Securing QUIC Initial Packets

Keely Podosin
Stanford University

Nicholas Ingino
Stanford University

Ashley Dai
Stanford University

1 Introduction

On April 7, 2024, the Great Firewall of China began blocking QUIC connections to specific domains by decrypting the QUIC client Initial packet and censoring the destination based on the listed SNI [15]. This was the first known instance of the censorship of QUIC traffic, and it resulted because the QUIC client Initial packet can be decrypted with a predetermined salt and hash (specified by the RFC) by any party that sees the connection [14].

Internet censorship is a problem that affects millions of people and their ability to access information online. The emergence of the QUIC protocol was thought to make censorship more difficult, as packets are fully encrypted and not vulnerable to deep packet inspection or other data-level means of censorship; however, these newfound abilities of the GFW to censor QUIC packets based on the client Initial put Internet users at risk of blockage once again. Although a large portion of the Internet is now naturally using QUIC [4], censors' new ability to block these connections still leaves Internet users from censored regions relying on censorship circumvention tools like VPNs and Tor. If censors became unable to block QUIC traffic due to encrypted client Initials, and enough of the Internet was using QUIC that it would be an economic disadvantage for the GFW to block all QUIC connections, the GFW would likely be forced into allowing all QUIC traffic unless they found another fingerprinting mechanism. In the cat-and-mouse game of censorship circumvention, forcing the GFW into allowing QUIC traffic regardless of SNI would be a large success.

Our goal with this project is to encrypt these QUIC Initial packets under a server's known public key so that an on-path adversary like the GFW cannot decrypt the QUIC Initial packet and block the rest of the conversation if the SNI hostname is on a block list. We plan on achieving these goals by hard coding public keys that can be used to encrypt a QUIC client Initial packet, and measuring the effects of this encryption in terms of latency and accessibility. We also aim to discuss means of storing these public keys so a large number of QUIC Initials can be encrypted without requiring hardcoding. We store these keys within the QUIC implementations now as a proof of concept, but in the future these public keys could be stored in the browser, or ultimately shared via a service like DNS.

2 Related Works

2.1 QUIC Overview.

QUIC is a UDP-based network protocol that was standardized by the IETF as RFC 9000 [9]. QUIC is the transport protocol used for HTTP/3 and is now used for over 30% of web requests according to estimates from Cloudflare [4]. QUIC is important for the anti-censorship community as it encrypts all packets by default, blocking a passive adversary from learning the contents of a packet.

QUIC Client Initial. The QUIC Client Initial is the first packet in a QUIC handshake. Like all QUIC packets, the QUIC Client Initial is encrypted, but, unlike the rest of the packets in the QUIC exchange, the QUIC Client Initial is sent before key exchange has taken place. This requires the QUIC Client Initial to be encrypted under a known public key that can be decrypted by the target server. These known public keys are derived from the Destination Connection ID and a salt determined by the version of QUIC used [14]. This public decryption algorithm does not provide confidentiality or integrity, as it does not stop an observing party from decrypting the Initial packet with this known key. Instead, it only protects against off-path spoofing attacks [15]. Once the Initial packet is decrypted, it reveals a TLS 1.3 Client Hello message, which often includes SNI information for the host the client is attempting to connect to. All of this information can be read by a party that decrypts the QUIC Initial packet.

SNI-Based QUIC Blocking. SNI-based QUIC blocking was not observed prior to 2025 [15]. Instead, QUIC blocking was performed by blocking all UDP traffic to QUIC endhosts [7].

However, recently, Zohaib et al. found that the GFW decrypts QUIC Initial packets at scale, applies heuristic filtering rules, and uses a blocklist distinct from other censorship mechanisms to block QUIC-specific traffic [15]. They also discover that a critical flaw in this censorship system: the computational overhead required to decrypt each QUIC Initial packet reduces the effectiveness of QUIC censorship under larger traffic loads.

Format of QUIC Initial Packet. The header structure of a QUIC Initial packet contains bytes for: header from, fixed big, long packet type, reserved bits, packet number length, version, destination connection ID length, destination connection ID, source connection ID length, source connection ID, token length, length, and packet number. The IETF Draft on Protected Initial QUIC Packets suggests that an encryption context length and encryption context could be added to the Initial packet, which allows information about the encryption context of this packet to be added, including the configuration parameters for encryption, symmetric cipher key derivation function, and the symmetric key cipher AEAD ID [6]. We do not organize our packet this exact way, instead opting for an encryption technique that wraps the Initial packet without the header; however, this is a feasible alterative.

Encrypted Client Hello. Encrypted Client Hello (ECH) is a tool that allows clients to encrypt part of their TLS Client Hello using asymmetric encryption with a key obtained from DNS [15]. This is done by separating the Client Hello into an outer unencrypted message, and an inner message containing the encrypted SNI [3]. There has been some research done on ECH's ability to circumvent censorship, but the authors note that ECH is currently not resistant to censors in Russia or Uzbekistan [10] [15]. ECH is a model that we base our encryption on; however, we choose to encrypt the whole Initial packet, not just the SNI, in order to provide further measures of privacy and not allow intermediate sources like an ISP to decrypt the SNI.

2.2 Public Key Acquisition

Certificate Pinning Approach Certificate pinning is security tool used to validate certificates. Certificate pinning allows the browser to maintain a list of preconfigured public keys or certificates that it trusts [13]. We choose to explore an approach similar to certificate pinning in our exploration of encrypted QUIC Initial packets where we save commonly used public keys in the browser to automatically encrypt Initial packets with. This approach has the drawback of maintenance complexity and storage; storing public keys for a lot of different sites will increase the usability of the system, but also increase the storage complexity required. To account for the drawbacks of pinning public keys, we also consider the DNS-based approach below.

DNS-based Approach IETF has published a draft on protected QUIC Initial packets, which suggested that the client could obtain a public key that is distributed via DNS to support Encrypted Client Hello (ECH) [6]. This public key would then be used to encrypt a shared secret, sent in the Initial packet header, that both the client and server can use. This has been done in practice with several deployed systems already using DNS as a practical channel for public-key distribution. The DomainKeys Identified Mail (DKIM) standard [5] publishes public keys in DNS TXT records to authenticate email domains. Similarly, SSHFP records [12] and the DANE framework [8] enable clients to verify TLS or SSH keys using DNSSEC-authenticated responses rather than relying solely on centralized certificate authorities. The IETF's Encrypted Client Hello (ECH) draft extends this idea to HTTPS by embedding HPKE public keys in DNS HTTPS/SVCB records [2, 11]. ECH has already been deployed in major browsers and CDNs such as Cloudflare and Firefox, proving that large-scale DNS-based key distribution for encrypting handshake metadata is feasible in practice. These precedents show that distributing QUIC Initial encryption keys through DNS is consistent with deployed security mechanisms.

3 Encryption Design

We chose a specific design for encrypting our packets to minimize latency and space complexity, while providing key-private encryption and forward secrecy.

3.1 Key-Private Public Key Encryption

It is important that our design uses key-private public key encryption. Key-private public key encryption ensures that an eavesdropper who sees our ciphertexts gains no knowledge about which public key created the ciphertext [1]. Encrypting Initial packets would not be useful without key-private public key encryption; if the ciphertexts leaked which public key was used to encrypt it, it defeats the purpose of hiding the SNI of an Initial packet. Bellare et al. demonstrates that Diffie Hellman based key systems are naturally key-private, therefore, we choose to use Diffie Hellman based public and priavte keys in our implementation. To generate a Diffie Hellman based key pair, we ran `openssl genpkey -algorithm x25519 -out x25519-priv.pem` and `openssl pkey -in x25519-priv.pem -pubout -out x25519-pub.pem`. These keys are hardcoded into our code (which is publicly available on GitHub) and therefore should never actually be used in practice. They are only for the purpose of demonstrating the functionality of this system. If used outside a testing environment, be sure to follow best practices storing the key pairs.

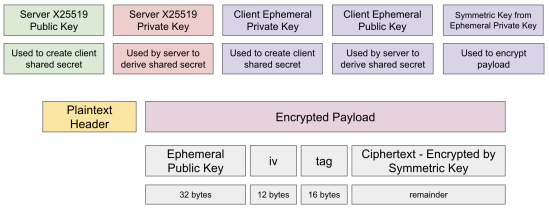

Figure 1: The structure of the protected Initial packet and keys used for the encryption.

3.2 Encryption and Decryption Technique

To encrypt the client Initial packet, we first load in the server public key. We then create our own ephemeral private key (new each time a new Initial packet is encrypted) and a matching ephemeral public key to go with our ephemeral private key. This ephemeral private key is used in conjunction with the server public key to create a shared secret between the client and server. We then use an HKDF to transform the shared secret into a symmetric key that is able to encrypt the payload data of the Initial packet. We encrypt the Initial packet's payload using AES-GCM with a randomly generated nonce (iv). This data is then sent to the server as a ciphertext of the form: ephemeral public key | iv | tag | ciphertext. To decrypt the packet on the server side, these steps are run in reverse, using the server private key and ephemeral public key to derive the shared secret. This technique can be visualized in Figure 1.

4 Implementation

Our original plan was to implement encryption of the client Initial packet directly in Chromium. However, after a few weeks of struggling with Chromium, we were unable to get a working implementation. Given our limited remaining time, we pivoted to first implementing a prototype in Python and then implementing encrypted client Initials in Google's `QUICHE` library.

4.1 Initial Prototype in Python

To test out our initial methodology, we implemented a version of encrypted client Initials in Python [1]. To do this, we utilized the `aioquic` package on Github. This library is a complete implementation of QUIC in Python, and we were able to modify the cryptography files to include specific encryption and decryption methods on the Initial packets. To encrypt the packets, we use the Python library `cryptography.hazmat.primitives`.

[1]This aioquic-based implementation is available in the following Github repo: https://github.com/kpodosin/350s_project_aioquic

4.2 Implementation in Google QUICHE

Once we had our initial Python prototype working, we implemented encrypted client Initials Google's `QUICHE` library [2]. We chose this library because it is the default implementation of QUIC used in Chromium browsers and is widely deployed. To do this, we added the `ProtectedInitialEncrypter` and `ProtectedInitialDecrypter` classes, which contain our methods to encrypt the Initial packets. We then replaced the default classes responsible for handling the Initial packet with our added classes. The underlying encryption was done using the `OpenSSL` library.

5 Evaluation

For both the `aioquic` and `QUICHE` libraries, encrypting the QUIC Iitial packet does not incur much overhead in terms of storage. The unmodified libraries use 16 additional bytes, beyond the bytes of the actual header and payload, to encrypt the packet. After modification, 60 bytes are used to encrypt the packet, meaning that this new approach uses 44 additional bytes per QUIC client Initial packet.

5.1 Evaluation of Python Prototype

To evaluate the speed overhead, we tested the protected QUIC packets using a test from the `aioquic` library that performs a simple client-to-server connection. The client initiates a HTTP/3 connect request to the server, which includes the Initial packets, handshake, etc. of the beginning of a QUIC connection. Then, it sends a few GET requests and closes the connection.

The unmodified `aioquic` library takes 0.021925 seconds to run the simple server-client test. After modification to enable protected Initial packets, the test takes 0.028552 seconds to complete, which is roughly a 30% increase.

5.2 Evaluation of Google's QUICHE

We tested our Google `QUICHE` implemetation using the `quic_server` and `quic_client` programs which are bundled with the library. The `quic_server` program locally hosts a web server that expects QUIC packets. We chose to host `www.example.org`, which is the suggested website in the `QUICHE` documentation. The `quic_client` program connects to a server using QUIC and sends GET requests to retrieve a webpage. Once the webpage has been retrieved, `quic_server` sends a few keepalive packets before terminating the connection. For our measurements, we exclude the keepalive packets.

The unmodified `QUICHE` library takes 0.005580 seconds to retrieve the webpage. After enableing protected Initial packets,

[2]This QUICHE-based implementation is available in the following Github repo: https://github.com/kpodosin/350s_project_google_quiche

it takes 0.005894 seconds to retrieve the webpage, which is a 5.6% increase.

6 Security + Privacy Properties

Our design aims to protect the contents of QUIC Initial packets from a passive on-path adversary while preserving QUIC's constraints and deployability.

6.1 Confidentiality of Handshake Metadata

The primary privacy property our scheme provides is confidentiality of the initial handshake metadata. In standard QUIC, the client's Initial packet is encrypted under a version specific publicly known key derived from a fixed salt and the destination connection ID [14]. Any passive adversary, such as the GFW, can deterministically derive this key, decrypt the Initial packet, and extract the TLS ClientHello fields, including SNI and other metadata which allow the adversary to fingerprint the destination server and block connections.

Our scheme replaces this predictable encryption with strong symmetric encryption derived from a fresh ephemeral Diffie Hellman key exchange. Assuming X25519, HKDF, and AES-GCM are secure, a passive observer cannot compute the symmetric key without the server's private key and therefore cannot recover the initial plaintext.

Our approach preserves the privacy and security properties already provided by QUIC. QUIC's use of AEAD encryption provides ciphertext integrit and QUIC's built in packet number validation and handshake state machine prevent replay attacks. Beyond the Initial packet, QUIC encrypts all subsequent handshake and application data packets, ensuring full confidentiality and integrity for the remainder of the connection. By strengthening the privacy of the Initial packet while maintaining QUIC's existing protections, our design preserves the security posture of QUIC and improves the resistance of QUIC users to passive network level censorship.

6.2 Forward Secrecy

Our scheme uses a fresh ephemeral X25519 key pair for every Initial packet, ensuring that each connection derives a unique symmetric key. This prevents linkability between connections and limits the blast radius of key compromise on the client side. In particular, compromise of a client's ephemeral private key does not reveal past or future session keys, nor does it allow an attacker to derive the server's long term private key.

However, the construction does not provide full forward secrecy. If the server's long term private key is compromised, a passive adversary who recorded encrypted Initial packets can decrypt them.

Achieving true forward secrecy would require the server to also contribute an ephemeral Diffie Hellman public key for each connection. However, at the point where the client sends the Initial packet, the server has not yet sent anything the client only has the server's long term static public key, which is distributed out-of-band (via DNS). The QUIC Initial format does not provide a way for the server to deliver a per connection ephemeral key before decrypting the Initial packet, and adding such a field would require changing the QUIC wire format.

6.3 Key Privacy

A critical requirement for circumventing censorship is key privacy, meaning ciphertexts should not reveal which public key they were encrypted to [1]. Without key privacy, a censor could classify traffic simply by recognizing which server the ciphertext's public key corresponds to, even if it canot decrypt the payload. Our design with DH based encryption (X25519), is naturally key private since:

- The only public value the client sends is its ephemeral DH public key.
- The ephemeral public key is statistically independent of the server's public key.
- The AES GCM ciphertext is keyed by the DH derived symmetric key, but the ciphertext distribution depends only on the shared secret, not on which public key produced it.
- The server's public key never appears in any on wire field.

Therefore, a passive adversary cannot determine which server's public key was used. Many alternative encryption approaches are not key private and would leak server identity to the attacker. For example, if we encrypted the Initial packet using RSA, the resulting ciphertext would directly depend on the server's RSA modulus. Since the modulus is a long term, globally unique identifier for the server's public key, a passive adversary could simply maintain a map from modulus to domain. The ciphertext distribution is also key dependent. As a result, a censor could identify the destination server solely by examining the ciphertext, even without decrypting it.

6.4 Assumptions

Our guarantees rely on the following assumptions:

- X25519 is a secure DH group
- HKDF-SHA256 securely derives symmetric keys from DH shared secrets
- AES-GCM provides confidentiality and integrity, and nonces are not reused
- The server's long term private key remains uncompromised
- DNS servers are secure and honest

7 Design Limitations

While our approach significantly improves the confidentiality of QUIC Initial packets, it introduces several limitations and relies on assumptions that may not hold in all deployment environments.

7.1 Server Side Computational Overhead

A core limitation of our design is the additional computational cost imposed on servers. Each Initial packet requires a full X25519 Diffie Hellman operation, HKDF expansion, and an AES GCM decryption. Under high traffic volumes, these operations may place substantial load on large deployment endpoints. In contrast, default QUIC Initial decryption uses only a lightweight hash based derivation with publicly known keys, which is considerably cheaper. Large scale deployments handling millions of Initial packets per second may find the additional overhead prohibitive without dedicated cryptographic acceleration or offloading. Quantifying this overhead in real deployments remains an important area for future evaluation.

7.2 DNS Based Key Distribution

If we were to expand our functionality to support DNS-based key acquisition, our privacy guarantees would rely on authentic distribution of the server's static public key through DNS. If DNS responses are forged, censored, or manipulated, an attacker could inject their own public key and cause clients to encrypt the initial package with an attacker controlled key, enabling trivial decryption. This would make our scheme dependent on DNSSEC, HTTPS/SVCB validation, or other authenticated DNS channels. In practice, many networks do not deploy DNSSEC, and DNS infrastructure is often controlled by ISPs or state level actors. For example, if the authoritative DNS servers for a domain are operated or intercepted by a censor such as the GFW, key distribution through DNS becomes a single point of failure and provides no protection. Thus, while DNS is convenient for distributing static keys, it may not be appropriate for all threat environments.

7.3 Methodology and Implementation Limitations

Due to time constraints and the complexity of modifying Chromium's QUIC stack, we were unable to fully test our design in a production browser environment. Our prototype operates in controlled Python and QUICHE based repository, which may not reflect performance, compatibility, or real world deployment challenges in modern browser stacks. A full browser integration would be necessary to assess latency impacts, compatibility with existing intermediate network devices, and interactions with QUIC version negotiation.

7.4 Deployment Challenges

Adopting this design at scale requires coordination across DNS operators, server operators, CDNs, and browser vendors. If some servers use an old key and others use a new one, or if DNS has not updated everywhere yet, clients may fail to connect. Keeping the key consistent across many servers and making sure DNS updates reach everyone can be difficult in real deployments.

7.5 No Protection Against Active Attackers or Side Channels

While encrypting the Initial packet prevents an adversary from directly reading the SNI, it does not stop them from learning information through traffic patterns. Packet sizes, timing, retry behavior, and QUIC version negotiation messages are still visible on the wire. A resourced attacker may use these patterns to guess which site a user is connecting to, especially if the site has distinctive traffic characteristics. Our design also adds a small amount of extra data to the Initial packet, such as the client's ephemeral public key. Although this increase is minor, it expands what the adversary can observe and could possibly create new fingerprinting opportunities.

In addition, our threat model assumes a passive attacker who only observes traffic. An active attacker can still block packets, forge or tamper with DNS responses, interfere with public key distribution, or force the client to restart the handshake. QUIC prevents some spoofing attacks, but an active censor still has many ways to disrupt or block connections even if it cannot decrypt the Initial packet.

8 Conclusion

We showed that it is possible to protect QUIC client Initials by using key-private public key encryption on the payload data of QUIC client Initials. Doing so required few additional bytes and minimal additional latency in the context of the entire connection. However, based on our limitations, we do not recommend deploying this version of protected Initials, as it does not afford any more protection than Encrypted Client Hello, which is already deployed. Instead, we recommend furthering support for Encrypted Client Hello and helping the increase the usage of ECH across different implementations of QUIC.

9 Acknowledgments

We would like to thank the CS350S class for a great quarter, and professor Emma Dauterman and CA Teddy Zhang for their feedback and advice. We would also like to thank Phillip Stephens for setting us up with a VM for this project, and Thea Rossman for her suggestions on which QUIC implementations to focus on.

References

[1] Mihir Bellare, Anna Boldyreva, Anita Desai, and David Pointcheval. Key-privacy in public-key encryption. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001 (Lecture Notes in Computer Science, Vol. 2248)*. Springer-Verlag, 2001. Extended abstract; full version available at https://cseweb.ucsd.edu/~mihir/papers/anonenc.pdf.

[2] Cloudflare. Good-bye ESNI, hello ECH! https://blog.cloudflare.com/encrypted-client-hello/, October 2020.

[3] Cloudflare. Encrypted client hello – the last puzzle piece to privacy. Cloudflare Blog, Sep 2023. Accessed: 2025-11-24.

[4] Cloudflare. Adoption & usage worldwide. https://radar.cloudflare.com/adoption-and-usage, 2025. Accessed: 2025-10-10.

[5] D. Crocker, T. Hansen, and M. Kucherawy. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376, September 2011.

[6] Martin Duke and David Schinazi. Protected quic initial packets. Internet Draft draft-duke-quic-protected-initial-03, IETF, October 2021. Expires 28 April 2022.

[7] K. Elmenhorst, B. Schütz, N. Aschenbruck, and S. Basso. Web censorship measurements of http/3 over quic. In *Proceedings of the ACM Internet Measurement Conference (IMC)*. ACM, 2021.

[8] P. Hoffman and J. Schlyter. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698, August 2012.

[9] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[10] Niklas Niere, Felix Lange, Nico Heitmann, and Juraj Somorovsky. Encrypted client hello (ech) in censorship circumvention. In *Proceedings of the 2025 Free and Open Communications on the Internet (FOCI '25)*, 2025. Issue 2 of FOCI 2025.

[11] Eric Rescorla, Martin Thomson, et al. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni, IETF, 2025. Work in Progress.

[12] J. Schlyter and W. Griffin. Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints. RFC 4255, January 2006.

[13] SSL Support Team. What is certificate pinning? Blog post, October 2023. Accessed: 2025-11-10.

[14] Martin Thomson and Sean Turner. Using TLS to Secure QUIC. RFC 9001, May 2021.

[15] Ali Zohaib, Qiang Zao, Jackson Sippe, Abdulrahman Alaraj, Amir Houmansadr, Zakir Durumeric, and Eric Wustrow. Exposing and circumventing sni-based quic censorship of the great firewall of china. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, page 20, USA, 2025. USENIX Association.