIC protocol was vulnerable and proposed a security model, namely *QACCE*. The first one is a replay attack where an attacker with a MitM position could replay either some key establishment parameters to the client or the packet that contained the source address token (stk) of the client to the server. In both cases, the attacker could mount a Denial of Service (DoS), causing the protocol handshake to fail. As stated in the analysis of [47], such attacks are still feasible during a 0-RTT connection; the authors compared this attack with the legacy TCP SYN flood one. The second attack they examined is based on packet manipulation; as discussed in [47] further down, unprotected parameters, such as the CID or the *stk* token, could be possibly manipulated by an attacker with a MitM position to force a user into downgrading to a TLS connection over TCP. The result is the same as with the replay attack above, that is, for as long as the attack is active the client will be unable to connect with the server due to handshake failure. At this point, the authors correlated this attack with an SSL downgrade one, say, from SSL v3 to SSL v2. The third attack tested by the authors was a crypto stream offset assault, where they injected a four character string, that is, "REJ\0" into the handshake message. This addition was enough to make the handshake fail and prevent the establishment of a QUIC connection between the client and the server either by denying access to the requested application or forcing the client to fall back to a TCP/TLS connection. Additionally, this attack was correlated to a TCP ACK flooding attack, in which an attacker sends multiple acknowledged (ACK) responses to the server to confuse it and drop the connection.

The work in [47] provided a comparable study between TLS 1.2 and gQUIC from a security perspective. Specifically for gQUIC, the authors described (although not practically evaluated) three different attacks in the context of HTTP/2 services. All the attacks assume that the opponent has obtained a MitM position between the legitimate client and server and actively listens and interacts with the communication between the two ends. In the first attack, the adversary waits for the client to initiate the connection establishment process, that is, through a *ClientHello* message, and in a race condition with the server, they respond to the client with a reject message, containing a spoofed source address token, that is, an authenticated-encryption block of the client's IP address and a timestamp. Upon reception, and given that the source address token is invalid, the server sends a reject message to the client, who needs to repeat the connection establishment process from the onset. The second attack considers the manipulation of unprotected parameters, including the CID or the source address token. This drives both the client and the server to possess different initial keys, eventually wrecking the connection establishment. The last attack capitalizes on the fact that all handshake messages in gQUIC comprise a logical byte-stream. If the adversary is able to inject random data to interrupt the byte-stream, they can render the byte-stream invalid, breaking the connection establishment. It is obvious that all three attacks have no real impact on the server and only induce a DoS effect on the client. It is also important to add that according to § 5 of RFC 9000, *"0-RTT provides no protection against replay attacks"*, while § 9.2 of RFC 9001 states that *"QUIC is not vulnerable to replay attack, except via the application protocol information it might carry"* and *"Disabling 0-RTT entirely is the most effective defense against replay attack"*. Moreover, on-path active attacks and handshake DoS are discussed in § 21.1.3.1 and § 21.2 of RFC 9000.

In work [48], the focus is on the 0-RTT mode used in gQUIC and TLS 1.3. Since this mode allows the client and server to exchange data before switching to fully secure keys, the authors studied the way replay attacks affect the security of these protocols. Their results show that, at least at that time, replay attacks were accepted as inevitable by the protocol creators. This is further confirmed in IETF QUIC in RFC 9000.

The authors in [49] compared gQUIC with other common protocols, such as TCP and UDP, in an attempt to pinpoint security issues in gQUIC. Their analysis led to the identification of two 0-RTT oriented attacks, namely *QUIC RST* and *Version Forgery*. In QUIC RST, the attacker sends a reset packet to the client, making it believe that the server refused the connection; this could lead the client to give up the connection. In Version Forgery, the attacker pretends to be the server and sends to the client a version negotiation packet with a version that is not supported by the client; again the client is deceived to drop the connection. The authors argue that both these attacks are more easily mounted in a LAN environment due to Address Resolution Protocol (ARP) spoofing.

The authors of [50] examined downgrade attacks in TLS 1.3 in 2020. More specifically, they studied the Signaling Cipher Suite Value (SCSV) downgrade protection mechanism offered in TLS 1.3 and its support in 10 major Web browsers, including Firefox, Chrome, and Edge, in five operating systems. The SCSV mechanism is a pseudo cipher suite that is included in the *ClientHello* message of TLS 1.3 and mentions the latest supported TLS version of the client; for this protection to be present, both the client and the server must support it. During their analysis, they discovered that all browsers in iOS v12.2, Safari and QQ in macOS v10.14.4, and Internet Explorer, Edge, and QQ in MS Windows 10 v1809, were vulnerable to a version downgrade attack, due to not supporting the protection mechanism of SCSV. In MS Windows, the TLS version could be downgraded to 1.0, whereas in iOS and macOS to 1.2. Indeed, such an attack is practically feasible and could be quite easily mounted due to the lack of SCSV protection.

Gagliardi and Levillain [51] studied the QUIC transport and cryptographic layers in the related Internet drafts from version 18 to version 23. They focused on the connection establishment in existing implementations, testing them for security issues with the aid of the *Scapy* Python library. They tested four different attacks in ten live demo servers, each one with a diverse QUIC implementation. The first attack was *version negotiation*, similar to the one already mentioned in [49], where the attacker sends a different, yet-to-be-defined protocol version. Only the *ngtcp2* implementation was affected by this attack, leading to a timeout. The second attack checked the behavior of implementations against the *client initial packet length*. To defend against amplification attacks, QUIC specifies that the initial packet sent should have a length of at least 1,200 bytes and the server's response length should be at most 3 times the client's packet length. The authors sent short initial packets (300 bytes long) to all the implementations and even though two of them replied, they did not send more than 3 times the length of the initial packet, providing a basic level of protection against amplification attacks. In the third attack, namely *missing parameters*, different parameters described as mandatory in both QUIC and TLS 1.3, such as the maximum size of each packet, were sent to all implementations. The authors mentioned that some servers accepted packets with missing mandatory parameters without specifying which ones had this behavior. Finally, they mounted *frame mangling* attacks by sending forbidden frames to the demo servers. In QUIC, four different types of packets could be used inside an initial packet, namely, *crypto*, *ack*, *connection close*, and *padding* frames. However, the authors identified that in four different tested cases, at least four different packets could be encapsulated inside an initial QUIC packet, causing a successful handshake instead of an error. First off, they used a Ping packet, that got a response from *aioquic*, ats, and *ngtcp2*. Second, they sent a split crypto packet, dividing the TLS *ClientHello* into two crypto frames, with only pandora and quant responding correctly, that is, rejecting the handshake. The third and fourth ones were based on sending overlapping crypto packets, one with consistent and one with inconsistent contents respectively. The correct behavior, that is, producing an error, was observed only from ats, pandora and quant, in both cases.

Nawrocki et al. [52] examined the QUIC handshake protocol against two type of attacks, namely, state-overflow and reflective amplification, to eventually prove that QUIC is vulnerable to resource exhaustion attacks. This choice stems from the latest QUIC RFC [3], which, to some extent, mitigates amplification attacks (see Section 2). State-overflow attacks, similarly to TCP SYN flood, are based on the fact that QUIC keeps a state from each connected client. The authors stated that another defensive mechanism has been added to the QUIC RFC to retry authentication if the IP address cannot be verified. While the latter protection method can assist into mitigating attacks, such as state-overflow, the same defensive mechanism is considered to be against the QUIC design goals since it introduces an additional overhead and delays the authentication procedure. To evaluate their observations and to identify DoS attacks that may exist "in the wild", the authors collected and analyzed QUIC traffic, that is, 92M QUIC packets, from UCSD Network Telescope [53]. Then, they selected sessions with: a) more than 25 packets, b) duration longer than 60 sec, and c) a maximum packet rate of higher than 0.5 pps, calculated over all 1-min slots of the respective event. Their analysis revealed 2905 attacks, corresponding to 11% of all response sessions. According to the authors, 98% of all of these attacks target well-known QUIC servers. Additionally, they made a twofold observation on these attacks: a) QUIC floods are shorter but on average as severe as common TCP/ICMP floods and b) QUIC floods are part of multi-vector attacks and highly correlated with TCP/ICMP floods. The latter correlation means that QUIC floods were used in conjunction with TCP/ICMP floods against a target. To cross-validate their results, the authors set up an NGINX webserver, on a 128-core computer with 512 GB RAM and recorded 500K packets using the QUIC client *quiche* [14]. To simulate attacks, the authors sent only client Initial messages with random packet rates to new server instances. The authors found that servers with 4 or 128 workers which did not support the *RETRY* defensive mechanism presented slower responsiveness by 32% or 64% after 100 or 10,000 pps, respectively. On the other hand, servers that did implement the *RETRY* protection mechanism, with only four workers could withstand even a load of 100,000 pps. This clearly indicates the importance of the *RETRY* scheme; to further validate this observation, the same experiment has been carried out in subsection 5.2.7 against six different production-grade QUIC servers.

4.3 Fuzzing attacks

McMillan et al. [54] presented a formal specification of the QUIC's wired protocol that generates automated randomized testers for QUIC. The authors evaluated their method on a quartet of diverse QUIC implementations. For each of them, they performed test runs on a HTTP server based on a QUIC library. Their results revealed various issues in the server implementations, which according to the authors, point to issues in the draft standard. From a total of 27 errors, four cases were deemed to be exploitable; these were protocol or progress violations that revealed possible vulnerabilities in the implementations or the (at that time draft) standard itself. Specifically, one of these was a potential DoS attack by an off-path attacker. Another one was an information leak comparable to the OpenSSL well-known *heartbleed* vulnerability; it occurs when a server instance sends incorrect number of bytes in a retransmission of a stream frame, violating the specification and resulting in a stream frame that is longer than intended, thus, causing arbitrary server memory data to leak into the network. Yet another data leak case was an *APPLICATION_CLOSE* frame transmitted in response to the unexpected initial packet, sending application data in the clear. The last issue was a case where a server instance allocated a number of records in memory that was proportional to the gap in packet sequence numbers, allowing an attacker to eventually exhaust the server.

G.S. Reen et al. [55] presented *DPIFuzz*, a stateless fuzzing framework, which can find exploit strategies for Deep Packet Inspection (DPI) in QUIC implementations with the assist of "Mutations", such as using duplicate packet numbers and exploiting the diverging handling of overlapping stream offsets. The authors evaluated their framework against five open-source QUIC implementations, namely *QUICLY* [25], *QUANT* [56], *NEQO* [19], *MVFST* [20] and *QUICHE* [14], identifying the following vulnerabilities. The implementation of QUANT tries to access the state of a stream, i.e., either open or closed, by calling the *q_is_stream_closed()* function after the memory allocated to the stream has already been freed. The NEQO server calls the *close()* function on a connection with state CLOSED.

In QUICLY, the server runs into a segmentation fault due to a reference on a null pointer; more specifically, the *QUICLY_get_ingress_max_streams()* function tries to access the value of a NULL pointer, causing the server to crash. Finally, the MVFST server runs into a segmentation fault due to a reference on a null pointer; this is caused when a stream frame with a non-zero Offset field has the OFF bit set to 0, indicating that the Offset field is absent. In all the above cases, the issues were reported and fixed. In general, while a stateful, structure-aware fuzzer could identify more bugs in QUIC servers, it could also be considered to be too implementation-specific; for example, Table 1 shows that at least five different languages are used across 18 QUIC implementations.

J. Zhang et al. [57] presented a modeling and verification method for cryptographic protocols. The authors performed analysis based on the symbolic protocol model, to verify the security of the QUIC handshake protocol. Their results reveal that an attacker can impersonate the client by forging two plaintext messages, i.e., *CEPub* and *CHLO*, and sending them towards the server to complete the establishment of the initial session and final session key with the server. Additionally, the authors proposed a revised QUIC handshake protocol which uses the client's private key to sign *CEPub* and *CHLO*, and thus prevent attackers from forging the aforementioned messages.

K.Thimmaraju et al. [58] analyzed the Connection ID (CID) mechanism of the IETF QUIC draft v.30. According to the authors, an attacker can exploit servers, which do not permit the use of the same destination CID across new connections, to get the number of server instances behind a load balancer. The authors dubbed this situation "enumeration attack". This information can be used to estimate the load necessary to mount a Distributed Denial of Service (DDoS) assault against the server. Their analysis on 15 QUIC implementations concluded that 25% of them are vulnerable to this attack, namely, Apache Traffic Server (ATS), Chromium, LiteSpeed, and ngtcp2. To solve that, the authors proposed an enumeration algorithm that can be exploited to prevent an attacker to perform this enumeration attack. We argue that the enumeration attack discussed in the above work, is practically infeasible on the *Chromium* and *ngtcp2* QUIC implementations due to the fact that both of them are not load balancers.

4.4 Transport-layer attacks

Gbur et al. [59] showed how stateful firewalls handling QUIC are prone to *UDP hole punching*, i.e., a well-known firewall bypass technique; typically, this technique is used to establish UDP connections with systems behind Network Address Translation (NAT). The authors used the QUIC implementation *quiche* [14] on two Virtual Machines (VMs), a client and a server running Ubuntu OS, and a transport layer stateful firewall, that is, the Linux *conntrack* module. This experiment assumes that the aggressor either has compromised the web server via a remote code execution vulnerability on the application level or has access to another computer in the same internal network as the web server. Considering that UDP is connectionless by nature, it mimics the TCP 3-way handshake, whereas the firewall tries to guess which packets belong to the same connection based on the following 5-tuple values: source IP address, source port, destination IP address, destination port, and transport protocol. The firewall treats every UDP packet as a new connection and adds a new entry in the connection table with the 5-tuple and the status *UNREPLIED*. If the firewall receives a packet with a 5-tuple listed in the table, it immediately assumes that it belongs to the same connection and updates the entry by removing the *UNREPLIED* status. In practice, every packet stemming from the server side with this 5-tuple "punches a hole" in the firewall that can be used for all server side packets having this same 5-tuple. Subsequently, this hole can be used by the attacker to spawn a reverse shell using the compromised machine mentioned earlier.

4.5 Privacy attacks

Although the work from Van et al. [60] focuses on traffic classification, the described techniques could be used for attacking the privacy of users as well. Their proposal is a static-based method which relies on the Convolutional Neural Network (CNN) to detect various QUIC-based Google services, such as Google Hangout Chat, Google Hangout Voice Call, YouTube, File transfer, and Google play music. Specifically, their method uses a Random Forest model trained with NetFlow-based features to detect Google Hangout services due to the unique network flow of these services. It also uses the CNN model trained with packet-based features to classify file transfer, YouTube, and Google play music network traffic. The authors built a proof-of-concept implementation of their approach on an Ubuntu workstation using *quic-go* [13] to evaluate its performance. Precisely, they captured around 150GB of real network traffic including over 20K flows with five kinds of QUIC-based services. The experimental results showed that the proposed method can detect the five aforementioned QUIC-based Google services with high accuracy (approximately 99%).

The authors of [61] tested the privacy of the full handshake and session resumption mechanisms of TLS 1.3. Their results showed TLS 1.3 does provide a level of privacy, when considered in isolation. According to the authors, in full handshakes, TLS 1.3 provides a perception of server unlinkability. On the other hand, session resumption, a fundamental feature of TLS, introduces a method of linking sessions between two objects. In addition, the authors demonstrated that privacy in TLS 1.3 is reduced when resumption is considered, due to the use of session tickets. In that sense, TLS 1.3 offers an rather optimal degree of privacy.

Govil et al. [62] presented *MIMIQ*, a privacy enhancing tool which leverages on QUIC to mitigate traffic-analysis attacks and protect user identity. To evaluate the proposed tool, the authors created a proof of concept MIMIQ implementation. More specifically, they used an off-the-shelf Dynamic Host Configuration Protocol (DHCP) server and configured short DHCP lease duration, to make old IP address leases expire in time for new requests. Additionally, they modified a QUIC client to act as a MIMIQ agent. This latter sends DHCP requests to the DHCP server, and triggers connection migration by changing its interface IP using the *ifconfig* system administration utility. Furthermore, *Mininet* [63] was used to simulate two networks, i.e., a network with only one QUIC client running and a multi-client network with four QUIC clients, each running on a different host. Each network had both a DHCP and a QUIC server. Finally, the authors evaluated MIMIQ against state-of-the-art website fingerprinting attacks, and found that setting migration every 25 to 100 packets can reduce attack accuracy to less than 10%.

The authors in [64] examined website fingerprinting attacks on QUIC, gQUIC, and HTTPS/2 from a traffic analysis perspective. The purpose of this type of passive attack is to deduce the website a user is visiting, which can be translated to a multi-class classification task. The tests employed by the authors concluded that gQUIC and QUIC are more at risk to these attacks than HTTPS/2 in the early traffic scenario, but are similar in the normal full traffic. Specifically, the authors collected network traffic using a controlled environment which comprised three web servers running Ubuntu 18.04. Additionally, the authors selected the official landing pages of the top 100 schools, which are split among the three hosting servers. Docker was used in each server to isolate resources from these websites. To compare the protocols against fingerprinting attacks, the authors utilized five machine learning models, i.e., Random Forest, Extra Trees, k-nearest neighbors, Naive Bayes, and Support Vector Machine. The results were obtained using 10-fold cross-validation. According to the authors, with only 40 packets, the attack accuracy reaches 95.4% for gQUIC, 95.5% for QUIC, whereas only 60.7% for HTTPS/2.

Ludovic et al. [65] examined whether network-layer padding, a feature included natively in QUIC, can be used to defend against website fingerprinting attacks. Specifically, two datasets were used during the experiments, namely a mixed and a QUIC dataset. The authors observed that in the mixed dataset, which has only 4% QUIC traffic, the classifier favors TLS-specific features. For that reason, the authors created the QUIC-dominated dataset, which contains 70% of QUIC traffic. The authors implemented an attack using the Random Forest model and their results showed that: a) Network-layer defenses allow website identification with an F1 score of more than 92%, b) web development practices, such as hosting resources on multiple servers, reduce the effectiveness of any defense, and c) the IP addresses can help the attacker associate QUIC packets with a client; defences include MIMIQ [62] or IP hiding solutions, such as Near-Path NAT [66] and MASQUE [67].

5 Hands-on evaluation

The current section offers a hands-on security evaluation performed against the six most popular QUIC and HTTP/3 enabled servers based on April 2022 figures provided by W3Techs [68]. That is, from the list in [68], we only consider servers that support both QUIC and HTTP/3; in this respect, Apache web server has been left out as it currently does not support QUIC. Particularly, we examined the QUIC implementation of these servers, while they were specifically configured to support HTTP/3. All the servers were installed in the Azure VM cloud to simulate a real-life environment. Therefore, every VM had a specific subdomain bound to the Azure VM public IP address.

5.1 Testbed

Table 2 details the Operating System (OS), VM specifications, and version of each tested server. Each server stored a single, simple HTML webpage. For the sake of repeatability, the configuration details per server are succinctly given in the subsequent subsections.

5.1.1 OpenLiteSpeed

The *OpenLiteSpeed* latest stable version v1.7.15 has been installed in an Azure VM with Ubuntu 18.04 server 1xCPU, 1 GB RAM and standard SSD. First, we acquired a domain name and assigned it to the DNS zone of Azure. Next, we setup a *VirtualHost* and a Listener to enable a website, and uploaded a simple HTML webpage created with the *SingleFile* Chrome extension. This was done to avoid having issues with the HTTP/3 in case of broken HTTP requests. Also, a wildcard TLS certificate from `www.shieldsigned.com` website was configured in the server. The OpenLiteSpeed admin panel and the Listener were configured to use HTTP/3 and the TLS certificate, and server's port 443 has been left open for UDP traffic. The *HTTP/2 and SPDY indicator* extension was installed in Chrome

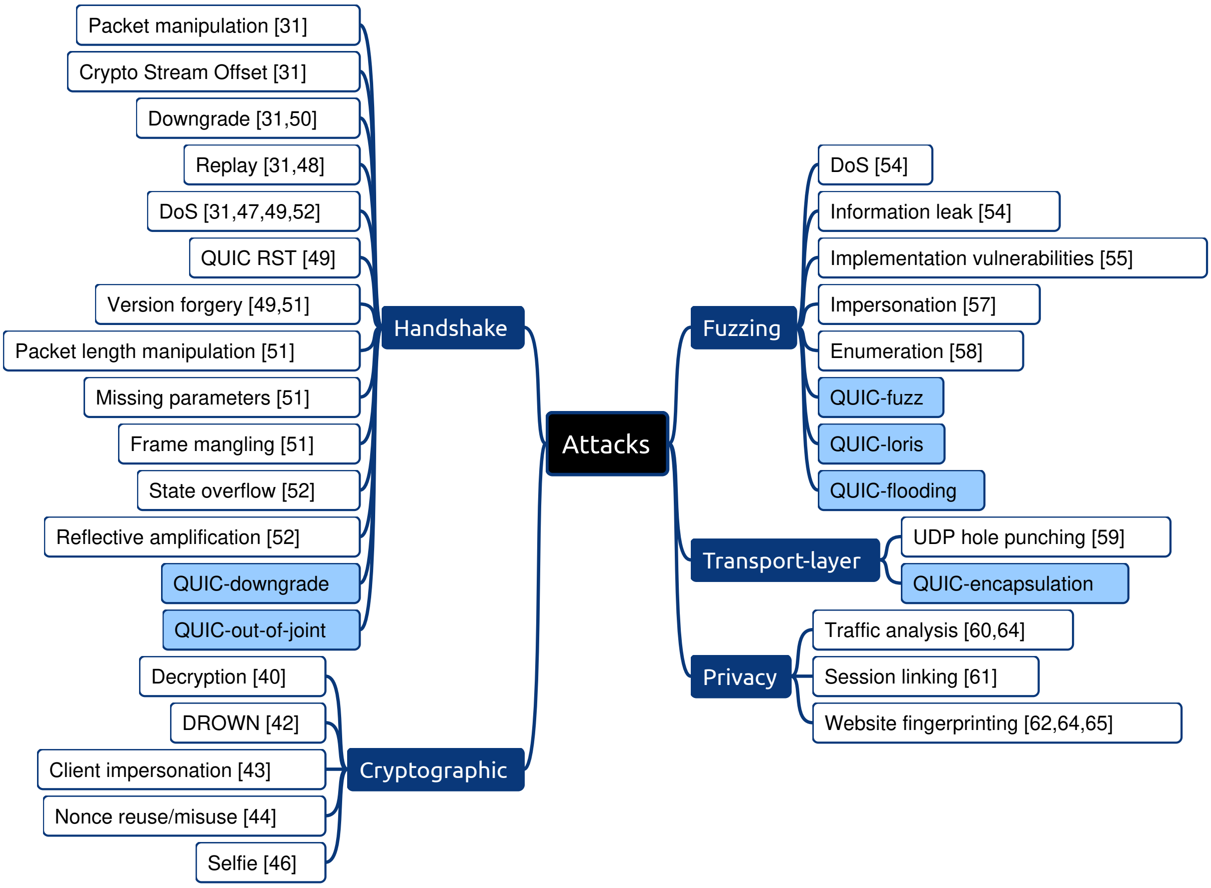

Fig. 2 A taxonomy of attacks on QUIC (white: attacks in the literature, light blue: attacks we performed during our evaluation)

Table 2 Testbed setup

Name	OS	Spec	Version
OpenLiteSpeed	Ubuntu 18.04	1xCPU/1GB RAM	1.7.15
Caddy	Ubuntu 18.04	1xCPU/1GB RAM	2.4.6
NGINX	Ubuntu 18.04	2xCPU/4GB RAM	1.21.7
H2O	Ubuntu 18.04	1xCPU/1GB RAM	2.3.0-DEV
IIS	Windows Server 2022	2xCPU/4GB RAM	10
Cloudflare	Ubuntu 18.04	2xCPU/4GB RAM	1.16.1

to obtain a visual indication when visiting an HTTP/3 website. To validate that our configuration was sound, we visited the `www.ssllabs.com` and examined our certificate, which received an "A" rating. Last, through `www.http3check.net`, we verified that our website was indeed able to handle QUIC and HTTP/3 requests.

5.1.2 Caddy

The Caddy v2.4.6 server supports an experimental HTTP/3 server edition, and it was installed on an Ubuntu 18.04, with 1xCPU and 1 GB RAM Azure VM. Following the same methodology as with *OpenLiteSpeed*, we registered and assigned a domain name to the server. Caddy uses a TLS certificate out-of-the-box, which is rated as "A" by `www.ssllabs.com`. Through `www.http3check.net` the usage of HTTP/3 and QUIC was verified. Note that for QUIC, we perceived a delay, that is, the server needs 2 to 3 min after it is booted to be able to communicate with QUIC; this is due to a known issue with the current version reported in [69].

5.1.3 NGINX

As stated in the official installation guide[1], *NGINX* experimental version v1.21.7[2] supports QUIC with HTTP/3. It should be noted that the latter guide suggests using either the *BoringSSL* or the *QuicTLS* libraries. Since the former library could not compile with 1xCPU and 1 GB RAM, we exploited a more powerful VM, i.e., 2xCPU, 4 GB RAM and standard SSD in Ubuntu 18.04. After server compilation, it was required to create a service for binding *NGINX* to it. By following the *NGINX* official guide[3], we created the service, placing the correct paths, i.e., those used during *NGINX* installation.

5.1.4 H2O

H2O server vh2o/2.3.0-DEV@a429117ba was the latest edition as of Feb 22, 2022. We followed the official installation process[4] as well as the quick start guide[5]. The server was installed on an Ubuntu 18.04 Azure VM, with 1xCPU and 1 GB RAM. First, we installed the TLS certificate and assigned a domain name, similarly to the other servers. To configure the server to communicate with HTTP/3 and QUIC, we utilized the *h2o.conf* example file [6] taken from the server's official repository; this however made the server to use HTTP/2 over TCP. To fix this issue, we configured the minimum version of the TLS certificate to TLS 1.3; this can be done through the option *minimum-version: TLSv1.3* under the ssl category. Moreover, the header set field was wrong, forcing browsers to communicate with HTTP/2 instead. To correct this bug, the draft-25 version from the h3 header was removed and the *header.set: "Alt-Svc: h3-25=\ ":443\ ""* option was changed to *header.set: "Alt-Svc: h3=\ ":443\ ""*. Lastly, the default port of the server was changed to 443.

5.1.5 Internet Information Services (IIS) 10

IIS 10 can be operated from a MS Windows Server 2022 Azure VM. Due to the Globally Unique Identifier (GUID) this server requires, we created a 2xCPU with 4 GB RAM and standard SSD Azure VM and installed IIS on it. The Dynamic Update Client (DUC) aided us in using a subdomain as the main domain name of the server. However, some issues arose with the TLS certificate; the server did not accept any TLS (.cer, .pfx, .p7b, .crt) certificate. In most cases, we received an error, while in the case of .cer, after its successful installation, the server deletes the certificate. To overcome this issue, the *win-acme*[7] tool was used. The latter can create a *Let's encrypt* TLS certificate and install it smoothly to the IIS. Since IIS 10 operates with HTTP/2 out-of-the-box, it was necessary to enable QUIC, which in turn facilitates HTTP/3. To this end, the following registry commands were executed from an admin terminal:

- `reg add "HKEY_LOCAL_MACHINE\SYSTEM\Current ControlSet\services\HTTP\Parameters" /v EnableHttp3 /t REG_DWORD /d 1 /f`
- `reg add"HKEY_LOCAL_MACHINE\SYSTEM\Current ControlSet\services\HTTP\Parameters" /v EnableAltSvc /t REG_DWORD /d 1 /f`

Moreover, it was required to disable the "Disable Legacy TLS" checkbox from the site binding of HTTPS and advertise the QUIC usage by adding a custom HTTP response header, namely "alt-svc" with values "h3=":443"; ma=86400; persist=1". After restarting the server and unblocking the port 443 for UDP connections, both the client's browser and `www.http3check.net` verified that the ISS was using HTTP/3 with QUIC.

5.1.6 Cloudflare

Cloudflare offers an open-source QUIC library, namely *quiche*[8]. This library uses *Rust* v1.53 or later, and in the same repository, Cloudflare provides a guide to install NGINX with the quiche implementation[9]. As mentioned in subsection 5.1.3, NGINX uses the *BoringSSL* TLS library. Therefore, to be able to compile BoringSSL, quiche, and NGINX, all the necessary dependencies, including *build-essential* and *rustup* had to be installed as well. For the same reason, we used an Ubuntu 18.04 Azure VM, with 2xCPU and 4 GB RAM. After compilation, as mentioned in subsection 5.1.3, the creation of a NGINX service was required. However, NGINX v1.16.1 comes with a bug[10], according to which NGINX is unable to locate the process ID (*.pid*) file. To solve this issue, NGINX developers suggest linking the service file with the "/logs/nginx.pid" file, which is the default location of the PID file. Finally, we followed the suggested configuration of the NGINX server, as given

[1] https://quic.nginx.org/readme.html
[2] https://hg.nginx.org/nginx-quic
[3] https://www.nginx.com/resources/wiki/start/topics/examples/systemd/
[4] https://h2o.examp1e.net/install.html
[5] https://h2o.examp1e.net/configure/quick_start.html
[6] https://github.com/h2o/h2o/blob/master/examples/h2o/h2o.conf
[7] https://github.com/win-acme/win-acme
[8] https://github.com/cloudflare/quiche
[9] https://github.com/cloudflare/quiche/tree/master/nginx
[10] https://bugs.launchpad.net/ubuntu/+source/nginx/+bug/1581864

in the *quiche* guide, forcing the usage of TLS 1.3 and removing the "http2" option.

5.2 Empirical security assessment

This subsection reports on the conducted security assessment against the six servers. First, we detail the methodology followed to examine each server, while our findings follow. For easy reference, Table 3 recapitulates the identified issues per server.

5.2.1 Methodology

After examining the related work on QUIC in Section 4, we realized that some research directions have not yet been considered: (a) no work has examined the effect of sending malformed QUIC packets to a QUIC server; this should be done based on RFC 9000 and not a QUIC draft implementation, and (b) no work heretofore targeted IETF QUIC through malformed packets at the UDP layer. To this end, we utilized two different fuzz testing tools (fuzzers), one for each of the aforementioned cases, and exploited them against each of the six servers of Table 2. Next, a manual examination phase took place based on different misconfigurations observed during the first phase.

The first fuzzer, namely *Mutiny-fuzzer*[11], is capable of fuzzing custom packets, including QUIC ones. After the user assigns a pcap type of file to the fuzzer and configures its communication port, the fuzzer creates a specific file for that pcap to use as a test case. Munity-fuzzer uses *Radamsa*[12] to create seeds. The seeds are usually added at the end of each packet and the testing rate was set to one seed per sec. In most cases, the QUIC handshake was ended after 2-3 packets, unable to complete the TLS handshake, since the transmitted QUIC packet was malformed. However, in some fewer cases, the fuzzer managed to perform a full handshake and receive some data, validating the observations of the work in [51] regarding the frame mangling attack.

Fuzzotron[13] was the second fuzzer employed in the context of our tests. It can be used to create custom TCP or UDP packets and exploit them against a server implementation. To this direction, it uses a grammar-based mutator, namely *Blab*[14] or *Radamsa*. Since Radamsa was utilized along with Munity-fuzzer, for this second fuzzer, we used Blab. The testing rate of the fuzzer was 100 seeds per sec.

Figure 3 provides an overview of the fuzzing procedure. The fuzzing VM was an Ubuntu 18.04 with a quad core CPU and 16 GB RAM. Every server was tested through each fuzzer for 12 hours, therefore the total duration of the fuzzing process was 24 hours per examined server. In total, approximately 43,200 and 43,200,000 different seeds were tested against each server for Mutiny-fuzzer and Fuzzotron, respectively. During the fuzzing process, the connection with the tested server was continuously monitored. This was done with the help of a custom-made Python3 script, which was requesting every one sec the URL of the website stored in the server. If the fulfilment of the request exceeded 5 sec, the script logged the date and time of the request for subsequently finding the matching seed. This was done by means of the *Wireshark* v3.6.5 tool running in the background and filtering only the traffic that was relevant to the targeted server. After the fuzzing stage was over, we examined manually different attack scenarios that may apply in real-life situations as explained in the following subsections. It is to be noted that IP spoofing was not assessed, as Azure VMs come with native anti-spoofing protection[15].

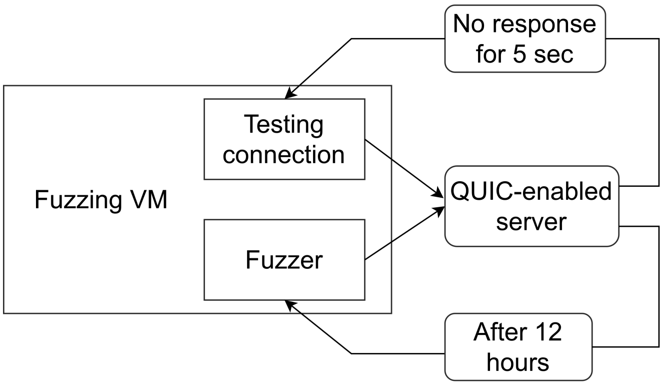

Fig. 3 A high-level view of the fuzzing procedure.

5.2.2 QUIC-fuzz

For specific fuzzing seeds, IIS 10 presented a delay of more than 5 sec for both the fuzzers, while OpenLiteSpeed and Caddy had a response delay for Fuzzotron and Mutiny-fuzzer, respectively. By examining the packet structure and manually inspecting the Proof of Concept (PoC) through a trial-and-error process, we managed to construct an exploit, as given in GitHub repository[16]. The exploit is written in Python 3 and

[11] https://github.com/Cisco-Talos/mutiny-fuzzer

[12] https://gitlab.com/akihe/radamsa

[13] https://github.com/denandz/fuzzotron

[14] https://gitlab.com/akihe/blab

[15] https://docs.microsoft.com/en-us/azure/security/fundamentals/production-network

[16] The (currently) private repository is at https://github.com/efchatz/QUIC-attacks. The reposi-

Table 3 Vulnerabilities identified per server. For QUIC-fuzz, the first and second figure in the parenthesis indicate the number of issues identified by manual or automatic fuzzing, respectively, while the ☆ and ✹ symbols designate the use of either the *Fuzzotron* or *Munity-fuzzer* fuzzing tool. The QUIC-downgrade attack depends on the capabilities of the client and the server, i.e., if both of them support gQUIC.

Name	OpenLiteSpeed	Caddy	NGINX	H2O	IIS	Cloudflare	Total
QUIC-fuzz	✓(0/1☆)	✓(1/1✹)	✓(0/1☆)	✓(1/0)	✓(1/2)	✗	5
QUIC-downgrade	–	–	–	–	–	–	–
QUIC-out-of-joint	✗	✓	✓	✓	✗	✗	3
QUIC-loris	✗	✓	✗	✗	✗	✗	1
QUIC-encapsulation	✗	✓	✗	✗	✗	✗	1
QUIC-flooding	✗	✓	✓	✓	✗	✗	3
Total	1	5	3	3	1	0	–

takes advantage of the Scapy v2.4.5 library. At a next step, the PoC was tested against all the servers of Table 2 and half of them (H2O, IIS 10, and Caddy) presented an increased delayed response of 6, 3, and 6 sec, respectively. Precisely, the opponent needs to send a small number of packets (about 30), and the delay when requesting a simple webpage persists for about 10 to 30 min depending on the server. It is to be noted that the attack does not affect OpenLiteSpeed, although it was pinpointed as potentially vulnerable by Fuzzotron. For Caddy, it was observed that the server presented identical behavior as with the QUIC-loris attack given in subsection 5.2.5.

Strangely, the current attack was not effective each time it was initiated. Namely, the PoC was tested 20 times against each server, but the affected servers were impacted in about 25% of the cases. The attack was also assessed under a greater number of packets (300), but the result was identical. Nevertheless, it is obvious that a more stable version of this attack could be critical against live infrastructure, i.e., adding a significant delay to the server responses is a high severity issue, especially for time-sensitive networking services like video streaming. Overall, it can be assumed that the attack affects the corresponding server's message parser, similarly to a hash-collision attack at the application layer[17].

5.2.3 QUIC-downgrade

With reference to the QUIC supported versions as tracked by W3Techs [70], currently about 7.9% of the total websites support gQUIC. Interestingly, the relevant list includes popular websites, like YouTube and Gmail. Recall that gQUIC is vulnerable to PKCS#1 v1.5 and DROWN attacks [40, 42] as explained in Section 4. Additionally, gQUIC is supported by Google Chrome[18]. This means that any user who visits an gQUIC-enabled website via Chrome may be susceptible to a MitM QUIC-downgrade attack. On the positive side, Firefox and Safari do not support gQUIC; Firefox[19] supports QUIC after QUIC's draft version 27, while Safari has this feature disabled by default.

To identify webpages that support gQUIC, the attacker can simply eavesdrop on the *alt-svc* HTTP header, which designates the server's QUIC supported versions. For instance, if the server supports gQUIC, it will advertise it along with other QUIC supported versions. For example, the *alt-svc* returned by the google.com website is: *alt-svc: h3=“:443”; ma=2592000,h3-29=“:443”; ma=2592000,h3-Q050=“:443”; ma=2592000,h3-Q046=“:443”; ma=2592000,h3-Q043=“:443”; ma=2592000, quic=“:443”; ma=2592000; v=“46,43”*. As observed, this header includes the gQUIC versions, referring to them as “Q0**”, say, Q050.

Interestingly, after examining older Google Chrome versions, namely 30.x.x to 80.x.x, on both a MS Windows 7 and 10 VM, none of them connected with gQUIC to any supported webpage as those included in [70]; the connection was always done with HTTP/2 over TCP. Therefore, in practice, this assault seems infeasible, but servers should drop gQUIC, at least versions equipped with gQUIC-native crypto.

5.2.4 QUIC-out-of-joint

Another identified issue is related to the Frame mangling attack [51]. Specifically, when running the Munity-fuzzer it was observed that, similarly to the Frame mangling attack, the fuzzer was able to pass arbitrary QUIC packets to the server either during the TLS handshake

tory will be made public upon paper acceptance. Note that the exploit has been also duly communicated to the development teams of the affected servers.

[17] https://github.com/reddit/snudown/security/advisories/GHSA-6gvv-9q92-w5f6

[18] https://www.chromium.org/quic/quic-faq/

[19] https://hacks.mozilla.org/2021/04/quic-and-http-3-support-now-in-firefox-nightly-and-beta/

or throughout the HTTP service. Half of the tested servers, namely Caddy, H2O, and NGINX, were found to be susceptible to this matter. Such an issue can break the establishment of the TLS tunnel or offer opportunity for exploiting other potential vulnerabilities as a knock-on effect.

5.2.5 QUIC-loris

The current attack, affecting the Caddy server, requires about 150 to 200 packets in 1 min, i.e., 2 to 3 packets per sec. Precisely, the assailant initiates about 100 parallel QUIC connections with the server, but drops each of them after 1 sec, typically as soon after the handshake completes. Then, they wait approximately 30 sec and re-initiate QUIC (TLS) handshakes again. This low-and-slow attack against the server significantly decreases its response capacity to user requests; the user needed more than 20 sec, and in some cases more that one min, to fetch the (basic) HTML webpage. Specifically, it was observed that the CPU of the targeted server was continuously under heavy stress exceeding 99%. Recall from section 2 that RFC 9000 does not cover stream fragmentation attacks. And indeed, based on our experiments, such assaults are feasible in practice. Given that this attack is especially drastic on this server, by following a Coordinated Vulnerability Disclosure (CVD) process, we reported it to the developers of the Caddy server, which in turn notified the *quic-go* team [13]. The developers acknowledged the issue, realizing that the probe timer of the Packetization Layer Path MTU Discovery *(plpmtud)* method is being overflowed, i.e., reaching values above 100K. This is because the opponent drops the connection soon after completing the QUIC handshake, forcing the *quic-go* to a state that is unable to sent a *Path MTU Discovery* packet. As a result, after that point, the CPU usage on the server is instantly over-utilized reaching 99.3% to 99.9%. Since this issue affects the *quic-go* implementation, other QUIC-enabled solutions also relying on this software may be susceptible to the same issue. The attack has been also reported to MITRE and has been assigned the ID CVE-2022-30591.

5.2.6 QUIC-encapsulation

Encapsulation is a legacy type of attack that is typically used to bypass firewall protection [71]. In fact, as detailed in Section 4, the work in [59] examined this type of assault for bypassing the firewall in QUIC realms. In our case, we examined similar attacks, by encapsulating different IP packets, e.g., by placing a TCP packet inside a UDP one or a UDP inside another UDP, and sent them against the server in an unsolicited manner. In most cases, these unsolicited packets did not have any effect to the targeted server. However, sometimes, Caddy responded with a TCP (keepalive probe) packet or a UDP one depending on the initial unsolicited packet received. Naturally, this behavior is aberrant and may provide opportunities for attacks, including bypassing the firewall protection, escalate a server-side request forgery (SSRF) assault, and so on.

5.2.7 QUIC-flooding

It was observed that by constantly attacking a server with 0-RTT connections increased its CPU usage significantly. This attack, particularly affecting Caddy, NGINX, and H2O, can be easily mounted by utilizing the *aioquic* Python library, which enables one to craft and send multiple QUIC requests. Obviously, the assault capitalizes on the number of concurrent QUIC requests; the more the requests, the greater the impact on the server's CPU. According to our tests, 100 parallel 0-RTT connections done by just a single attack instance are enough, but the impact seems to depend on the server's QUIC implementation. Specifically, NGINX was the only server that was temporarily paralyzed, i.e., after 2 to 3 attack rounds, or approximately 15 sec after the attack was launched, the server became unresponsive for about 30 sec. This behavior was observed every time the assault was active, meaning that the opponent is able to teardown all ongoing connections every 15 to 20 sec. For Caddy, we perceived an increased CPU usage of 99.9% almost instantly, a logical outcome since this server is also vulnerable to the QUIC-loris attack given in subsection 5.2.5. H2O demonstrated delayed HTTP responses exceeding 3 sec. No less important, this assault can be further exploited in a DDoS fashion, since a single aggressor is able to consume at least the one-fifth of the server's CPU. To mitigate this assault, servers can employ the *RETRY* feature of QUIC, as explained in subsection 4.2.

6 Challenges and future directions

The current section focuses on a set of challenges and future directions regarding QUIC security.

First, from a network operator's viewpoint, traffic analysis becomes more troublesome because QUIC traffic is fully obfuscated. This also poses a challenge for firewalls; at least for enterprise settings, next generation firewalls featuring deep packet inspection are of need. Moreover, with reference to § 2 of RFC 9000, QUIC does not provide any means of ensuring ordering between bytes on different streams. While QUIC encrypts the ordering number of the packet, an opponent may be able to exploit this mechanism, if they manage to find a latent vulnerability in this encryption method.

In certain cases, the application layer can utilize the connection during the QUIC (TLS) handshake phase. Namely, 0-RTT allows application data to be sent from or towards a client before even receiving a response from the server. This trait may for instance allow an application protocol to offer the option of trading some security guarantees for reduced latency. However, based on § 9.2 of RFC 9001, 0-RTT provides no protection against replay attacks. The work in [72] did propose a forward secrecy scheme for 0-RTT, but as recently demonstrated by [73], the protocol's speed is reduced when enabling forward secrecy on 0-RTT. That is, as expected, applying forward secrecy produces additional overhead, which as a result renders QUIC less antagonistic vis-à-vis TCP implementations.

As shown in Table 1, a plethora of QUIC implementations have been published as open access projects in GitHub or elsewhere. In fact, the Table contains 18 QUIC implementations, created with 7 different programming languages. Additionally, seven diverse TLS libraries are utilized across all these implementations. In this respect, and because all these implementations are new, not having passed the test of time, it is rather expected that they will not be bug-free. Moreover, due to this diversity, an identified vulnerability existent in one QUIC implementation may or may not affect others.

Future directions can focus on different unexplored areas of QUIC security. For instance, none of the works so far has examined the QUIC endurance against IP spoofing attacks. While QUIC does implement *Address Validation* protection, it should further be examined if this protection is indeed effective against all QUIC implementations, and how this protection along with the other QUIC protections, i.e., having specific UDP length in a packet, could be exploited in the context of a UDP amplification attack.

It seems that QUIC is rapidly gaining ground replacing TCP in several settings, say, P2PoverQUIC [74] for Peer-to-Peer (P2P) connections and RTPoverQUIC [75] for Real-time Transport Protocol (RTP) services, DoQ for encrypted DNS [76] etc. However, studies on QUIC for HTTP/3, DNS, Tor, SMBoverQUIC, and others, from the perspective of security are scarce. Precisely, thus far, only one short paper examined the privacy related issues of DNS over QUIC [77]. In this respect, future studies should concentrate more on the security model of the aforementioned QUIC implementations.

While web cache poisoning attacks are quite problematic against HTTP-based implementations [78, 79], thus far no work assessed cache poisoning attacks against QUIC, which does not offer a relevant countermeasure. Since QUIC is implemented in different proxies and load balancers, a possible direction for future work is to examine cache poisoning attacks in such infrastructures.

No less important, a stateful QUIC fuzzer could be beneficial towards identifying misconfigurations in the plethora of QUIC implementations. Since the testing carried out in Section 5.2.2 demonstrated that fuzzing can indeed identify potential issues, a QUIC-focused fuzzer could provide more accurate results, potentially revealing a much larger set of misconfigurations or bugs. Also in the same vein, a specialized tool able to craft and send QUIC packets is largely needed. Right now, only the *aioquic* Python library can be used to some extent for testing different attack behaviors against QUIC servers.

7 Conclusion

IETF QUIC is a newfangled transport for the Internet. It provides enhanced privacy and superior performance in demanding network conditions, rendering it highly desirable as a transport layer for HTTP. Therefore, opposite to HTTP/1.1 and HTTP/2, HTTP/3 uses QUIC as a transport, with TLS 1.3 acting as a security component of QUIC. The current work aspires to serve two purposes regarding QUIC security. First, to deliver a full-fledged and meticulous review of relevant literature contributions spanning two axes: those covering QUIC security evaluation and those identifying attacks against both IETF QUIC and gQUIC components. Actually, the latter part of this review effort covers five categories of attacks, namely cryptographic, handshake, fuzzing, transport-layer, and privacy. The second objective of the work at hand is to contribute the first to our knowledge empirical evaluation of QUIC in terms of security through fuzz testing. This effort, conducted on the currently six most common IETF QUIC production-grade server implementations, yielded a number of zero-days, some of which allowing an adversary to easily paralyze the targeted server. This was quite anticipated, given that QUIC implementations have not yet stood the test of time, and additionally there is currently a great diversity in the implementations; we identified 18 distinct QUIC implementations, making use of 7 different TLS libraries, created with 7 different programming languages. Given its dual purpose and its associated practical findings, we anticipate this work to serve as a cornerstone and point of reference for further research in this timely and challenging field.

Funding This study received no funding.

Conflict of Interest The authors declare that they have no conflicts of interest regarding the publication of this study.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Data Availability All data and code generated or used to support the findings of this study are included within the article.

References

[1] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/RFC7540. URL: https://www.rfc-editor.org/info/rfc7540.

[2] Adam Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 183–196. ISBN: 9781450346535. DOI: 10.1145/3098822.3098842. URL: https://doi.org/10.1145/3098822.3098842.

[3] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000.

[4] Martin Thomson and Sean Turner. *Using TLS to Secure QUIC*. RFC 9001. May 2021. DOI: 10.17487/RFC9001. URL: https://www.rfc-editor.org/info/rfc9001.

[5] Mike Bishop. *Hypertext Transfer Protocol Version 3 (HTTP/3)*. Internet-Draft draft-ietf-quic-http-34. Work in Progress. Internet Engineering Task Force, Feb. 2021. 75 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34.

[6] Joras,Matt and Chi,Yang. *How Facebook is bringing QUIC to billions*. last visited 03/11/2021. URL: https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/.

[7] lsquic. *LiteSpeed QUIC and HTTP/3 Library*. visited on 2022-02-15. URL: https://github.com/litespeedtech/lsquic.

[8] W3Techs. *Usage statistics of QUIC for websites*. last visited 29/03/2022. URL: https://w3techs.com/technologies/details/ce-quic.

[9] W3Techs. *Usage statistics of HTTP/3 for websites*. last visited 29/03/2022. URL: https://w3techs.com/technologies/details/ce-http3.

[10] Adam Langley, Wan-Teh Chang. *QUIC Crypto*. last visited 31/03/2022. URL: https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit#.

[11] Eric Rescorla and Brian Korver. *Guidelines for Writing RFC Text on Security Considerations*. RFC 3552. July 2003. DOI: 10.17487/RFC3552. URL: https://www.rfc-editor.org/info/rfc3552.

[12] RSnake and John Kinsella and Hugo Gonzalez and Robert E Lee. *Slowloris HTTP DoS*. visited on 2022-04-04. URL: https://web.archive.org/web/20150426090206/http://ha.ckers.org/slowloris/.

[13] quic-go. *A QUIC implementation in pure go*. visited on 2022-02-15. URL: https://github.com/lucas-clemente/quic-go.

[14] quiche. *Savoury implementation of the QUIC transport protocol and HTTP/3*. visited on 2022-02-15. URL: https://github.com/cloudflare/quiche.

[15] nghttp2. *nghttp2 - HTTP/2 C Library and tools*. visited on 2022-02-15. URL: https://github.com/nghttp2/nghttp2.

[16] msquic. *Cross-platform, C implementation of the IETF QUIC protocol*. visited on 2022-02-15. URL: https://github.com/microsoft/msquic.

[17] quinn. *Async-friendly QUIC implementation in Rust*. visited on 2022-02-15. URL: https://github.com/quinn-rs/quinn.

[18] reactor-netty. *TCP/HTTP/UDP/QUIC client/server with Reactor over Netty*. visited on 2022-02-15. URL: https://github.com/reactor/reactor-netty.

[19] neqo. *neqo - A QUIC library with Mozilla cooperation*. visited on 2022-02-15. URL: https://github.com/mozilla/neqo.

[20] mvfst. *An implementation of the QUIC transport protocol*. visited on 2022-02-15. URL: https://github.com/facebookincubator/mvfst.

[21] xquic. *XQUIC Library released by Alibaba is a cross-platform implementation of QUIC and*

HTTP/3 protocol. visited on 2022-02-15. URL: https://github.com/alibaba/xquic.

[22] aioquic. *QUIC and HTTP/3 implementation in Python.* visited on 2022-02-15. URL: https://github.com/aiortc/aioquic.

[23] ngtcp2. *ngtcp2 project is an effort to implement IETF QUIC protocol.* visited on 2022-02-15. URL: https://github.com/ngtcp2/ngtcp2.

[24] s2n-quic. *AWS — An implementation of the IETF QUIC protocol.* visited on 2022-02-17. URL: https://github.com/aws/s2n-quic.

[25] quicly. *A modular QUIC stack designed primarily for H2O.* visited on 2022-02-15. URL: https://github.com/h2o/quicly.

[26] picoquic. *Minimal implementation of the QUIC protocol.* visited on 2022-02-15. URL: https://github.com/private-octopus/picoquic.

[27] kwik. *A QUIC client, client library and server implementation in Java. Supports HTTP3 with "Flupke" add-on.* visited on 2022-02-15. URL: https://github.com/ptrd/kwik.

[28] quiche google. *Google's production-ready implementation of QUIC, HTTP/2, and HTTP/3.* visited on 2022-02-17. URL: https://github.com/google/quiche.

[29] nginx. *A QUIC server implementation for NGINX.* visited on 2022-02-15. URL: https://hg.nginx.org/nginx-quic/shortlog/quic.

[30] Hideki Sakurada et al. "Analyzing and Fixing the QACCE Security of QUIC". In: *Security Standardisation Research.* Ed. by Lidong Chen, David McGrew, and Chris Mitchell. Cham: Springer International Publishing, 2016, pp. 1–31.

[31] Robert Lychev et al. "How Secure and Quick is QUIC? Provable Security and Performance Analyses". In: *2015 IEEE Symposium on Security and Privacy.* 2015, pp. 214–231. DOI: 10.1109/SP.2015.21.

[32] Shan Chen et al. "Secure Communication Channel Establishment: TLS 1.3 (over TCP Fast Open) vs. QUIC". In: *Computer Security – ESORICS 2019.* Ed. by Kazue Sako, Steve Schneider, and Peter Y. A. Ryan. Cham: Springer International Publishing, 2019, pp. 404–426. ISBN: 978-3-030-29959-0.

[33] Jingjing Zhang et al. "Formal Analysis of QUIC Handshake Protocol Using Symbolic Model Checking". In: *IEEE Access* 9 (2021), pp. 14836–14848. DOI: 10.1109/ACCESS.2021.3052578.

[34] Marc Fischlin, Felix Günther, and Christian Janson. *Robust Channels: Handling Unreliable Networks in the Record Layers of QUIC and DTLS 1.3.* Cryptology ePrint Archive, Report 2020/718. https://ia.cr/2020/718. 2020.

[35] Antoine Delignat-Lavaud et al. "A Security Model and Fully Verified Implementation for the IETF QUIC Record Layer". In: *2021 IEEE Symposium on Security and Privacy (SP).* 2021, pp. 1162–1178. DOI: 10.1109/SP40001.2021.00039.

[36] Benjamin Dowling et al. "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol". In: *J. Cryptol.* 34.4 (2021), p. 37. DOI: 10.1007/s00145-021-09384-1. URL: https://doi.org/10.1007/s00145-021-09384-1.

[37] Joshua Maddux. "When TLS hacks you". In: *Black Hat* (2020).

[38] Vyron Kampourakis et al. "Revisiting man-in-the-middle attacks against HTTPS". In: *Network Security* 2022.3 (2022), null. DOI: 10.12968/S1353-4858(22)70028-1. eprint: https://doi.org/10.12968/S1353-4858(22)70028-1. URL: https://doi.org/10.12968/S1353-4858(22)70028-1.

[39] Daniel Bleichenbacher. "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1". In: *Advances in Cryptology — CRYPTO '98.* Ed. by Hugo Krawczyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–12. ISBN: 978-3-540-68462-6.

[40] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. "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.* CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, pp. 1185–1196. ISBN: 9781450338325. DOI: 10.1145/2810103.2813657.

[41] Elaine Barker (NIST). *Recommendation for Key Management: Part 1 – General — SP 800-57 Part 1 Rev. 5.* visited on 2022-03-29. URL: https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final.

[42] Nimrod Aviram et al. "DROWN: Breaking TLS Using SSLv2". In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* USENIX Association, 2016, pp. 689–706. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram.

[43] Cas Cremers et al. "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication". In: *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA,*

USA, May 22-26, 2016. IEEE Computer Society, 2016, pp. 470–485. DOI: 10.1109/SP.2016.35.

[44] B. Arunkumar and G. Kousalya. “Nonce reuse/misuse resistance authentication encryption schemes for modern TLS cipher suites and QUIC based web servers”. In: *J. Intell. Fuzzy Syst.* 38.5 (2020), pp. 6483–6493. DOI: 10.3233/JIFS-179729.

[45] D. Benjamin and C. A. Wood. *Importing External PSKs for TLS.* Internet-Draft draft-ietf-tls-external-psk-importer-07. Work in Progress. Internet Engineering Task Force, 2022. URL: https://www.ietf.org/id/draft-ietf-tls-external-psk-importer-07.html.

[46] Nir Drucker and Shay Gueron. “Selfie: reflections on TLS 1.3 with PSK”. In: *J. Cryptol.* 34.3 (2021), p. 27. DOI: 10.1007/s00145-021-09387-y. URL: https://doi.org/10.1007/s00145-021-09387-y.

[47] Antoine Saverimoutou, Bertrand Mathieu, and Sandrine Vaton. “Which secure transport protocol for a reliable HTTP/2-based web service: TLS or QUIC?” In: *2017 IEEE Symposium on Computers and Communications (ISCC).* 2017, pp. 879–884. DOI: 10.1109/ISCC.2017.8024637.

[48] Marc Fischlin and Felix Günther. “Replay Attacks on Zero Round-Trip Time: The Case of the TLS 1.3 Handshake Candidates”. In: *2017 IEEE European Symposium on Security and Privacy (EuroS P).* 2017, pp. 60–75. DOI: 10.1109/EuroSP.2017.18.

[49] Xudong Cao, Shangru Zhao, and Yuqing Zhang. “0-RTT Attack and Defense of QUIC Protocol”. In: *2019 IEEE Globecom Workshops (GC Wkshps).* 2019, pp. 1–6. DOI: 10.1109/GCWkshps45667.2019.9024637.

[50] Sangtae Lee, Youngjoo Shin, and Junbeom Hur. “Return of Version Downgrade Attack in the Era of TLS 1.3”. In: New York, NY, USA: Association for Computing Machinery, 2020, pp. 157–168. ISBN: 9781450379489. URL: https://doi.org/10.1145/3386367.3431310.

[51] Eva Gagliardi and Olivier Levillain. “Analysis of QUIC Session Establishment and Its Implementations”. In: *Information Security Theory and Practice.* Ed. by Maryline Laurent and Thanassis Giannetsos. Cham: Springer International Publishing, 2020, pp. 169–184. ISBN: 978-3-030-41702-4.

[52] Marcin Nawrocki et al. “QUICsand: Quantifying QUIC Reconnaissance Scans and DoS Flooding Events”. In: *Proceedings of the 21st ACM Internet Measurement Conference.* New York, NY, USA: Association for Computing Machinery, 2021, pp. 283–291. ISBN: 9781450391290. URL: https://doi.org/10.1145/3487552.3487840.

[53] caida. *UCSD Network Telescope.* visited on 2022-02-15. URL: https://www.caida.org/projects/network_telescope/.

[54] Kenneth L. McMillan and Lenore D. Zuck. “Formal Specification and Testing of QUIC”. In: *Proceedings of the ACM Special Interest Group on Data Communication.* SIGCOMM ’19. Beijing, China: Association for Computing Machinery, 2019, pp. 227–240. ISBN: 9781450359566. DOI: 10.1145/3341302.3342087.

[55] Gaganjeet Singh Reen and Christian Rossow. “DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies for QUIC”. In: *Annual Computer Security Applications Conference.* ACSAC ’20. Austin, USA: Association for Computing Machinery, 2020, pp. 332–344. ISBN: 9781450388580. DOI: 10.1145/3427228.3427662.

[56] quant. *a QUIC implementation written in C.* visited on 2022-02-15. URL: https://github.com/NTAP/quant.

[57] Jingjing Zhang et al. “A Systematic Approach to Formal Analysis of QUIC Handshake Protocol Using Symbolic Model Checking”. In: *Secur. Commun. Networks* 2021 (2021), 1630223:1–1630223:12. DOI: 10.1155/2021/1630223.

[58] Kashyap Thimmaraju and Björn Scheuermann. “Count Me If You Can: Enumerating QUIC Servers Behind Load Balancers”. In: *Conference on Networked Systems 2021 (NetSys 2021).* Vol. 80. Electronic Communications of the EASST, 2021. DOI: 10.14279/tuj.eceasst.80.1172.

[59] Konrad Yuri Gbur and Florian Tschorsch. “A QUIC(K) Way Through Your Firewall?” In: *CoRR* abs/2107.05939 (2021). arXiv: 2107.05939. URL: https://arxiv.org/abs/2107.05939.

[60] Van Tong et al. “A novel quic traffic classifier based on convolutional neural networks”. In: *2018 IEEE Global Communications Conference (GLOBECOM).* IEEE. 2018, pp. 1–6.

[61] Ghada Arfaoui et al. *The privacy of the TLS 1.3 protocol.* Cryptology ePrint Archive, Report 2019/749. https://ia.cr/2019/749. 2019.

[62] Yashodhar Govil, Liang Wang, and Jennifer Rexford. “MIMIQ: Masking IPs with Migration in QUIC”. In: *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20).* USENIX Association, Aug. 2020.

[63] mininet. *mininet.* visited on 2022-02-15. URL: http://mininet.org/.

[64] Pengwei Zhan, Liming Wang, and Yi Tang. "Website fingerprinting on early QUIC traffic". In: *Computer Networks* 200 (2021), p. 108538. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2021.108538.

[65] Ludovic Barman et al. *This is not the padding you are looking for! On the ineffectiveness of QUIC PADDING against website fingerprinting.* 2022. DOI: 10.48550/ARXIV.2203.07806. URL: https://arxiv.org/abs/2203.07806.

[66] *Near-path NAT for IP Privacy.* last visited 13/04/2022. URL: https://github.com/bslassey/ip-blindness/blob/master/near_path_nat.md.

[67] *Multiplexed Application Substrate over QUIC Encryption (masque).* last visited 13/04/2022. URL: https://datatracker.ietf.org/wg/masque/about/.

[68] W3Techs. *Usage statistics of web servers.* last visited 29/04/2022. URL: https://w3techs.com/technologies/overview/web_server.

[69] Caddy. *GitHub — HTTP/3 seems to stop working after any kind of reload.* visited on 2022-03-22. URL: https://github.com/caddyserver/caddy/issues/4348.

[70] W3Techs. *Usage statistics of QUIC for websites.* visited on 2022-03-22. URL: https://w3techs.com/technologies/details/ce-quic.

[71] Seny Kamara et al. "Analysis of vulnerabilities in Internet firewalls". In: *Computers & Security* 22.3 (2003), pp. 214–232. ISSN: 0167-4048. DOI: https://doi.org/10.1016/S0167-4048(03)00310-9. URL: https://www.sciencedirect.com/science/article/pii/S0167404803003109.

[72] Felix Günther et al. "0-RTT Key Exchange with Full Forward Secrecy". In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III.* Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10212. Lecture Notes in Computer Science. 2017, pp. 519–548. DOI: 10.1007/978-3-319-56617-7_18. URL: https://doi.org/10.1007/978-3-319-56617-7%5C_18.

[73] Fynn Dallmeier et al. "Forward-Secure 0-RTT Goes Live: Implementation and Performance Analysis in QUIC". In: *Cryptology and Network Security - 19th International Conference, CANS 2020, Vienna, Austria, December 14-16, 2020, Proceedings.* Ed. by Stephan Krenn, Haya Shulman, and Serge Vaudenay. Vol. 12579. Lecture Notes in Computer Science. Springer, 2020, pp. 211–231. DOI: 10.1007/978-3-030-65411-5_11. URL: https://doi.org/10.1007/978-3-030-65411-5%5C_11.

[74] draft-aboba-avtcore-quic-multiplexing-04. *QUIC Multiplexing.* visited on 2022-05-19. URL: https://datatracker.ietf.org/doc/draft-aboba-avtcore-quic-multiplexing/04/.

[75] draft-engelbart-rtp-over-quic-03. *RTP over QUIC.* visited on 2022-05-19. URL: https://datatracker.ietf.org/doc/draft-engelbart-rtp-over-quic/.

[76] Georgios Kambourakis and Georgios Karopoulos. "Encrypted DNS: The good, the bad and the moot". In: *Computer Fraud & Security* 2022.5 (2022), null. DOI: 10.12968/S1361-3723(22)70572-6.

[77] Guannan Hu and Kensuke Fukuda. "An analysis of privacy leakage in DoQ traffic". In: *CoNEXT-SW '21: Proceedings of the CoNEXT Student Workshop, Virtual Event / Munich, Germany, 7 December 2021.* Ed. by Gareth Tyson, Hannaneh Barahouei Pasandi, and Lars C. Wolf. ACM, 2021, pp. 7–8. DOI: 10.1145/3488658.3493782. URL: https://doi.org/10.1145/3488658.3493782.

[78] Omer Gil. "Web cache deception attack". In: *Black Hat USA* 2017 (2017).

[79] "Web Cache Deception Escalates!" In: *31st USENIX Security Symposium (USENIX Security 22).* Boston, MA: USENIX Association, Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/mirheidari.