nQUIC: Noise-Based QUIC Packet Protection

Mathias Hall-Andersen
NCC Group

David Wong
NCC Group

Nick Sullivan
Cloudflare

Alishah Chator
Cloudflare

ABSTRACT

We present nQUIC, a variant of QUIC-TLS that uses the Noise protocol framework for its key exchange and basis of its packet protector with no semantic transport changes. nQUIC is designed for deployment in systems and for applications that assert trust in raw public keys rather than PKI-based certificate chains. It uses a fixed key exchange algorithm, compromising agility for implementation and verification ease. nQUIC provides mandatory server and optional client authentication, resistance to Key Compromise Impersonation attacks, and forward and future secrecy of traffic key derivation, which makes it favorable to QUIC-TLS for long-lived QUIC connections in comparable applications. We developed two interoperable prototype implementations written in Go and Rust. Experimental results show that nQUIC finishes its handshake in a comparable amount of time as QUIC-TLS.

CCS CONCEPTS

• **Security and privacy → Security protocols**; **Web protocol security**; • **Networks → Network protocol design**; **Transport protocols**;

ACM Reference Format:
Mathias Hall-Andersen, David Wong, Nick Sullivan, and Alishah Chator. 2018. nQUIC: Noise-Based QUIC Packet Protection. In *CoNEXT '18: Workshop on the Evolution, Performance, and Interoperability of QUIC, December 4, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3284850.3284854

1 INTRODUCTION

QUIC is a modern secure transport protocol originally developed by Google [12, 23] and now under standardization by the Internet Engineering Task Force (IETF) [16]. Alongside protocols such as CurveCP [6] and MinimaLT [27], was borne out of a need to avoid the difficulties of changing protocols implemented in kernel space. QUIC is designed to run entirely in user space, using only UDP for packet transmission. It includes features such as flow control, congestion control, reliability, and, importantly, peer authentication and encryption by default. Originally, Google's QUIC used a custom cryptographic handshake protocol [3] to derive shared traffic secrets. The in-progress IETF standard, henceforth referred to as QUIC-TLS, defines TLS 1.3 [29] as its cryptographic handshake protocol [33]. Although TLS 1.3 is greatly simplified in comparison to prior versions, and also has some amount of formal proofs of security [8], it is an intrinsically complex protocol due to legacy compatibility demands and non-trivial negotiation logic.

In this paper, we propose yet another cryptographic handshake protocol for QUIC. It is based on the Noise Protocol Framework [26], a specification for designing cryptographic handshake protocols used by popular applications such as WhatsApp [5], protocols such as WireGuard [10], and technologies such as the Bitcoin Lightning Network [4] use Noise for this purpose.

We present QUIC-Noise – henceforth referred to nQUIC – an instantiation of QUIC with Noise as the handshake protocol. We present experimental results via two implementations of the protocol: one in Rust and one in Golang. Our results suggest that nQUIC offers performance improvements over QUIC-TLS with significantly less cryptographic code necessary for the key exchange. We emphasize that nQUIC is not a wholesale replacement for QUIC-TLS. In particular nQUIC is not intended for the traditional Web setting where interoperability and cryptographic agility is essential. It aims to be a viable replacement in cases where QUIC-TLS is either excessive or ill-suited.

In section 2, we compare nQUIC's handshake to that of QUIC-TLS with respect to design, trust and implementation. In section 3, we document the nQUIC protocol and justify the rationale behind our design choices. In section 4, we give abstract details on the number of cryptographic operations dominating an nQUIC handshake as opposed to different types of QUIC-TLS handshakes. In section 5, we give concrete figures on the performance of nQUIC as opposed to QUIC-TLS. In section 6, we conclude with our contribution's results and future work.

2 QUIC SECURITY LAYER

In this section we first introduce the Noise protocol framework and then compare its potential as a candidate for QUIC's cryptographic handshake instead of the current TLS 1.3-based design of QUIC-TLS.

2.1 The Noise Protocol Framework

Noise [26] is a *framework* for specifying and instantiating protocols based on the Elliptic Curve Diffie-Hellman key exchange. Briefly, it offers a vast choice of flexible handshake protocols, all specified according to a simple language. For example, consider an unauthenticated Diffie-Hellman key exchange, wherein Alice (client, or initiator in Noise terms) and Bob (server, or responder in Noise terms) derive fresh keys x and y, respectively, exchange g^x and g^y (in multiplicative notation), and derive a shared secret g^{xy}. In Noise, this *pattern* is described with the following transcript:

CoNEXT '18, December 4, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6082-1/18/12…$15.00
https://doi.org/10.1145/3284850.3284854

```
-> e
<- e, ee
```

This transcript means that Alice sends Bob her public ephemeral share g^x (e), and Bob replies with his public ephemeral share g^y (e). Then, both perform a Diffie-Hellman key exchange using the keys (ee). The Noise specification identifies several useful handshake patterns which all have different names.

2.2 Protocol and Implementation Complexity

TLS 1.3 [29] is significantly improved over prior versions, influenced by academic work such as OPTLS [19], and by industry's contributions like Google's QUIC-crypto [3] and Facebook's Zero Protocol [20]. While it is substantially less complex than prior versions, it is still 160-pages long and mentions 44 other RFCs in its specification. Its design has also been influenced by middleboxes that were refusing to acknowledge packets that would look too different from older versions of the protocols [32]. Because of this, half of the fields contained in the first messages of a TLS handshake (client hello and server hello) are of no use, messages are also advertised with the versions TLS 1.2. Furthermore, most TLS libraries used for QUIC-TLS carry legacy support from older versions and provides extended cryptographic agility [21] to support a multitude of endpoints. As of this writing, the most popular TLS library OpenSSL [1] is around 703,173 lines of code and lists 165 Common Vulnerabilities and Exposures (CVE) on its page [2]. Conversely, most of the complexity of the Noise protocol framework is only present at design time. After that, the designed protocol is set in stone and has a very simple run-time behavior with a linear state machine and no cryptographic agility. Most of its implementations (without the cryptographic primitives) are under a thousand lines of code.

2.3 Integration Complexity

In the current design of QUIC-TLS, the bootstrapping of the session's cryptographic protection is delegated to TLS 1.3. Specifically, TLS is used to build handshake messages that are re-framed by QUIC (in CRYPTO frames carried in initial and handshake packets) and exchanged on top of it. (QUIC effectively implements the TLS 1.3 record layer.) Once the handshake is complete, the exporter functionality of TLS is used to export session keys, which are then used by QUIC itself to protect its packets.

Because of this particular use of the TLS protocol, a TLS implementation cannot be used as is. It needs to be heavily modified in order to extract its relevant parts. Moreover, many of the popular TLS implementations support multiple versions of TLS which makes it cumbersome to integrate with QUIC as QUIC only needs the latest version, TLS 1.3.

On the other hand, the Noise protocol framework is often implemented as a "build your own protocol" library. As a result, developers can directly leverage Noise libraries to cleanly embed the protocol within QUIC. From there, an nQUIC implementation will require no further cryptographic-related configuration besides the actual keys, unlike QUIC-TLS which still needs to be configured in order to set the supported key exchange algorithms, cipher suites, pre-shared key modes, etc.

2.4 Peer Authentication and Pinning

Authentication of at least one peer – commonly, the server – is necessary for an authenticated key exchange protocol. TLS typically use public key infrastructures (PKI-based) authentication, wherein servers use public keys certified by one or more trusted Certificate Authorities (CA). This is an error-prone process as history has shown [13, 22, 34]. Moreover, trusting a large number of CAs means that a single CA compromise [28] can compromise the entire system. Such a large system, with multiple points of authority led to the establishment of transparency systems (i.e. Certificate Transparency [24]) to add verifiability and accountability to the web PKI ecosystem, although the use of PKI is still controversial in a lot of different settings (e.g. mobile applications). In cases where the complexity of the web PKI is unnecessary, one approach has been to "pin" certificates in a certificate chain. Pinning is the process of expecting a *specific* certificate (or public key) in a chain when connecting to a specific host. Conceptually, this restricts the set of candidate certificates offered up by a specific host, reducing the surface area of CA compromise. Pinning can be applied to a number of levels in a certificate chain, each varying in flexibility and effectiveness [31].

When pinning an end-entity certificate, there is no reason to also include a certificate chain. Clients expect exactly one unique key when establishing a session. Failure to produce this key (with a valid signature covering a client-originated nonce) results in a handshake failure.[1] In many deployment scenarios, the client may be difficult to update, so to retain functionality in the event of a key compromise, the client may support a backup key pin in the case the primary key is revoked. In such cases, intermediate certificate pinning may offer more flexibility. This variant requires the complete certificate chain to be presented during a handshake, thereby increasing complexity.

In some deployment scenarios, both the client and the server have access to a system to quickly and securely modify pinned values in response to key rotation or compromise. In such cases, using certificates to package and transfer public keys is unnecessary. This may be applicable in the following cases:

(1) Public keys are obtained out-of-band, e.g., via a DNSSEC-secured resource using DNS-Based Authentication of Named Entities (DANE) [14].
(2) Public keys are obtained from a valid certificate chain obtained through some other out-of-band mechanism, e.g., via an LDAP server or web page.
(3) Public keys are provisioned to peers via some bootstrapping or initialization step, e.g., when devices are fabricated.
(4) Public keys are managed (and rotated accordingly) using a trusted key management service.

nQUIC is explicitly designed for applications supporting public key pinning. Note that infrastructure necessary to support public key pinning is not substantially different than that used to distribute pre-shared symmetric keys (PSKs). However, as PSKs must be unique per pair of endpoints, using PSKs in lieu of pinned public keys leads to substantially more keys distributed and managed.

[1] Note that there are no signatures computed in Noise or nQUIC.

Specifically, for n peers, $\binom{2}{n} = \frac{n!}{2\cdot(n-2)!}$ PSKs would be needed, instead of n public keys.

2.5 Default Security

TLS 1.3 comes with controversial traits that were introduced to speed up the protocol. For example, session resumption does not enforce forward secrecy by default and this choice is left to the user. Another example is the optional zero round-trip time mode (0-RTT) that clients can use on top of session resumption to encrypt application data in their very first flight of messages. This new feature is unfortunately removing forward secrecy from 0-RTT messages and rendering them replayable. Another example is session resumption without an additional ephemeral key exchange. Noise has none of these features and, currently, does not support session resumption with pre-shared keys. Moreover, Noise allows the client options and transport parameters to be encrypted during the handshake. This makes successful passive eavesdropping and ossification [32] less likely.

2.6 Design Complexity and Formal Analysis

As was previously said, the latest version of TLS builds on decades of bad legacy decisions and extensions to support, making it a dense specification which involves many other dense specifications. This makes it extremely hard to formally analyze the protocol as was shown by the partial symbolic analysis done by Cremers et al. [9]. Conversely, the Noise protocol framework has been fully analyzed via different symbolic proofs done with Tamarin [11], ProVerif [17] and CryptoVerif [25].

3 NQUIC DESIGN

nQUIC is based on the principle that cryptographic operations should be kept as simple as possible to achieve the desired effect. Per [16], the cryptographic handshake should satisfy or provide the following features:

(1) Authenticated key exchange, where:
   (a) The server is always authenticated.
   (b) The client is optionally authenticated.
   (c) Every connection produces distinct and unrelated keys, providing strong forward secrecy for application data, suitable for both 0-RTT and 1-RTT packets.
(2) Authentication of client/server transport parameters.
(3) Authenticated version negotiation.
(4) Authenticated negotiation of an application protocol.
(5) For the server, the ability to carry data that provides assurance that the client can receive packets that are addressed with the transport address that is claimed by the client.

By design, Noise satisfies feature requirement 1. Also, by using the payload feature of handshake messages, Noise can transport, encrypt and authenticate any application information. This includes transport parameters and, with version negotiation information inside the parameters, version negotiation. In this way, nQUIC satisfies requirements 2 and 3. Currently, Noise does not provide any cookie-based retry mechanism for requirement 5. However, this functionality is supported with address validation tokens in QUIC. Therefore, it is not essential for Noise to support. Lastly, with respect to requirement 4, nQUIC addresses this by putting ALPN data into the transport parameters for authenticated negotiation. Thus, in total, with only a minor change to support application protocol negotiation, nQUIC satisfies the cryptographic handshake requirements defined by the IETF for QUIC.

In the rest of this section, we describe the nQUIC handshake and show how it satisfies these features.

3.1 Noise Pattern

To accommodate *optional* client authentication and encryption of transport parameter data, nQUIC uses the IK handshake pattern, described as follows:

```
<- s
...
-> e, es, s, ss
<- e, ee, se
```

Where the server's static public key `s` is already known in advance. The first message consists of the client's ephemeral public key `e`; a Diffie-Hellman key exchange between that key and the server's static key; the client's static public key; a Diffie-Hellman key exchange between the client and the server static keys. The second and last message consists of the server's ephemeral public key; a Diffie-Hellman key exchange between that key and the client's ephemeral key; a Diffie-Hellman key exchange between the client's static key and the server's ephemeral key.

By design there are no branches in the handshake state machine, except handshake failure, which always causes the immediate transmission of a `CONNECTION_CLOSE` frame and termination of the connection. Both participants therefore expect to receive exactly one type of message at any time.

3.2 Justification and Limitations of IK

In this section we document our reasoning behind the choice of the IK handshake pattern. We also note the different trade-offs we had to make. The security properties mentioned in this section come from the manual analysis written in the Noise specification as well as the proofs computed by the Noise Explorer symbolic analysis[2].

Unknown Server. Unlike the IX or XX handshake patterns (the second letter X indicates that the server's static key is sent as part of the handshake), IK does not have the server communicate its long-term key as part of the handshake. This limits how nQUIC can be used: the client must know the server's long-term key in advance, and this even after the server has, for example, rotated its long-term keys. This requirement lets nQUIC avoid using a PKI to authenticate peers and avoids computationally heavy signatures as part of the handshake. In addition to IX and XX, we also compared IK to XK:

```
<- s
...
-> e, es
<- e, ee
-> s, se
```

The XK handshake pattern introduces forward secrecy before communicating the client's long-term key to the server. However, it also requires three message patterns as opposed to two with IK.

[2]Handshake Pattern Analysis of IK https://noiseexplorer.com/patterns/IK/

As this complicates the handshake, we opted against this pattern, though we note that changing to support this variant is trivial.

Computational Cost. As opposed to the same handshake patterns IX and XX, the IK handshake pattern needs to process one more Diffie-Hellman key exchange, this in turns invokes more calls to other symmetric primitives needed for processing. However, connection establishment time remains dominated by network latency. Since IK is part of the subset of the simplest two-way handshakes with only 1 round-trip, it is subsequently faster and easier to implement. This is especially important as we are working on top of a lossy protocol (UDP). After the first server to client message, all subsequent packets can be delivered out-of-order.

Forward secrecy of the first packet. Inclusion of the client identity in the first message introduces a few limitations. Compared to the XK handshake pattern, client identities and QUIC transport parameters are not protected if the server's long-term secret key is compromised. Since this message is also subject to replay, adversaries are able to check if the client still has access to the service by replaying the message and observing the server's response. However, we have concluded that this downside is of negligible importance compared to the benefits of this handshake pattern.

Key Compromise Impersonation. If the server's long-term key is compromised, an attacker can decrypt any client initial packet and obtain the underlying identity and the client's transport parameters. Moreover, as client static keys are assumed to be public, an attacker with knowledge of a specific client's static key can pretend to be said client during the first handshake message without knowledge of corresponding long-term private key. However, without this client's private key, the attacker cannot complete the handshake because it cannot encrypt and authenticate a short header packet after the two handshake messages.

3.3 Handshake Messages

The cryptographic handshake takes one round trip and consists of two message types, a handshake request and handshake response. All handshake messages contain only fixed sized fields, followed by a single variable length payload. This allows implementation with very minimal parsing, e.g., by casting the first part of the message into a packed struct in C. The contents of each message are as follows:

Handshake Request message. The Handshake Request message (see Figure 1) serves to:

- Indicate the start of the cryptographic handshake.
- Transmit encrypted client transport parameters.
- Optionally present the client's identity.

The fields are processed sequentially according to the Noise protocol specification. The payload contains the encrypted and authenticated client transport parameters. Note that these transport parameters are encoded according to the QUIC-TLS specification, allowing us to re-use QUIC implementations' already existing code to create and parse them.

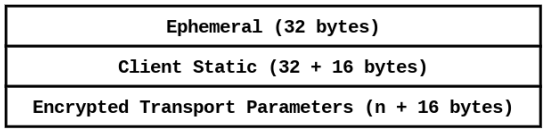

Figure 1: Handshake Request

Handshake Response message. The Handshake Response message (see Figure 2) serves the following purposes:

- Complete negotiation of transport keys.
- Transmit encrypted server transport parameters.
- Prove the server's identity.

The fields are processed sequentially according to the Noise protocol specification. The payload contains the encrypted and authenticated server transport parameters. After reception of this message, final keys are derived and passed to QUIC's packet protector (see 3.7)

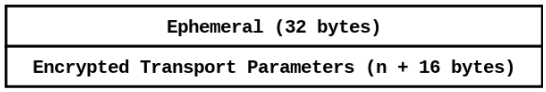

Figure 2: Handshake Response

Implicit Acknowledgement. After processing the Handshake Response message, the client must immediately transmit at least one packet encrypted under the sending key derived from the handshake. If no application data is available, `PADDING` or `PING` frames must be used to construct a probing packet. The server does not consider the handshake complete until it has received and successfuly decrypted a packet protected under the derived (1-RTT) transport keys. This mechanism serves to protect against replay and Key Compromise Attacks where a server would acknowledge a client's connection without verifying that the client can indeed process the last handshake message addressed to its public key. Note that `ACK` frames sent by the client in response to the server's last handshake message are not enough, as they must use the same level of encryption.

3.4 Authentication Modes

Static key validation is delegated to applications as described above. This allows multiple different authentication schemes to be constructed on top of nQUIC:

- Mutual authentication.
- Unauthenticated client, authenticated server.
- Server allowing both authenticated and unauthenticated clients.

To enable these different modes of authentication without introducing additional branches in the state machine nQUIC uses a Noise feature called "dummy keys" [26, Section 11.1, Dummy Keys], where a suitable constant is used as the client's public key (e.g. a 0 key) whenever client authentication is not desired.

3.5 Hostname Selection

In this document we do not specify a mechanism for routing nQUIC connections. Indeed, when multiple nQUIC endpoints are being

hosted on the same machine or when a load balancer sits in front of the destination endpoint, a mechanism needs to be employed in order to forward the initial QUIC packets to the requested machine. This is usually referred to as Server Name Indication or SNI in TLS[3]. To implement this mechanism in nQUIC, we can simply advertise a server's identification string in the first CRYPTO frame and postpone the actual Noise messages to the second CRYPTO frame. This indication can then be included as "prologue" data in the subsequent Noise handshake (as described in the Noise specification [26, 6. Prologue]), authenticating the client's intent to both peers and preventing the tampering of this value to lead to a successful handshake.

In cases where secrecy of this value matters, the client could use the static public key of the forwarding server to perform an N Noise handshake pattern and encrypt this value to the forwarding server. In order for this to work, the client must already know the forwarding server's public key. The concatenation of the client's ephemeral key and the encrypted value can then be used as "prologue" to the nQUIC handshake as is done in the previously proposed scheme. Note that this feature is non-existent in TLS at the moment, although efforts [15, 30] exist to encrypt the SNI extension as well.

3.6 Message Framing

Noise messages are encapsulated directly in CRYPTO frames, with the encrypted payload of the handshake messages containing the client or server transport parameters respectively. Figure 3 illustrates how the full handshake is encapsulated at the different layers.

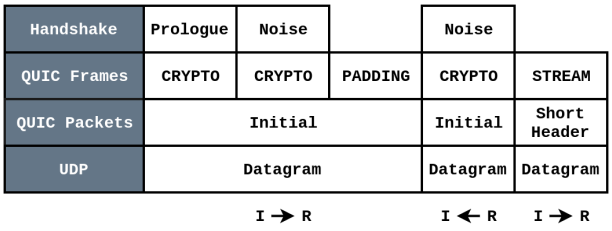

Figure 3: Cryptographic handshake framing

The first short header packet is included in Figure 3 to emphasize that it serves to confirm the handshake and that the server is prohibited from sending any further data before receiving and authenticating at least one packet from the client encrypted under 1-RTT keys. Note that both Noise messages are encrypted twice: once at the Noise layer and once by the packet protector at the QUIC packet layer. This is to simplify implementations of nQUIC and to directly leverage implementations of Noise without requiring additional modifications. The packet protector initially uses CID-derived keys until the handshake completes. This obfuscates the prologue from content-aware middle boxes. Noise encryption provides privacy for peer identities and transport parameters.

3.7 Ratcheting in nQUIC

After sending and processing the handshake response message, the nQUIC ratchet is initialized by using the Noise chaining key (ck) and handshake hash (hs) as the Input Key Material (IKM) and Salt respectively with HKDF [18] to derive three 256-bit outputs:

(1) The initial chain state.
(2) The first Server → Client transport key.
(3) The first Client → Server transport key.

Each pair of transport keys have an associated key phase bit, the initial pair of transport keys have phase 0. Every subsequent pair of transport keys have the negated phase of the previous. The KEY_PHASE field of short headers are set equal to the phase of the transport key encrypting the packet. When receiving a packet the KEY_PHASE field is examined to determine which of (at most) two possible transport key pairs should be used for decryption.

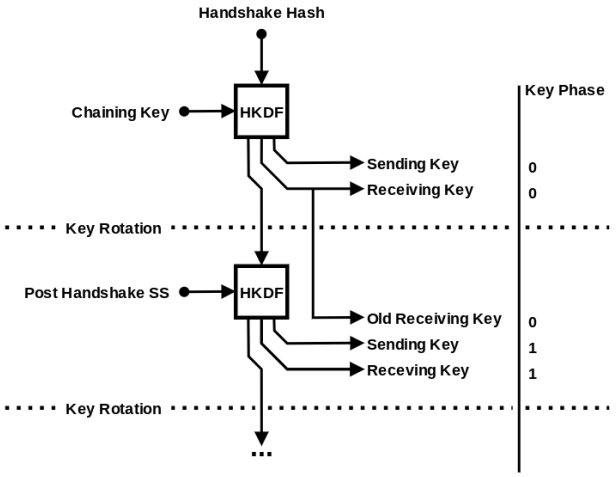

Figure 4: nQUIC double ratchet

To enable future secrecy, or post-compromise security [7], nQUIC periodically exchanges Diffie-Hellman shares to compute shared secrets after the handshake. When a shared secret (SS) has been computed the ratchet is updated by using SS and the chain state as IKM and Salt respectively with HKDF, as illustrated in Figure 4 (client side). New Diffie-Hellman shares are only sent after the endpoint successfully authenticates and decrypts a packet with a KEY_PHASE field equal to the key phase of the latest transport keys: encrypted packets serve to confirm the key exchange. The derivation of new input keying material and the one-way property of the KDF combined, provides both future and forward secrecy in nQUIC.

3.8 Versioning of nQUIC

nQUIC is designed as a minimal update to existing QUIC-TLS implementations. It could be specified as a new QUIC version compatible with the already existing QUIC RFCs. Thus, endpoints could support both TLS and Noise as possible handshakes without conflict. For now, we propose setting the first version of nQUIC to 0xff00000b. This is because the QUIC transport document states that "[QUIC] versions with the most significant 16 bits of the version number cleared are reserved for use in future IETF consensus".

[3]SNI is also used to figure out what certificate to present to the client

4 COST ANALYSIS

nQUIC's handshake costs may be estimated in terms of its constituent cryptographic operations. Furthermore, we will ignore symmetric cryptographic operations like key schedule, encryption, integrity protection, etc. as they are dominated by the cost of the public-key cryptography operations.

Let:

C_{key} be the cost of a key generation operation
C_{dh} be the cost of a key exchange operation
C_{sign} be the cost of a signing operation
C_{verif} be the cost of a signature verification operation

Table 5 and 6 compare the cryptographic costs of each QUIC protocols and their different key exchange variants.

Figure 5: nQUIC Key Exchange Cost Comparison

Authentication	Client Cost	Server Cost
server	$1C_{key} + 4C_{dh}$	$1C_{key} + 4C_{dh}$
mutual	$1C_{key} + 4C_{dh}$	$1C_{key} + 4C_{dh}$

Figure 6: QUIC-TLS Key Exchange Cost Comparison

Authentication	Client Cost	Server Cost
server	$1C_{key} + 1C_{dh} + XC_{verif}$	$1C_{key} + 1C_{dh} + 1C_{sign}$
mutual	$1C_{key} + 1C_{dh} + XC_{verif} + 1C_{sign}$	$1C_{key} + 1C_{dh} + YC_{verif} + 1C_{sign}$
psk	None	None
psk_dhe	$1C_{key} + 2C_{dh}$	$1C_{key} + 2C_{dh}$

Here X (resp. Y) refers to the number of certificates in the certificate chain presented by the server (resp. the client).

Notice that client and server costs are nearly symmetrical: for nQUIC, they include four Diffie-Hellman key exchanges and one ephemeral key generation. For QUIC-TLS, they include one ephemeral key generation, one Diffie-Hellman operation and at least one signature verification.

5 EXPERIMENTAL RESULTS

To assess nQUIC performance, we implemented and compared it against a similar QUIC-TLS implementations. In our analysis, we focus primarily on session establishment performance, specifically: we measure handshake performance and memory consumption. In practice the post-handshake phase of nQUIC has no difference compared to the one of QUIC-TLS. To make for a fair comparison, QUIC-TLS instances need to be configured with single-node certificate chains or PSKs. This is because nQUIC does not support standard PKI-based authentication. As discussed in Section 3, nQUIC assumes the equivalent of leaf public key pinning.

5.1 Implementations

We have created a proof of concept implementation (Ninn) of nQUIC based on a rust library (Quinn) which targets the IETF draft-11 version of the QUIC specification. The implementation (including both handshake and ratchet), consists of ≤ 6000 lines of Rust code counting the Noise library, of which the majority is not related to nQUIC specifically, but implements an extended version of the IETF transport-draft-11 which includes CRYPTO frames. Complexity can be further decreased by only implementing the particular handshake pattern used by nQUIC. Similarly, we have a Go implementation (nquic-go), based on (quic-go) which has an experimental library targeting IETF draft-11 as well. Reworking the extended draft-11 version of quic-go to support nQUIC required the addition of under 1000 lines of code.

5.2 Performance

Below we compare the connection establishment time between the original libraries ('Quinn', 'quic-go') based on 'Rustls' or 'Mint' respectively and the nQUIC handshake ('Ninn', 'nquic-go') based on the 'Snow' or 'Flynn/Noise' Noise libraries respectively. All tests are conducted over localhost on a Ubuntu 16.04.5 (Linux 4.4.0-131) machine with a Intel Xeon E5-2697A v4 CPU @ 2.60GHz and use x25519 for key exchange.

Implementation	Encryption	Hash	Handshake Time (s)	Handshake Bandwidth (B)
ninn (nQUIC)	AES-GCM-256	SHA-256	0.00135	1496
quinn (QUIC-TLS)	AES-GCM-256	SHA-384	0.00193	3426
quic-go (QUIC-TLS)	AES-GCM-256	SHA-384	0.02949	3230
quic-go (QUIC-TLS PSK)	AES-GCM-256	SHA-384	0.02689	2036
nquic-go (nQUIC)	AES-GCM-256	SHA-256	0.01023	1463

6 CONCLUSION AND FUTURE WORK

We presented nQUIC, a variant of QUIC-TLS that uses Noise for its key exchange algorithm and employs a DH-ratchet to provide post-compromise security. nQUIC offers improved performance, ease of implementation, drastically reduced code size, clear security properties and better protection in case of endpoint compromise. nQUIC does not work with standard PKI-based certificate authentication – it requires trust in static public keys – getting rid of an entire class of vulnerabilities. nQUIC may serve as a second standardized version of the QUIC protocol, which may help prevent network ossification around QUIC-TLS when deployed. For future work, we plan to bring nQUIC to the QUIC Working Group for discussion.

REFERENCES

[1] [n. d.]. OpenSSL. https://www.openssl.org/. ([n. d.]).
[2] [n. d.]. OpenSSL Vulnerabilities. https://www.openssl.org/news/vulnerabilities.html. ([n. d.]).
[3] [n. d.]. QUIC Crypto. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit. ([n. d.]).
[4] 2016. BOLT 8: Encrypted and Authenticated Transport. https://github.com/lightningnetwork/lightning-rfc/blob/master/08-transport.md. (2016).
[5] 2016. WhatsApp Encryption Overview. https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf. (2016).
[6] Daniel Bernstein. 2011. CurveCP: Usable security for the Internet. *URL: http://curvecp.org* (2011).
[7] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. 2016. On post-compromise security. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*. IEEE, 164–178.
[8] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, USA*.
[9] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. A Comprehensive Symbolic Analysis of TLS 1.3. http://doi.acm.org/10.1145/3133956.3134063. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM, New York, NY, USA, 1773–1788. https://doi.org/10.1145/3133956.3134063
[10] Jason A Donenfeld. 2016. WireGuard: Next generation kernel network tunnel. In *Proceedings of the 2017 Network and Distributed System Security Symposium, NDSS*, Vol. 17.
[11] Jason A. Donenfeld and Kevin Milner. 2017. *Formal Verification of the WireGuard Protocol*. Technical Report.
[12] Marc Fischlin and Felix Günther. 2014. Multi-stage key exchange and the case of Google's QUIC protocol. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1193–1204.
[13] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. The most dangerous code in the world: validating SSL certificates in non-browser software.
[14] Paul E. Hoffman and Jakob Schlyter. 2012. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. RFC 6698. (August 2012). https://doi.org/10.17487/RFC6698
[15] C. Huitema and E. Rescorla. 2018. *SNI Encryption in TLS Through Tunneling*. Internet-Draft. Internet Engineering Task Force. Work in Progress.
[16] Jana Iyengar and Martin Thomson. 2018. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-14. Internet Engineering Task Force. Work in Progress.
[17] Nadim Kobeissi. [n. d.]. Noise Explorer. https://noiseexplorer.com/. ([n. d.]).
[18] H. Krawczyk and P. Eronen. 2010. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869. (May 2010).
[19] Hugo Krawczyk and Hoeteck Wee. 2015. The OPTLS Protocol and TLS 1.3. Cryptology ePrint Archive, Report 2015/978. (2015). https://eprint.iacr.org/2015/978.
[20] Subodh Iyengar Kyle Nekritz. [n. d.]. Facebook Zero Protocol. https://code.fb.com/android/building-zero-protocol-for-fast-secure-mobile-connections/. ([n. d.]).
[21] Adam Langley. [n. d.]. Cryptographic Agility. https://www.imperialviolet.org/2016/05/16/agility.html. ([n. d.]).
[22] Adam Langley. 2014. PKCS1 signature validation. https://www.imperialviolet.org/2014/09/26/pkcs1.html. (2014).
[23] Adam Langley and Wan-Teh Chang. 2013. QUIC Crypto. (2013).
[24] B. Laurie, A. Langley, and E. Kasper. 2013. *Certificate Transparency*. RFC 6962. RFC Editor.
[25] Benjamin Lipp. 2018. *A Mechanised Computational Analysis of the WireGuard Virtual Private Network Protocol*. Technical Report.
[26] Trevor Perrin. 2017. Noise protocol framework. *https://noiseprotocol.org/* (October 2017).
[27] W Michael Petullo, Xu Zhang, Jon A Solworth, Daniel J Bernstein, and Tanja Lange. 2013. MinimaLT: minimal-latency networking through better security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 425–438.
[28] J Ronald Prins and Business Unit Cybercrime. 2011. DigiNotar Certificate Authority breach 'Operation Black Tulip'. *Fox-IT, November* (2011).
[29] E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. RFC Editor.
[30] Eric Rescorla, Kazuho Oku, Nick Sullivan, and Christopher A. Wood. 2018. *Encrypted Server Name Indication for TLS 1.3*. Internet-Draft draft-rescorla-tls-esni-00. Internet Engineering Task Force. Work in Progress.
[31] Chris McMahon Stone, Tom Chothia, and Flavio D Garcia. 2017. Spinner: Semi-Automatic Detection of Pinning without Hostname Verification. (2017).
[32] Nick Sullivan. [n. d.]. Why TLS 1.3 isn't in browsers yet. https://blog.cloudflare.com/why-tls-1-3-isnt-in-browsers-yet/. ([n. d.]).
[33] Martin Thomson and Sean Turner. 2018. *Using Transport Layer Security (TLS) to Secure QUIC*. Internet-Draft draft-ietf-quic-tls-14. Internet Engineering Task Force. Work in Progress.
[34] David A. Wheeler. 2014. The Apple goto fail vulnerability: lessons learned. https://www.dwheeler.com/essays/apple-goto-fail.html. (2014).