PQConnect:
Automated Post-Quantum End-to-End Tunnels

Abstract

This paper introduces PQConnect, a post-quantum end-to-end tunneling protocol that automatically protects all packets between clients that have installed PQConnect and servers that have installed and configured PQConnect.

Like VPNs, PQConnect does not require any changes to higher-level protocols and application software. PQConnect adds cryptographic protection to unencrypted applications, works in concert with existing pre-quantum applications to add post-quantum protection, and adds a second application-independent layer of defense to any applications that have begun to incorporate application-specific post-quantum protection.

Unlike VPNs, PQConnect automatically creates end-to-end tunnels to any number of servers using automatic peer discovery, with no need for the client administrator to configure per-server information. Each server carries out a client-independent configuration step to publish an announcement that the server's name accepts PQConnect connections. Any PQConnect client connecting to that name efficiently finds this announcement, automatically establishes a post-quantum point-to-point IP tunnel to the server, and routes traffic for that name through that tunnel.

PQConnect makes a conservative choice of post-quantum KEM for the foundation of security: the server's long-term public key used to encrypt and authenticate all PQConnect packets. PQConnect also uses a smaller post-quantum KEM for forward secrecy, and elliptic curves to ensure pre-quantum security even in case of security failures in KEM design or KEM software. Security of the PQConnect handshake has been symbolically proven using TAMARIN Prover.

1 Introduction

At the time of this writing (2023-10-17), the most recent CVEs mentioning TLS are CVE-2023-5554 ("lack of TLS certificate verification in log transmission of a financial module within LINE Client for iOS prior to 13.16.0") for the popular LINE messaging app; CVE-2023-5422 ("the functions to fetch e-mail via POP3 or IMAP as well as sending e-mail via SMTP use OpenSSL for static SSL or TLS based communication. As the `SSL_get_verify_result()` function is not used ...") for the OTRS service-management suite; CVE-2023-4807, an OpenSSL bug; CVE-2023-4586 ("the Hot Rod client does not enable hostname validation when using TLS") regarding the Hot Rod data-access protocol; CVE-2023-45199, a bug in another TLS library, Mbed TLS; CVE-2023-4420, regarding TLS not being used in the SICK LMS5xx laser sensors; CVE-2023-43615, another Mbed TLS bug; and CVE-2023-4331 (from 2023-08-15), saying the the Broadcom RAID Controller web interface has an insecure default TLS configuration.

As these CVEs illustrate, deploying TLS requires integrating TLS into a wide range of protocols and an even wider range of applications. Similar comments apply to other options for cryptography at the transport layer, such as DTLS and QUIC. There have been heroic efforts to expand the use of transport-layer cryptography and in particular of TLS, but this is a very large programming project—as reflected by the breadth of TLS-related security failures—and is still far from complete.

According to Mozilla's Firefox Telemetry [8], the percent of all web-page loads using TLS has increased from around 25% in 2014 to around 80% in 2020, where it has stayed roughly since. One cannot tell from these numbers how many of the remaining 20% are using web-server software that, apparently like the software in CVE-2023-4420, do not support TLS at all, and how many are devices where TLS is supported but, despite the availability of Let's Encrypt, still not configured. A more direct view of the scale of the software problem is a GitHub search for "SSL", which currently finds 4 million commits; a skim of the most recent 50 found that all 50 were in fact referring to SSL rather than something else by the same name.

Everything then has to change again for post-quantum cryptography, and this needs to happen *as soon as possible*. Deployment speed matters. Large-scale attackers have been recording Internet traffic for years in the hopes of decrypt-

ing it later. Plenty of data that is encrypted today will still be interesting to attackers armed with future quantum computers.

There have been efforts to develop and standardize post-quantum cryptographic primitives, and to incorporate post-quantum primitives into both new and existing cryptographic protocols, including TLS; see, e.g., [3, 10, 21]. Chrome 116, released 2023-08-15, automatically encrypts using an experimental post-quantum X25519Kyber768 TLS option whenever the server supports that option. Supporting post-quantum encryption on web servers is then "simply" a matter of upgrading every popular TLS library, checking all web-server software using those libraries to fix any incompatibilities with the new post-quantum options (such as overly narrow lists of TLS cipher suites), and adding TLS support to web-server software that doesn't have it already. Everything then has to be repeated for clients and servers for SMTP, IMAP, Hot Rod, and a very long list of further application-layer protocols. This is a clear path, but also a slow, labor-intensive path.

1.1 End-to-End Post-Quantum Cryptography Without Touching the Applications

To bypass the deployment bottleneck described above, this paper's PQConnect introduces end-to-end post-quantum cryptography as an application-independent "bump in the wire" at the *network* layer of the network stack, without modifying the *transport* layer.

PQConnect automatically creates post-quantum network tunnels that encrypt entire packets between each client device and each server device. Packets generated by higher-level protocols running on top of TCP/IP or UDP/IP are intercepted by PQConnect, encrypted with post-quantum cryptography, and delivered to the other end, where they are decrypted by PQConnect and given back to the higher-level protocols.

From a software-engineering perspective, the critical feature of PQConnect is that it does not touch the applications that it is protecting. For example, PQConnect adds post-quantum cryptography as a wrapper protecting an SMTP connection with no changes to the SMTP client software, no changes to the SMTP server software, and no changes to SMTP: the SMTP packets, like all other packets between the client and the server, and transparently routed through a PQConnect tunnel. The packets are protected today against a future quantum adversary, even if the packets originally had just pre-quantum cryptography or no cryptography at all.

VPNs have the same software-engineering benefit. Typically a VPN is configured on a client device to route all outgoing traffic through an encrypted tunnel from the client device to a proxy specified as part of the VPN configuration. More complicated configurations are possible, such as routing traffic via a corporate proxy if the traffic's outgoing IP address is within the corporate IP range. Some VPNs have been adding support for post-quantum cryptography; notable examples include Mullvad [24], the new Rosenpass [23], and VPNs based on OpenSSH, which has been using Streamlined NTRU Prime by default [16] since 2022.

The critical difference is that PQConnect *automatically* creates an end-to-end tunnel from the client to any PQConnect server that the client is connecting to. VPNs, with their typical configurations, protect traffic from the client device to a proxy but not all the way to the server: they do nothing to protect against attackers controlling the proxy or controlling the network between the proxy and the server.

Users sometimes add specific servers to VPN configurations so that VPNs create tunnels all the way to those servers. PQConnect automates the creation of post-quantum tunnels, eliminating the need for any server-specific configuration on the client. Configuring a PQConnect server means creating a long-term post-quantum public key for that server and publishing an announcement saying that the server name supports PQConnect. A client device, with PQConnect installed and running, notices whenever it is connecting to a server name that has such an announcement; it then creates a PQConnect tunnel to the server, and routes subsequent traffic for that name through that tunnel. See Section 3 for further explanation of PQConnect's automatic peer discovery.

We have implemented PQConnect for Linux. Our PQConnect implementation is attached to this paper (as a PDF attachment).

1.2 Reasons to Pursue Two Paths

The current state of encryption on the Internet still contains large holes, especially considering the threat of quantum computers. Our PQConnect software is new (see attached and [1]), so obviously it has done nothing yet to plug these holes—but it provides another clear path to broad deployment of end-to-end post-quantum cryptography, and an inherently easier path than TLS.

There are many different TLS libraries supporting the integration of TLS into applications in different environments. This profusion of software puts heavy weight on backwards compatibility, slowing down the evolution of TLS. Broken algorithms have remained in the TLS standards long after attacks were published. Some of the first attacks against RC4, for example, were published already in 2001 (see [9, 12]), but RC4 was not completely removed from TLS until version 1.3 in 2018 [19], 17 years later.

We emphasize that PQConnect is not in a race against TLS; rather, PQConnect and TLS are jointly racing against attackers. Application designers are taking important steps in extending the use of TLS and other transport-layer cryptography. PQConnect is not an alternative to TLS for that—it does not plug into applications or into application-layer protocols. PQConnect instead works at a different layer, as something to be installed by host administrators. Pursuing two parallel approaches to the deployment of post-quantum cryptography means that each device is protected as soon as one of the

approaches has covered that device. The sooner this happens, the less data is exposed to future quantum attacks.

TLS works transparently on top of PQConnect when both of them are deployed on the same device. One should not think of these two end-to-end security layers as redundant:

- The PQConnect approach of protecting all applications at one stroke relies on not touching application software, but some applications *want* to interact with the security layer—for example, giving HTTPS special treatment not given to HTTP—and already know how to interact with TLS.
- There have been many security issues in TLS implementations, and sometimes in TLS itself. If new post-quantum cryptography in TLS turns out to have security problems, a second layer of defense could still stop attacks, especially when the second layer is using different cryptosystems.
- PQConnect makes a particularly conservative choice of post-quantum cryptosystem, namely the 1978 McEliece cryptosystem at a very high security level, for the server's long-term public key. This KEM is the foundation of security for PQConnect's packet encryption, packet authentication, and server identification.
- PQConnect provides a stronger notion of forward secrecy than TLS does: PQConnect uses time-based key erasure within a session, ensuring that within minutes it is unable to retroactively decrypt previous data—although, for performance reasons, this relies on lattice-based cryptography rather than the McEliece system.
- PQConnect also encrypts more information than TLS does, such as IP packet headers, although attackers can deduce some of that information via traffic analysis.

Given the low median age and high fatality rate of proposed post-quantum cryptosystems, there is broad (although not universal) agreement that post-quantum cryptography should be rolled out only as "hybrid cryptography" on top of a conventional layer of security, typically the X25519 pre-quantum ECDH system. The McEliece cryptosystem is older than ECDH, but, to avoid complicating a simple recommendation of always using hybrid cryptography, PQConnect uses X25519 here too. PQConnect tunnels thus establish a shared key with a combination of X25519 and the McEliece system for long-term security, and a combination of X25519 and lattice-based cryptography for fast key erasure. Security of the PQConnect handshake has been symbolically proven using TAMARIN Prover; see Section 5.

2 Threat Model

PQConnect aims to provide confidentiality and authenticity of all packets between clients and servers against attackers who have access to a large quantum computer, can store large amounts of network traffic for future cryptanalysis, and can insert, drop, and modify packets on the network.

What is most urgent today is encrypting data stored today for decryption by future quantum computers, but PQConnect is also designed so that its authentication will not need a subsequent post-quantum upgrade.

We also assume that the attacker can gain future physical access to a peer device. We want to ensure that such an attacker cannot use this access to decrypt traffic that was sent more than a few minutes earlier.

We assume attackers can also alter the system clock of any peer, for example by forging NTP packets. NTP is unencrypted and unauthenticated by default; also, even if NTP is run over PQConnect or an NTP-specific security protocol, we do not want to assume security of NTP servers. An attacker-controlled system clock should not affect the rapid erasure of private decryption keys on any host.

As a lower priority, we consider attackers who want to compromise availability of services. It is important to note that a powerful-enough network attacker can always affect availability of services.

3 Data Flow for PQConnect Integration

This section explains PQConnect's application integration: how a PQConnect client—specifically, our PQConnect implementation for Linux—recognizes PQConnect servers and arranges for applications on the same machine to send traffic through the PQConnect tunnel, *without* changes to the application software. Later sections of the paper describe the cryptographic details of the tunnel.

The closest previous integration work that we are aware of is MinimaLT [18], which automatically creates end-to-end pre-quantum tunnels covering all application traffic. The applications in [18] were written for a new network API designed from the outset to use these tunnels, whereas in this section the applications are unmodified Linux programs unaware of PQConnect.

3.1 High-Level Data Flow

The administrator of a PQConnect client device installs and runs the PQConnect client software.

This software automatically recognizes when application software on that device—for concreteness, imagine an XMPP client—is connecting to a server that supports PQConnect. Section 3.2 explains how.

This software then creates a PQConnect tunnel to that server, if it doesn't have a recently used tunnel to that server.

This software then captures packets from the application, and routes those packets through that tunnel. Section 3.3 explains how.

3.2 Server Identification

A web browser that sees an `https` URL knows that it has to use TLS for that URL. An SMTP client that sees an SMTP server saying `STARTTLS` in response to `EHLO` knows that it is allowed to issue a `STARTTLS` command to upgrade the connection to TLS. PQConnect is in a different situation: it does not have application-specific indicators such as `https` or `STARTTLS`.

What a PQConnect client does see—without pestering non-PQConnect servers with extra questions—is a DNS response with information configured by the server administrator, such as `www.google.com A 216.58.214.4` indicating that `www.google.com` has IP address 216.58.214.4. Sometimes this information is a multi-part response: for example, `www.amazon.com CNAME g4hukkh62yn.cloudfront.net` indicating that `www.amazon.com` has a canonical name of `g4hukkh62yn.cloudfront.net`, followed by `g4hukkh62yn.cloudfront.net A 18.239.34.131` indicating the server's address.

PQConnect reuses the idea from DNSCurve of inserting cryptographic announcements into server names that are naturally returned to clients. For example, a client looking up the address for the server `www.pqconnect.net` receives a `CNAME` pointing to `pq16f7p57wspt3272chysg9fgjctl rtzzkyhy4ntdm0vnl6grtvylj0.pqconnect.net` and an `A` pointing this `pq1...lj0.pqconnect.net` name to the actual server address.

A non-PQConnect client will connect as usual to the server. A PQConnect client sees the name component consisting of `pq1` followed by 52 symbols from the DNSCurve alphabet `0123456789bcdfghjklmnpqrstuvwxyz`, and treats this as saying (1) that the server supports PQConnect and (2) that those 52 symbols are the hash of the server's long-term public key. The DNSCurve alphabet is designed to minimize the risk of accidental collisions, and PQConnect's `pq1` is separated[1] from DNSCurve's `uz5`.

Instead of, or as a supplement to, distributing the short name `www.pqconnect.net`, server administrators can distribute the name `pq16f7...lj0.pqconnect.net` shown above. This is harder to read and needs to be updated if the server's long-term public key changes, but provides stronger security; see Section 3.4 below.

3.3 Capturing Application Traffic

The PQConnect client software goes beyond inspecting the DNS packet: it also rewrites the packet, to replace the server's IP address with an address assigned by PQConnect within a local address space.

[1]The `1` in `pq1` is intended as a master version number, allowing for the possibility of a structured PQConnect protocol upgrade in which clients are first required to add multi-protocol support for `pq1`/`pq2` by a specified date, and then servers are allowed to switch to `pq2` and are required to do so by a specified date, and finally clients disable `pq1`.

Currently our software uses `netfilter` to capture and rewrite packets. There are many other options here, such as using `/etc/resolv.conf` to route DNS queries through a PQConnect DNS proxy, or intercepting the `systemd` resolver. Firefox automatically uses DNS over HTTPS in some cases, but a DNS proxy (or rewriting) can disable this by creating an IP address for `use-application-dns.net` (and can still pass DNS queries locally to a modular DNS-over-HTTPS client if desired). A user *manually* configuring Firefox to use DNS over HTTPS will prevent Firefox from using PQConnect.

Normally the application software ends up seeing the PQConnect-assigned address rather than the server's actual IP address, and ends up sending packets to that address. The PQConnect client receives those packets and—after a tunnel is set up using the server's long-term public key—tunnels those packets to the server. It also receives packets back through the tunnel, and delivers decrypted packets back to the application.

The PQConnect client software keeps track of a mapping of server-key hashes to local IP addresses and tunnels. If the same server-key hash shows up again (from the same application or another application), the software replaces the server's IP address with the same local IP address, reusing the tunnel.

It would be possible to use the server's IP address as a further input for this mapping. Often DNS is configured to announce multiple IP addresses for a server name, sometimes to spread load across multiple machines and sometimes because the same machine is reachable through multiple networks. This is compatible with PQConnect if all of the machines are configured with the same public key (and know the corresponding private key), but it is not clear that converting each address into a separate tunnel is better than reusing a single tunnel for the server name.

Rather than rewriting the server's IP address as a local IP address, we could capture all traffic to the server's IP address. One reason to use local IP addresses is to support the following deployment possibility: multiple virtual machines handle different server ports on a public IP address, and one of the VMs starts supporting PQConnect for traffic to its ports, with a key hash announced via a name dedicated to that VM, while the others do not support PQConnect or have their own PQConnect keys. On the other hand, it is not clear how useful this possibility is compared to the host administrator setting up PQConnect to cover the whole machine, with all of the VMs free to announce the host's key hash.

3.4 DNS Security Analysis

A name in DNS such as `www.tiktok.com` is controlled not just by TikTok but also by the `.com` servers and the root servers. Furthermore, because DNS traffic has no cryptographic protection by default, the name is also controlled by any attacker putting in the effort necessary to forge pack-

ets. Our threat model includes attackers controlling network routers near the legitimate DNS servers. It also includes attackers using NSA's QUANTUMINSERT man-on-the-side attack technique (no relation to quantum computing), which injects fake responses to DNS queries before the real responses arrive; the real responses are then ignored.

There has been some deployment of cryptographic addons to DNS, such as DNSSEC, DNSCurve, DNS over TLS, and DNS over HTTPS. Today these techniques are using prequantum cryptography—for example, DNSSEC keys are usually ECDSA or RSA keys, both of which a quantum attacker can break in advance to generate forged DNSSEC-signed responses on the fly—and it is not clear that they will be upgraded before attackers have quantum computers. A bigger problem today is that these techniques are far from universally deployed.

The large gaps in DNS security pose problems for protocols whose security relies on DNS. In particular, these gaps pose security problems for TLS, at least the way that TLS is normally deployed, trusting the usual X.509 PKI. Control over DNS for a server name suffices to obtain a certificate for that server name and man-in-the-middle control over TLS connections to that name; this is true even if the long list of certificate authorities is narrowed to just Let's Encrypt. Let's Encrypt tries multiple DNS queries, but this obviously does not achieve security in the threat model considered in this paper. Let's Encrypt also publicly logs all of the certificates that it issues, making attacks more likely to be detected, but detection is not the same as security.

For PQConnect, it is more obvious how security relies on the security of DNS, since PQConnect simply looks up a server name in DNS. An attacker forging DNS packets can man-in-the-middle a PQConnect connection by returning a forged CNAME/A record with a different PQConnect key hash and a different A record, or simply stripping away PQConnect by removing the CNAME in favor of a different A record, in both cases pointing the client to an attacker-controlled machine. This takes fewer forged packets than man-in-the-middling a TLS connection.

PQConnect supports two techniques for improving security here. The first technique is running DNS itself over PQConnect, by deploying PQConnect on machines that run DNS clients, DNS resolvers, and DNS servers, so as to protect all applications of DNS (not just to protect PQConnect itself). More deployment of DNS over PQConnect means fewer points in the network that are available to an attacker to insert malicious response packets. Deployment of PQConnect at every level up to the root servers would eliminate network substitution of DNS packets, even by future quantum attackers, although `www.tiktok.com` would still be controlled by the `.com` servers and the root servers.

The second technique is taking whatever previous channel is used to distribute a DNS-trusting name such as `www.pqconnect.net`, such as a link on a web page, and using the same channel to instead distribute a PQConnect name such as `pq16f7...lj0.pqconnect.net`. Attackers subsequently forging DNS packets can still deny service but cannot remove or modify the already-distributed key hash. Of course, one also needs to make sure that the previous channel is secure.

3.5 Rebinding Analysis

There are four basic layers in a PQConnect DNS response: the original name such as `www.pqconnect.net`, the PQConnect name such as `pq16f7...lj0.pqconnect.net`, the key hash inside that name, and the server's IP address. The first layer disappears if the application is starting with the PQConnect name.

To the extent that DNS is secure (see Section 3.4), the attacker cannot forge a DNS response that matches the original name without matching all of the other data. However, the attacker is still free to send a DNS response that, e.g., maps another name to the same PQConnect name, or to a name having the same key hash, or to a name having the same address.

The case of none of these colliding (000) is uninteresting. The PQConnect name colliding without the key hash colliding (100, 101) is impossible since the PQConnect client computes the key hash from the PQConnect name. The PQConnect name colliding without the IP address colliding (110) would be another DNS security failure. This leaves four interesting possibilities.

We see two possibilities as typical deployment scenarios: pointing another name to the same PQConnect name (111), or to a different PQConnect name with the same key hash and the same IP address (011). Both of these end up creating a tunnel to the IP address; PQConnect does not care what the original name was.

Having a different name and different key hash pointing to the same IP address (001) is not obviously typical but is a potential deployment possibility noted above. This does not cause any confusion for PQConnect: tunnels are indexed by key hash rather than by IP address.

This leaves one attack possibility (beyond DNS attacks): pointing another name to a different PQConnect name with the same key hash but a different IP address (010). In this case, PQConnect will try setting up a tunnel to that IP address, and will fail, since it checks the key exchange against the key hash, as described later in the paper. PQConnect will still point clients to the local IP address for the non-functional tunnel until the tunnel times out, so there is a denial-of-service attack. If we indexed tunnels by key hash *and* IP address then this denial-of-service attack would disappear, although this interacts with a usability question noted above.

4 Protocol Specification

4.1 Cryptographic Notation

The full list of notation, variables, and functions used in PQConnect is as follows:

- $a \leftarrow b$ means assigning the value of b to the variable a. $a \xleftarrow{\$} S$ indicates randomly choosing an element s from the set S and assigning its value to a. Sometimes $:=$ is used synonymously with $\leftarrow$.
- $\mathbf{spk_p^X}, \mathbf{ssk_p^X}, \mathbf{epk_p^X}, \mathbf{esk_p^X}$ are static public, static private, ephemeral public, and ephemeral private keys for public key cryptosystem/KEM **X** and peer **p**.
- **X.keygen()** Generates a random (public, private) keypair for the public key cryptosystem **X**.
- **X.Encap(pk)** Generates a random 32-byte key **k** and its encapsulation **c** under the public key **pk**.
- **X.Decap(sk, c)** Takes a private key **sk** and ciphertext **c** and outputs the encapsulated key **k** if **c** is a valid encapsulation. Otherwise outputs $\perp$.
- **ChaCha20(k, n, ic, m)** Generates the ChaCha20 encryption of message **m** under 32-byte key **k**, 4-byte initial counter **ic** and 12-byte nonce **n**. **ic** and **n** are represented as little-endian integers. The initial counter is usually assumed to be 0, in which case we omit it and write **ChaCha20(k, n, m)**
- **AEAD.Enc(k, n, m, ad)** Generates the ChaCha20-Poly1305 authenticated encryption of message **m** and associated data **ad** under key **k** and nonce **n** (as specified in [15]). The output is a ciphertext **c** and 16-byte authentication tag **t**. $\mathbf{c}^*$ denotes the concatenation of $\mathbf{c}||\mathbf{t}$.
- **AEAD.Dec(k, n, $\mathbf{c}^*$, ad)** Decrypts and verifies an authenticated ciphertext $\mathbf{c}^*$ under key **k**, nonce **n**, and with associated data **ad**. Successful decryption outputs a message **m** of length $|\mathbf{c}^*| - 16$. Failure outputs $\perp$.
- **Hash(m)** Generates a 32-byte SHAKE256 digest of the message **m**.
- **HMAC(k, m)** Generate a HMAC-SHA256 on message **m** under key **k**.
- **$\mathbf{KDF}_n$(k, i)** Generates **n** 32-byte keys from key **k** and optional input **i**. PQConnect uses ChaCha20as the underlying pseudo-random-function (PRF) for the KDF and is described in detail in Section A.
- $\mathbf{C_p}, \mathbf{H_p}$ are the CipherState and HandshakeState objects for peer **p**. During the handshake each peer p maintains these two state variables.
- $\mathbf{T_p}$ is the root sending key for peer **p**. Thus $\mathtt{T_C}$ is the root sending key for the client (and the root receiving key for the Responder).
- **tunnelID** This is a 32-byte pseudo-random value that uniquely identifies a tunnel.

4.2 Key Distribution

If a client has never seen a particular server before, it needs to obtain the server's long term and ephemeral public keys. PQConnect servers distribute their public keys via a keyserver. When a client obtains a PQConnect server's public key hash from DNS, it needs two more pieces of information: 1) the IP and port number the keyserver, and 2) the PQConnect listening port number for the PQConnect server. Both of these pieces of information are published in additional DNS TXT records, which the client queries at the start of a connection with a new server. For the example of `pq1...j0.pqconnect.net`, PQConnect listening port and keyserver address are published as `pq1...j0.pqconnect.net TXT "p=42424"` and `ks.pq1...j0.pqconnect.net TXT "ip=131.155.71.58; p=42425"`, respectively.

Due to the large size of mceliece6960119 keys, the public keys are broken into packet-sized chunks, each of which can be requested individually. To allow for instant verification that a key packet is authentic, the server's public keys are distributed as a Merkle tree, the root of which is the published key hash. The process for requesting and verifying the server's long term keys is described in Section 4.3

Once the long term keys are verified, the client requests short-lived sntrup4591761 and X25519 ephemeral keys from the keyserver to compute a handshake message against them. The public keys have an issuing period of 30 seconds, and the private keys are erased after 120 seconds. That is, a pair of public keys distributed to a client at time t_0 may be also issued at $t_0 + 29$, but it will not be issued at $t_0 + 30$, and any ciphertext created with that key will be unrecoverable by the server at time $t > t_0 + 120$.

After a tunnel is established, PQConnect clients cache servers' long-term X25519 keys and cache pre-computed ciphertexts against the mceliece6960119 key. This allows future handshake messages to be sent quickly without having to transmit or store a large mceliece6960119 public key for each server. However, once a client has used up their cache of McEliece ciphertexts, they will need to re-request the public key from the server.

4.3 Streaming Verification of Long-term Keys

To allow for instant key packet verification, PQConnect constructs and transmits long term keys as a Merkle tree, with packet-sized chunks of the public keys stored in the leaves,

and the published hash of the long-term public keys in the root. Each internal node is at most 1152 bytes, which is equal to the length of 36 concatenated 32-byte hashes, and is small enough to fit into a UDP packet without fragmentation.

Clients request key packets at increasing depths of the Merkle tree and verify the packet's authenticity by comparing its hash to the appropriate segment of its parent node. Any packet whose hash does not match its parent can be immediately discarded and re-requested.

The procedure for distributing and verifying PQConnect public keys is depicted in Figure 1. The mceliece6960119 public key is divided into 910 parts, each 1152 bytes (except for the last one). The 32-byte long-term X25519 public key is concatenated to the end of the last part (which becomes 183 bytes in total). These parts form the leaves of the tree (at depth 3). Each of the 910 leaf nodes is hashed, and the hashes are concatenated and then again divided into 26 depth 2 nodes, each of which is 1152 bytes (except for the last one). The hashes of all 26 depth 2 nodes are then concatenated to form a single depth 1 node of 832 bytes. The depth 1 node is then finally hashed to obtain the 32-byte public hash provided by DNS. The full tree is thus a {36,26,1} -ary Merkle tree.

To obtain and verify the server's long term public keys, the client first requests the depth 0 and depth 1 packets and checks that the $H_{0,0}$ the hash obtained via DNS. Then it checks that the depth 1 node's hash matches the hash $H_{0,0}$ which was obtained via DNS. They then request and verify the 26 depth 2 packets. Finally they request and verify the 910 leaf packets, which comprise the public keys. Requests at each level can occur in parallel, so the entire process can be performed in three RTT. Each round trip is indicated to the left of the tree.

4.4 The PQConnect Handshake

Once the client has obtained and verified the server's long-term keys, they can proceed to establish a tunnel by sending a handshake message to the server. The PQConnect handshake protocol is performed in 0-RTT, meaning the client can send encrypted, tunneled packets to the server immediately after sending the handshake message.

To hedge against future attacks on post-quantum schemes, PQConnect's handshake protocol uses two different post-quantum KEMs along with Curve25519 in a nested hybrid pre- and post-quantum handshake protocol. "Nested" here means that public key operations are performed sequentially, and each subsequent (inner) operation is cryptographically protected by a secret key derived from all previous (outer) operations.

Because the final shared secret depends on all the public key schemes (both pre- and post-quantum), even if an attacker does successfully break the post-quantum schemes, the combined system will at least not have *less* security than if ECDH alone were used.

4.5 Nesting of Cryptographic Primitives

Nesting offers the benefit that an attacker must work sequentially rather than in parallel to recover the final handshake key. Nesting also provides a small mitigation against wasted CPU cycles from invalid handshake messages: Servers don't need to perform public key operations for inner layers once an operation in the outer layer fails.

There are several options one has for this ordering, each with its own benefits and costs. For example, using X25519 keys in the outermost layer may be attractive if the protocol is trying to extend an existing pre-quantum handshake that uses X25519 ECDH key agreement and needs to match its specification. Additionally, computing a shared X25519 secret has a median cycle count of around 106,000 cycles on a 2022 Intel Core i3-1215U, while decapsulating a mceliece6960119 ciphertext has a median cycle count of 269665 cycles on the same machine [5]. Using X25519 as the first layer may therefore waste fewer cycles if an invalid handshake packet is received. On the other hand, it also would potentially expose a pre-quantum algorithm to an attacker with a quantum computer, giving away the outermost layer for free.

Even among pre-quantum attackers, it is also good to consider attackers that can observe many handshake messages between different peers and may be happy to break a subset of them. For an attacker who wants to break the confidentiality of many servers, putting X25519 in the outer layer may allow the attacker to peel back this layer on more handshakes than if we used mceliece6960119 keys instead. This is because once an attacker has broken a single X25519 key, the cost of breaking t more keys is an additional multiplicative factor of only $(O\sqrt{t})$ [11]. This means, for example, that once the attacker has successfully broken one server's long-term X25519 key, breaking the next 10,000 X25519 keys it sees is only 100 times as much work. Furthermore, solving the discrete log problem once means that *all* connections to that server are now compromised.

By contrast, the best attacks against Classic McEliece recover an encapsulated secret key from a ciphertext, not the private key corresponding to the public key under which the ciphertext was generated. This means attackers are attacking individual sessions, not long-term keys. An attacker who has successfully decrypted one out of N McEliece ciphertext may have reduced attack complexity when attacking subsequent ones, but they do not immediately get all plaintexts encapsulated to the same public key for free.[2].

Taking the above considerations into account, the nesting order for PQConnect, from outermost to innermost, is Classic McEliece (long-term) → Curve25519 (long-term–ephemeral) → X25519 (ephemeral–ephemeral) → Streamlined NTRU prime (ephemeral). This places the oldest and most confidently quantum-resistant primitive as the first line

[2]For more on DOOM (decoding one out of many) attacks on McEliece, see [22]

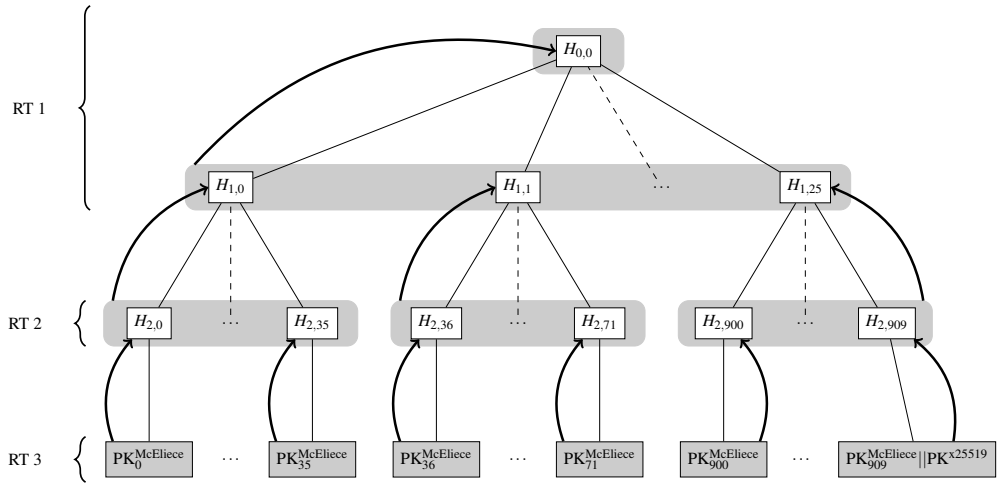

Figure 1: The structure and verification process for the PQConnect public key Merkle tree. Nodes of the tree are shown as rectangles. Leaf nodes are 1152 bytes, except for the last one, which is 183 bytes. Internal nodes are 32-byte hashes H indexed by tree depth and position within their given depth. Their values are equal to the hash of the total concatenated bytes of their children. The shaded regions indicate how children of internal nodes are grouped into packets, and so each shaded region represents a single packet. Arrows pointing upwards from packets to internal nodes represent verifying a packet by hashing its contents and comparing this hash with its parent.

of defense. ECDH comprises the middle two layers for robust pre-quantum security. The newest KEM forms the last line of defense in the handshake. This ordering forces a quantum attacker to successfully recover the secret McEliece key before even learning the client's ephemeral X25519 key. Without knowing this, even a successful discrete log computation for the server's X25519 public keys is of little use.

4.6 The 0-RTT Handshake Protocol

In this section we describe the handshake in detail. Some inspiration for the handshake comes from the Noise Protocol Framework by Perrin [17], such as the use of CipherState and HandshakeState variables. However, all the patterns in Noise are 1-RTT instead of 0-RTT. Being able to make connections in 0-RTT means that clients who already have a server's public keys can send tunneled packets immediately with the handshake message, which reduces latency. On the other hand, they are vulnerable to replay attacks, since all of the live randomness comes from the client. We discuss mitigations against replay attacks in Section 4.7.

During the handshake, a secret is created using each of the server's four public keys, and these are incorporated into the CipherState variable as they are derived, with the original secrets being immediately erased. Every publicly transmitted value is also incorporated into the HandshakeState variable after it is created. In the following description, updates to the state variables are omitted but are shown in detail in Figure 2.

The client first generates a random 32-byte ephemeral X25519 key and its corresponding 32-byte public key. They then encapsulate a random key $\mathtt{k}_0$ to the server's long term mceliece6960119 public key as $\mathtt{c}_0$. The client encrypts their ephemeral X25519 public key $\mathtt{epk}_\mathtt{c}^{\mathtt{x25519}}$ under $\mathtt{C}_\mathtt{c} = \mathtt{k}_0$, nonce 0, and associated data $\mathtt{H}_\mathtt{c}$, producing $\mathtt{c}_1^*$. Next, they compute shared ECDH keys $\mathtt{k}_1$ and $\mathtt{k}_2$ using the server's static and ephemeral X25519 public keys, respectively. Finally, the client encapsulates $\mathtt{k}_3$ under the server's sntrup4591761 public key, generating $\mathtt{c}_2$, and then encrypts this under the updated $\mathtt{C}_\mathtt{c}$ and $\mathtt{H}_\mathtt{c}$ values and nonce 0 to produce $\mathtt{c}_3^*$. The $\mathtt{tunnelID}, \mathtt{T}_\mathtt{c}$ and $\mathtt{T}_\mathtt{s}$ shared secrets are then computed using the CipherState and HandshakeState variables as inputs to the KDF. The client sends a 2-bytes "initiation msg" prefix `0x1 0x0`, $\mathtt{c}_0, \mathtt{c}_1^*$, and $\mathtt{c}_3^*$ to the server. The total length of this message is 1307 bytes: 2 bytes for message type (`0x01` as a 16-bit little-endian integer), 194 bytes for the mceliece6960119 ciphertext $\mathtt{c}_0$, 48 bytes for $\mathtt{c}_1^*$ (32 bytes for the key and 16 bytes for the authentication tag), and 1063 bytes for $\mathtt{c}_3^*$ (1047 byte sntrup4591761 ciphertext + 16 byte authentication tag).

Upon receipt of the client's message, the server checks the message type, decapsulates $\mathtt{c}_0$ to obtain $\mathtt{k}_0$, decrypts $\mathtt{c}_1^*$ to obtain $\mathtt{epk}_\mathtt{c}^{\mathtt{x25519}}$, computes the two ECDH keys $\mathtt{k}_1$ and $\mathtt{k}_2$, and finally decrypts $\mathtt{c}_3^*$ and decapsulates $\mathtt{c}_2$ to obtain k_3. They can then obtain the same shared values $\mathtt{tunnelID}$, $\mathtt{T}_\mathtt{c}$, and $\mathtt{T}_\mathtt{s}$.

For each AEAD operation, the HandshakeState variable is used as authenticated data, ensuring that the handshake only succeeds if both participants have an identical view of the handshake transcript. Except for the first encapsulated secret key, all message fields containing data are authenticated and encrypted using ChaCha20-Poly1305. At the conclusion of the handshake, both parties erase all remaining non-public

values aside from the `tunnelID`, T_c, and T_s.

4.7 Replay mitigations

By virtue of being 0-RTT, client handshake messages are replayable, since the server cannot enforce freshness. Processing the same handshake message twice will result in the same `tunnelID` and session keys. This raises two concerns. The first is that at the protocol level. Replaying a handshake message that the server has already seen should not affect an existing tunnel with a client.

The second is at the cryptographic level. If the server receives two identical handshake messages from different IPs and creates identical tunnels with both of them, an attacker may force the server to reuse packet keys to send different packet data, breaking the security of the symmetric cryptography.

PQConnect servers mitigate the effects of handshake replays in two ways. The first is a byproduct of the way keys are erased. Because ephemeral keys are short lived, the server will only be able to process handshake messages no older than 120 seconds. Any handshake message older than 120s was made using public keys whose corresponding private keys have been deleted, and is thus unusable by the server to create a new tunnel.

To prevent replay attacks for fresh handshake messages, the server keeps an updated list of McEliece ciphertexts used in successful handshake messages within the last two minutes. When ephemeral keys are erased, the list removes corresponding old McEliece ciphertexts from this list. This allows servers to quickly discard replays of still-valid handshake messages.

4.8 Denial-of-Service mitigation

PQConnect aims to provide strong cryptographic security while being as transparent as possible to users. That means availability should not be affected by PQConnect. Unfortunately, PQConnect's use case makes it particularly vulnerable to DoS. Contrasted with a closed network like a traditional VPN, servers' public keys are actually public, so a Wireguard-like MAC proving knowledge of the server's public key is a bar that everyone with an internet connection can clear. PQConnect is also UDP-based, making it easy for attackers to spoof source IP addresses. Most notably, the public key operations performed by the server during a handshake are expensive. Public key operations in general require many more cycles to perform than symmetric operations, and for every (successful) handshake, PQConnect performs four: two post-quantum KEM decapsulations and two Elliptic Curve Diffie-Hellman scalar multiplications.

We want to prevent an attacker wishing to exhaust a server's resources from doing so by flooding the server with bogus handshake messages, forcing the server to perform pointless McEliece decapsulations.

We also do not want any mitigations to flooding attacks to be useful in reflection or amplification attacks. That is, any response sent to clients by the server should not be larger than the data sent to the server itself.

Finally, we do not wish to keep state relating to who has been issued a challenge, especially if doing so scales linearly with the number of (real or spoofed) distinct IP addresses. Only honest users should be able to affect the state.

If a server is under load, or when it receives an invalid handshake from a client, it can send a `Handshake fail` message to the client with a challenge. The challenge consists of a MAC over a timestamp value, the client's IP address and port, and a difficulty level `hardness_bits`, generated using a secret MAC key. The timestamp indicates freshness, while the ip address and port binds the Mac to the client.

```
mac_chall := HMAC(cookiekey, (IP_client||
    port_client || timestamp || hardness_bits)
```

When a client receives a challenge MAC they compute a solution `mac_prefix` consisting of a byte string that, when prefixed to the MAC value and authenticated using the MAC as the key, produces a new MAC with a leading number of zero bits equal to the difficulty level. They then resend their handshake message along with the original timestamp, hardness, original MAC, and solution.

```
mac_solution := (hardness_bits, timestamp,
    mac_chall, mac_prefix)
```

When the server receives `mac_solution` it checks that the timestamp is fresh, that

```
HMAC(mac_chall,mac_prefix||mac_chall)
```

begins with $0^{\texttt{hardness\_bits}}$, and that `mac_chall` is a valid mac under its secret MAC key. If the challenge verifies, the server will process the accompanying handshake message.

4.9 The PQConnect Key Ratchet

PQConnect tunnels encrypt each packet with a one-time key that the *sender* immediately erases after using it. The more subtle question is how long a *receiver* keeps a packet key. Packets are delayed in the network and can arrive out of order, or not at all.

Our threat model includes an attacker that can prevent a recipient from receiving packets, and thus erasing decryption keys. It also includes an attacker that later compromises the recipient's machine, obtaining whatever keys have *not* been erased yet.

To limit the potential damage of long-term key storage, PQConnect sets a built-in time limit of two minutes for erasing each one-time key. The sender stops using the key after just 30 seconds, so that packets encrypted under the key are still

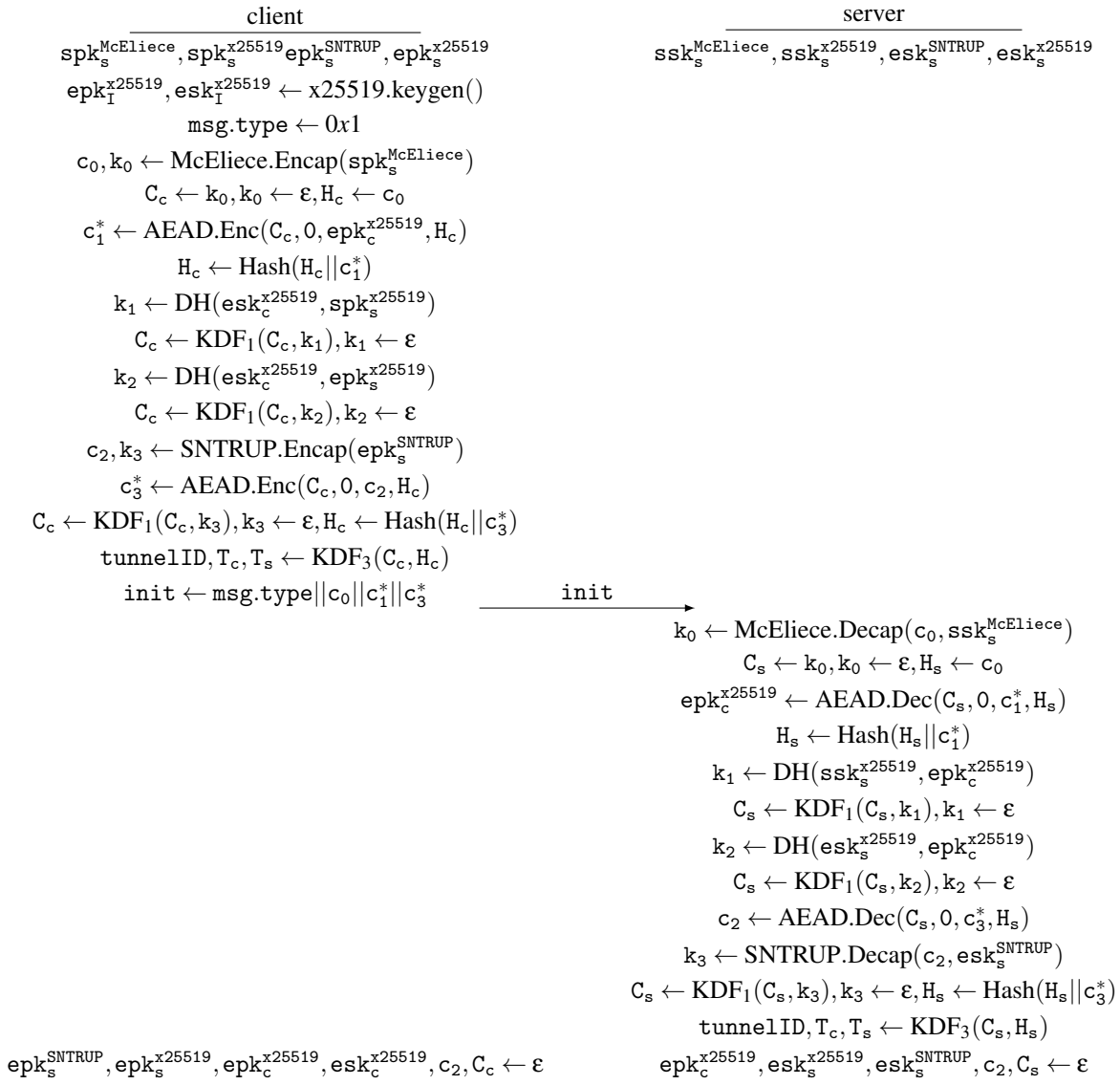

Figure 2: 0-RTT PQConnect Handshake

decrypted correctly by the receiver even if there are network delays as long as 90 seconds.

To obtain many one-time keys from an initial shared secret, PQConnect uses the standard idea of re-keying encrypted communication using a KDF (see, e.g. [2]). PQConnect's ratchet, depicted in Figure 3, is designed to be able to handle large packet volumes including out-of-order and delayed packets; compared to Signal's symmetric-key ratchet [13], there is an extra dimension in Figure 3.

The starting point for the ratchet is as follows. Once both peers have completed the handshake, they are left with three shared 32-byte-long values: `tunnelID`, T_C, and T_S. To limit the coordination required between the two sides of the tunnel, packets are encrypted using two chains of keys, one for client encryption (and server decryption), and one for server encryption (and client decryption). T_C is the root of the client's sending chain, and T_S is the root of the server's sending chain. Figure 3 depicts one of these two chains. The main vertical dimension shows 30-second epochs; the diagonal dimension shows a chain of keys within each epoch. We use the following notation for keys: e_x is the root epoch key for epoch x, $c_{y,i}$ is the i^{th} chain key from epoch y, and $p_{y,i}$ is the actual encryption key for the i^{th} packet sent in epoch y. Upper case $P_{y,i}$ denotes the packet encrypted with $p_{y,i}$.

The sending ratchet computes the two keys $e_1, c_{0,0} \leftarrow \mathrm{KDF}_2(T_C)$ and immediately erases T_C (assuming this is the client; the server does the same operation but starting with T_S). e_1 is the root key for epoch 1, and $c_{0,0}$ is the 0^{th} chain key for epoch 0. The sender then computes $c_{0,1}, p_{0,0} \leftarrow \mathrm{KDF}_2(c_{0,0})$ to obtain the 0^{th} packet key $p_{0,0}$ and the next chain key $c_{0,1}$. For the next 30 seconds, each new outgoing packet $P_{0,i}$, the sender computes $c_{0,i+1}, p_{0,i} = \mathrm{KDF}_2(c_{0,i})$.

Once 30 seconds have elapsed, the sender stops using keys derived from the $c_{0,i}$ chain keys and ratchets to a new chain. The sender initializes this new chain by computing $e_2, c_{1,0} \leftarrow \mathrm{KDF}_2(e_1)$. The same pattern continues through all subsequent epochs.

On the receiver side, if the first packet $P_{0,1}$ arrives, the receiver decrypts it and immediately overwrites $c_{0,1}$ with $c_{0,2}$. If $P_{0,2}$ instead arrives first, then the receiver decrypts $P_{0,2}$, keeps $c'_{0,1}$ to be able to subsequently decrypt $P_{0,1}$ if that appears, and overwrites $c_{0,1}$ with $c_{0,3}$. Similar comments apply to subsequent packets throughout the lifetime of the tunnel.

Note that each packet is labeled with its position. The usage of epochs means that this shows only short-term information about the volume of tunnel traffic. The same information is available through traffic analysis in any case.

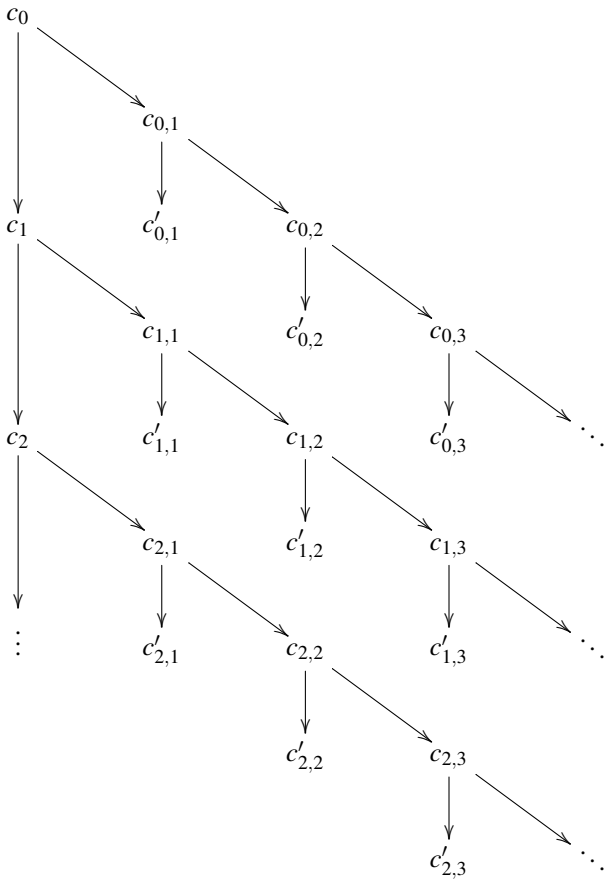

Figure 3: The PQConnect key ratchet. Keys are erased as soon as they are used, and in any case within two minutes. Key $c'_{0,i}$ is used for the ith client packet between time 0 and time 30, and is erased by the server as soon as it is used, or at the latest at time 120. Key $c'_{1,i}$ is used for the ith client packet between time 30 and time 60, and is erased by the server as soon as it is used, or at the latest at time 150.

4.9.1 Synchronizing local clocks

Key erasure is based on a monotonic clock so that changes to system time (e.g., from NTP) cannot cause delays in key erasure. Still, it is possible that because of local clock variation, two peers will not be completely in sync for the duration of a tunnel. If one peer receives a packet from epoch n at time $t < 30n$, they move their clock forward, setting the start time for epoch n to t (and the start time for $n+1$ to $t+30$, etc.). The expiration time for previous epochs is not affected.

However, if packets from epoch n arrive *later than* $30n$, the peer does not slow down their clock. Instead, they continue sending packets from the current epoch and let the other party advance their clock as above.

These rules together mean that the clock can be adjusted forward but never backward. This prevents an attacker from delaying the erasure of keys by delaying the arrival of packets.

Additionally, a peer's view of time should be consistent for both sending and receiving packets. If a peer makes a forward adjustment to a new epoch on their receiving ratchet, they should make the same adjustment on their sending ratchet.

This way a peer is never sending a message in an older epoch than the most recent one in which they have received a packet.

4.10 Message Format

When C wishes to send packet j in epoch i of a PQConnect tunnel, they encrypt the packet using key $c'_{i,j}$, then prepend the 32-byte `tunnelID`, 2-byte epoch number i, and 4-byte packet number j, and authenticate the encrypted data along with these fields. The epoch and index values are little-endian. This is then encapsulated as the payload of a UDP datagram and sent to the remote host. The packet arrives on the receiver's PQConnect UDP port. The receiver identifies that this packet is for tunnel `tunnelID` and then retrieves or computes $c'_{i,j}$ from their receiving ratchet to decrypt and verify the packet. The packet is then routed to the appropriate destination IP (if not the host machine) and port, and $c'_{i,j}$ is deleted. The structure of a PQConnect message is shown in Figure 4.

4.11 Session Cookies and Resumption

PQConnect servers can set a limit `MAX_CONNS` on the number of tunnel connections to maintain simultaneously. If the maximum number of active tunnels is reached and a new handshake message arrives, the server exports the state of its least recently active session as an encrypted session cookie to that client and replaces it with the new tunnel from the fresh handshake.

Session cookies are encrypted under a rotating secret key that updates every epoch, and only the four most recent keys are held at any given time. This gives session cookies the same lifetime as the tunnel itself.

A client who receives a session cookie simply stores it until the next time they wish to send a packet to the server. They prepend their packet with the cookie. If it is sufficiently fresh, the server decrypts the cookie and uses it to reconstruct the tunnel. Otherwise, the client must establish a new tunnel with a fresh handshake message.

5 Formal Verification of the PQConnect Handshake

To prove the security of the PQConnect handshake protocol we generated proofs symbolically using TAMARIN prover. We abstract slightly away from a client-server model and speak of an **I**nitiator, the one who sends the handshake, and **R**esponder, who receives it. The derived secrets T_I, and T_R are synonymous with T_C and T_S, respectively.

For a short introduction to TAMARIN, see Appendix B. For more extensive background on TAMARIN we direct the reader to [14].

See Appendix C for the full model and lemmas.

5.1 Security Properties

This section enumerates the security properties that the PQConnect handshake achieves.

5.1.1 Executability and correctness

Two parties are able to complete the handshake, and as a result derive the same `tunnelID`, $\mathtt{T_I}$, and $\mathtt{T_R}$ values.

5.1.2 Key confidentiality

In the absence of a break in the underlying cryptographic primitives, two parties who perform the handshake derive transport keys $\mathtt{T_I}$ and $\mathtt{T_R}$, and these keys are unknown to any third party.

5.1.3 Quantum confidentiality

An attacker who only obtains the long-term and ephemeral X25519 private keys of the Responder cannot recover the keys.

5.1.4 Forward secrecy

If the long-term private keys of the server are compromised after two parties perform the handshake, the transport keys remain confidential. We denote an attacker who later gains access to these keys a forward secrecy (FS) attacker.

5.1.5 Quantum forward secrecy

A FS attacker who obtains the Responder's long-term private keys and ephemeral X25519 private key, but not the sntrup4591761 private key, does not break the confidentiality of the transport keys.

5.1.6 Responder to initiator authentication

If I and R complete the handshake, then I knows they are communicating with R.

5.2 Verified Lemmas in TAMARIN

In this section we present and discuss the lemmas that we used to prove the security properties from the previous section. We discuss the validity of the lemmas and point out noteworthy actions to clarify what the lemma states. The lemmas along with the full model for the PQConnect handshake are included in Appendix C.

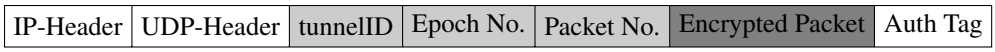

Figure 4: Structure of a PQConnect packet: Light gray areas are authenticated but not encrypted. Dark gray is authenticated and encrypted

5.2.1 Protocol executability and correctness

First we check that the modeled protocols execute as expected. If the model (or the protocol itself) contains errors that prevent it from successfully completing, then other properties about the handshake may trivially be true.

For both handshakes we prove the following lemmas, which TAMARIN verifies:

```
lemma 0_RTT_executable:
  /* There exists a trace, such that */
  exists-trace
  /* There exists a responder R, tunnelID id, transport keys ti */
  /* and tr, and times #i and #j*/
  "
    Ex R id ti tr #i #j.
      /* Such that the 0-RTT handshake finished for id at time #i */
      Zero_RTT(id) @ #i
      /* The initiator established a tunnel with R at time #i*/
      & InitiatorTunnel(R,id,ti,tr) @ #i
      /* and the R established a tunnel with the same */
      /* tunnelID ad transport keys at time #j*/
      & ResponderTunnel(R,id,tr,ti) @ #j
  "
```

5.2.2 Key confidentiality and forward secrecy

Any quantum FS attacker trying to break the confidentiality of the PQConnect handshake is of course free to perform pre-quantum attacks as well. Thus, showing quantum forward secrecy implies classical forward secrecy, quantum confidentiality, and classical confidentiality. We therefore prove a single lemma that satisfies all four confidentiality properties.

For the 0-RTT handshake, the Responder's ephemeral public keys are already public, so a quantum FS attacker is an attacker who can later learn the long-term private keys and the ephemeral X25519 private key of the Responder. We therefore simply require that the private sntrup4591761 key is never known to anyone besides *R*:

```
lemma 0_RTT_FS_confidential:
    /* Because ephemeral keys are public in this case, it is
    sufficient to show that if the ephemeral sntrup secret key is
    never revealed, the adversary cannot learn ti or tr. That is,
    even if the long term keys have been compromised (even before
    the handshake!), we achieve confidentiality */

    /* It cannot be that a  */
  "
    not(
      Ex S id ti tr #i.
        /* client has performed a 0-RTT handshake with a server'S' */
        InitiatorTunnel(S,id,ti,tr) @ #i
        & Zero_RTT(id) @ #i
        /* and the adversary knows ti or tr */
        & (
            (Ex #j. K(ti) @ #j)
            |(Ex #k. K(tr) @ #k)
          )
        /* without a reveal of ephemeral sntrup key occurring*/
        & not(Ex #r. PqEskReveal(S) @ #r )
    )
  "
```

5.2.3 Responder to initiator authentication

Finally we show that for the 0-RTT handshake, if an initiator has created a tunnel and a Responder has created the same tunnel, then the initiator must have created the tunnel with that particular Responder.

```
lemma responder_client_auth:
  /* For all Servers R and S and shared values tid,ti,tr, If a client
     has created a tunnel with R, and S has created a tunnel with the
     same values, then S must be R*/

  "
    All R S id ti tr #i #j.
      InitiatorTunnel(R,id,ti,tr) @ i & ResponderTunnel(S,id,ti,tr) @ j
          ==> S = R
  "
```

References

[1] PQConnect,. https://pqconnect.net, 2023.

[2] Michel Abdalla and Mihir Bellare. Increasing the lifetime of a key: A comparative analysis of the security of re-keying techniques. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 546–559. Springer, 2000.

[3] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 553–570. IEEE Computer Society, 2015.

[4] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1773–1788, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to. Accessed: 2021-08-04.

[6] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983.

[7] Jason A. Donenfeld and Kevin Milner. Formal Verification of the WireGuard Protocol. https://www.wireguard.com/papers/wireguard-formal-verification.pdf, 2018. Draft Revision b956944.

[8] SSL Ratios (public) - Mozilla Data Documentation. https://docs.telemetry.mozilla.org/datasets/other/ssl/reference. Accessed: 2023-10-02.

[9] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In Serge Vaudenay and Amr M. Youssef, editors, *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[10] Andreas Hülsing, Kai-Chun Ning, Peter Schwabe, Florian Weber, and Philip R. Zimmermann. Post-quantum WireGuard. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 304–321, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[11] Fabian Kuhn and René Struik. Random walks revisited: Extensions of Pollard's rho algorithm for computing multiple discrete logarithms. In Serge Vaudenay and Amr M. Youssef, editors, *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers*, volume 2259 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2001.

[12] Itsik Mantin and Adi Shamir. A practical attack on broadcast RC4. In Mitsuru Matsui, editor, *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2-4, 2001, Revised Papers*, volume 2355 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2001.

[13] Moxie Marlinspike and Trevor Perrin. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf, 2016.

[14] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[15] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF protocols. *RFC*, 8439:1–46, 2018.

[16] OpenSSH. Openssh 9.0 release notes. https://www.openssh.com/txt/release-9.0, 2022.

[17] Trevor Perrin. The Noise Protocol Framework, 2018. https://noiseprotocol.org/noise.pdf.

[18] W. Michael Petullo, Xu Zhang, Jon A. Solworth, Daniel J. Bernstein, and Tanja Lange. MinimaLT: minimal-latency networking through better security. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 425–438. ACM, 2013.

[19] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[20] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In Stephen Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 78–94. IEEE Computer Society, 2012.

[21] Peter Schwabe, Douglas Stebila, and Thom Wiggers. Post-quantum TLS without handshake signatures. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 1461–1480. ACM, 2020.

[22] Nicolas Sendrier. Decoding one out of many. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, volume 7071 of *Lecture Notes in Computer Science*, pages 51–67. Springer, 2011.

[23] Karolin Varner, Benjamin Lipp, Wanja Zaeske, and Lisa Schmidt. Rosenpass. https://rosenpass.eu/, 2023.

[24] Mullvad VPN. Experimental post-quantum safe VPN tunnels. https://mullvad.net/en/blog/2022/7/11/experimental-post-quantum-safe-vpn-tunnels/, 2022.

A ChaCha20-based KDF

PQConnect uses a Key Derivation Function (KDF) to deterministically compute, among other things, fresh keys for each packet it sends and receives. Applications frequently use

HMAC-based KDF (HKDF) to derive keys, but when large numbers packets are being sent or received, this can result in a significant amount of calls to the KDF function. When HKDF is used, that means a large amount of CPU resources are potentially devoted just to hashing.

PQConnect employs two strategies to reduce the CPU cost of key derivation. The first is that instead of HKDF, PQConnect uses a stream cipher based KDF, which is instantiated with ChaCha20, which we call ChaChaKDF. The second strategy is to batch key derivation to amortize the number of cycles used per byte of KDF output.

ChaCha20 is efficient in software and offers both the one-wayness and keyed pseudo random function (PRF) properties of HMAC, which is sufficient for a KDF. Unlike with HKDF, however, the number of cycles per byte of output key material with stream cipher-based key derivation amortizes as the number of output bytes grows, meaning that batch key derivation further reduces CPU load.

ChaCha20 maintains an internal state of 16 4-byte words (64 bytes total). Words 0-3 consist of the 16-byte ascii string "`expand 32-byte k`", words 4-11 consist of a 32-byte key, and words 12-15 usually consist of a counter and nonce (usually divided equally into two 8-byte values).

ChaChaKDF works as follows. It takes as input (1) an integer n, (2) a secret key k, and (3) an optional additional 32-byte value i, which may or may not be secret. It outputs n 32-byte keys derived from k (and optionally i).

If i is absent, ChaChaKDF simply uses ChaCha20 as a keyed pseudorandom generator (PRG) to output n 32-byte blocks of stream under key k, each of which is a separate output key. The output keys are uncorrelated from each other, following from the PRF" property of the underlying stream cipher. The one-wayness of the function follows from ChaCha20's security against key-recovery attacks.

If the 32-byte input i is present, ChaChaKDF performs the following: It first divides i into two 16-byte chunks, i_0 and i_1, and derives a new key k' from the first 32 bytes output when ChaCha20 is initialized with key k and counter/nonce i_0. The ChaCha20 stream cipher is then re-initialized with key k' and counter/nonce i_1, and we return n 32-byte blocks of stream as the output keys.

The entire function is given in pseudocode as:

```
ChaChaKDF(n, k, i):
  output_length = n * 32

  if i == null:
    nonce = {0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0} # 8 bytes
    counter = nonce
    return ChaCha20Stream(
      key = k,
      length = output_length,
      initial_counter = counter,
      nonce = nonce)

  else:
    counter = i[0: 8]
    nonce = i[8: 16]
    k_prime = ChaCha20Stream(
      key=k,
      length = 32,
      initial_counter = counter,
      nonce = nonce)

    new_counter = i[16: 24]
    new_nonce = i[24: 32]
    return ChaCha20Stream(
      key = k_prime,
      length = output_length,
      initial_counter = new_counter,
      nonce = new_nonce)
```

ChaChaKDF's use of i_0 and i_1 as the counter/nonce block of ChaCha20 mirrors the way in which the extended 192-bit nonce is used in XChaCha20. Because ChaCha20's security rests only on the secrecy of the 32-byte key k, and optional input i only takes the place of the 64-bit counter and nonce values, i can be public without affecting PRF property of ChaChaKDF.

B TAMARIN Prover

We give a brief introduction to TAMARIN and its semantics.

TAMARIN prover is a formal verification tool for proving properties of cryptographic protocols, such as confidentiality, peer-to-peer authentication, and forward secrecy [20]. It has been used to analyze security properties of widely deployed cryptographic network protocols, such as TLS 1.3 [4] and the WireGuard protocol handshake [7]. Protocol properties in TAMARIN are proven (or disproven) in the Dolev-Yao model—that is, where the adversary has complete control to eavesdrop, intercept, modify, and insert messages into the channel [6]. Additionally, the adversary may be given additional capabilities such as revealing the long term private keys of honest protocol participants, which can be useful for reasoning about properties such as forward secrecy.

B.1 Functions and equations

TAMARIN has built-in support for public key and symmetric cryptographic functions, hashes, and Diffie-Hellman operations. Additionally, users can define their own functions and equational theories. A function in TAMARIN is simply a name and an arity, for example `aenc/2`, which represents public key (asymmetric) encryption. By default, functions are one-way unless an equational theory is also defined. For example, the functions `aenc/2`, `adec/2`, and `pk/1`, can be combined in the equation `adec(aenc(m,pk(sk)),sk) = m`. This equation tells TAMARIN that given variables `m` and `sk`, the public key decryption of the public key encryption of a message `m` is `m`. By contrast, if the user wishes to define three hash functions `h1/1`, `h2/1`, and `h3/1`, TAMARIN will treat these functions as one-way functions with complete domain separation, without the user having to further specify any information.

B.2 Facts and rules

TAMARIN models are constructed from a multi-set of facts and rewriting rules for those facts. A fact is a unit of information about the state, which consists of a name and fixed arity. For example, the fact `!PQ_Ssk(A,sk)` is a binary fact on the variables `A` and `sk`. Rewriting rules are named triples consisting of a premise, an optional labeled action, and a result. The rule

```
rule Reveal_PQ_Ssk:
    [ !PQ_Ssk(A, sk) ]      //! denotes that this fact is persistent
  --[ PqSskReveal(A)]->
    [ Out(sk) ]
```

is a labeled transition that consumes fact `!PQ_Ssk(A,sk)` and replaces it with the special fact `Out(sk)`, which in TAMARIN signifies sending `sk` onto the public channel. A state transition in TAMARIN can occur if the facts of a rule's premise are in the current state. When that rule is applied, the facts in its premise are consumed and replaced by the facts in the result.

Facts can be labeled as persistent using the bang symbol `!`, meaning that they can be consumed indefinitely without being removed from the state. This is useful for facts that remain public throughout the duration of the protocol, such as the facts binding an actor's identity to their long-term keys.

TAMARIN provides a set of special facts that help modeling fresh values and network operations. The **Fr**`(x)` fact creates a fresh random value `x`. To model some untrusted value `x` arriving from the network, the **In**`(x)` fact is used, and as already shown in the example rule above, the **Out**`(x)` sends `x` onto the network.

Finally, TAMARIN allows the user to specify detailed protocol information for rules using the `let-in` keywords. This is easiest to explain by example, so consider the following rule:

```
rule example:
  let
    a = h(~nonce)
    b = kdf(a)
    c = kdf(<a,b>)
  in
    [ Fr(~nonce) ]
  -->
    [ Out(h(c)]
```

This rule generates a fresh nonce `~nonce` in its premise. It then computes values `a, b,` and `c` as described in the `let` clause. The result of the rule is to put `h(c)` onto the public channel (the rule contains no actions). The `let-in` construction makes it easy to implement rules that exactly match the computations performed during individual steps of the protocol.

B.3 Lemmas

In TAMARIN, properties about models are analyzed by writing and proving/disproving lemmas. Lemmas are guarded first-order logical statements about labeled actions over all possible rule executions. These sequences of actions are called traces. In the example rule above, the action `PqSskReveal(A)` is produced when the rule `Reveal_PQ_Ssk` is executed for some entity `A`. A trace where the predicate `PqSskReveal(A)` holds is one where a rule that produced this action fact was executed. Actions can be used to both reason about and restrict the executions of rules in a lemma.

Quantifiers and logical connectives, such as $\forall, \exists, \neg, \wedge, \vee$, and $\Rightarrow$ are expressed in TAMARIN as text or ASCII symbols such as `"All", "Ex", "not", &, |, and ==>`. The notion of time and ordering of actions can also be reasoned about with time variables. Time variables can be declared using the `#` symbol, and predicates can be bound to times using the `@` symbol. For instance, to say there exists a trace where actions `P(x)` and `Q(x)` both occur for some `x`, and `P(x)` occurs before `Q(x)`, you could express it as the formula

```
Ex x,#i,#j. P(x) @ #i & Q(x) @ #j & (#i < #j)
```

In addition to user-defined actions, TAMARIN internally defines actions that model the behavior of a Dolev-Yao adversary. For purposes of this work, the most important one of these rules is the special action fact **K**`(x)`, which indicates that the adversary **K**nows the value `x`. Thus to check that some value `x` is secret, you check whether there is any protocol trace where `K(x)` is true.

The TAMARIN prover uses a constraint solving algorithm to find counterexamples to the provided lemmas [20]. When a lemma is intended to prove a universally quantified statement, like $\forall_x P(x)$ for some predicate P, TAMARIN converts this to the equivalent existentially quantified statement $\neg\exists_x(\neg P(x))$, and then tries to find a contradiction. The decidability of statements in first-order logic is undecidable in general, so there exist lemmas for which the algorithm will not terminate. However, when the prover does terminate, then it either proves the statement to be true over unbounded executions or derives a contradiction.

C TAMARIN Prover Model

In this section we provide the TAMARIN model of the PQ-Connect handshake.

```
/*
Model of PQConnect Handshake
====================================
*/

theory PQCHandshake
begin

builtins: hashing, asymmetric-encryption, symmetric-encryption, diffie-hellman
functions: aeadenc/4, aeaddec/4, kdf/1
/*kdf2 and kdf3 represent the 2nd and third key output by the kdf on a given input*
functions: kdf2/1, kdf3/1
equations: aeaddec(k,n,aeadenc(k,n,m,ad),ad) = m

/* PKI */
rule Register_static_pq_pk:
```

```
    [ Fr(~ssk) ]
  -->
    [ !PQ_Ssk($S, ~ssk), !PQ_Spk($S, pk(~ssk)) ]

rule Register_static_npq_pk:
    [ Fr(~ssk) ]
  -->
    [ !NPQ_Ssk($S, ~ssk), !NPQ_Spk($S, 'g'^~ssk) ]


rule Register_ephemeral_pq_pk:
    [ Fr(~esk) ]
  -->
    [ !PQ_Esk($S, ~esk), !PQ_Epk($S, pk(~esk)) ]

rule Register_ephemeral_npq_pk:
    [ Fr(~esk) ]
  -->
    [ !NPQ_Esk($S, ~esk), !NPQ_Epk($S, 'g'^~esk) ]

/* These rules model key comprimse */
rule Reveal_npq_ssk:
    [ !NPQ_Ssk(A, ssk) ]
  --[ NpqSskReveal(A) ]->
    [ Out(ssk) ]

rule Reveal_pq_ssk:
    [ !PQ_Ssk(A, ssk) ]
  --[ PqSskReveal(A) ]->
    [ Out(ssk) ]

rule Reveal_pq_esk:
    [ !PQ_Esk(A, esk) ]
  --[ PqEskReveal(A) ]->
    [ Out(esk) ]

rule Reveal_npq_esk:
    [ !NPQ_Esk(A, esk) ]
  --[ NpqEskReveal(A) ]->
    [ Out(esk) ]


/* 0-RTT Handshake */
rule 0RTT_PQConnectI:
  let
    c0 = aenc(~k0, spkRmceliece)
    CI = ~k0
    HI = c0
    c1 = aeadenc(CI,'0','g'^~eskIx25519,HI)
    HI = h(<HI,c1>)
    k1 = spkRx25519^~eskIx25519
    CI = kdf(<CI,k1>)
    k2 = epkRx25519^~eskIx25519
    CI = kdf(<CI,k2>)
    c2 = aenc(~k3, epkRsntrup)
    c3 = aeadenc(CI,'0',c2,HI)
    CI = kdf(<CI,~k3>)
    HI = h(<HI,c3>)
    tid = kdf(<CI,HI>)
    TI = kdf2(<CI,HI>)
    TR = kdf3(<CI,HI>)
  in
    [ !PQ_Spk(R,spkRmceliece),
      !NPQ_Spk(R,spkRx25519),
      !PQ_Epk(R,epkRsntrup),
      !NPQ_Epk(R,epkRx25519),
      Fr(~eskIx25519),
      Fr(~k0),
      Fr(~k3)]
    --[Zero_RTT(tid), InitiatorTunnel(R,tid,TI,TR)]->
    [ Out(<'1',c0,c1,c3>) ]

rule 0RTT_PQConnectR:
  let
    k0 = adec(c0,~sskRmceliece)
    CR = k0
    HR = c0
    epkIx25519 = aeaddec(CR,'0',c1,HR)
    HR = h(<HR,c1>)
    k1 = epkIx25519^~sskRx25519
    CR = kdf(<CR,k1>)
    k2 = epkIx25519^~eskRx25519
    CR = kdf(<CR,k2>)
    c2 = aeaddec(CR,'0',c3,HR)
    k3 = adec(c2,~eskRsntrup)
    CR = kdf(<CR,k3>)
    HR = h(<HR,c3>)
    tid = kdf(<CR,HR>)
    TI = kdf2(<CR,HR>)
    TR = kdf3(<CR,HR>)
  in
    [ !PQ_Ssk($R,~sskRmceliece),
      !NPQ_Ssk($R,~sskRx25519),
      !PQ_Esk($R,~eskRsntrup),
      !NPQ_Esk($R,~eskRx25519),
      In(<'1',c0,c1,c3>)  ]
    --[Secret(tid), Secret(TR), Secret(TI), ResponderTunnel($R,tid,TR,TI)]->
    []

/* lemmas */

lemma 0_RTT_executable:
  /* There exists a trace, such that */
  exists-trace
  /* There exists a responder R, tunnelID id, transport keys ti */
  /* and tr, and times #i and #j*/
  "
    Ex R id ti tr #i #j.
      /* Such that the 0-RTT handshake finished for id at time #i */
      Zero_RTT(id) @ #i
      /* The initiator established a tunnel with R at time #i*/
      & InitiatorTunnel(R,id,ti,tr) @ #i
      /* and the R established a tunnel with the same */
      /* tunnelID ad transport keys at time #j*/
      & ResponderTunnel(R,id,tr,ti) @ #j
  "

lemma 0_RTT_FS_confidential:
    /* Because ephemeral keys are public in this case, it is
    sufficient to show that if the ephemeral sntrup secret key is
    never revealed, the adversary cannot learn ti or tr. That is,
    even if the long term keys have been compromised (even before
    the handshake!), we achieve confidentiality */

    /* It cannot be that a  */
  "
    not(
      Ex S id ti tr #i.
        /* client has performed a 0-RTT handshake with a server'S' */
        InitiatorTunnel(S,id,ti,tr) @ #i
        & Zero_RTT(id) @ #i
        /* and the adversary knows ti or tr */
        & (
            (Ex #j. K(ti) @ #j)
            |(Ex #k. K(tr) @ #k)
          )
        /* without there having been a reveal of the ephemeral sntrup key */
        & not(Ex #r. PqEskReveal(S) @ #r )
    )
  "

lemma responder_client_auth:
  /* For all Servers R and S and shared values tid,ti,tr, If a client
     has created a tunnel with R, and S has created a tunnel with the
     same values, then S must be R*/

  "
    All R S id ti tr #i #j.
      InitiatorTunnel(R,id,ti,tr) @ i & ResponderTunnel(S,id,ti,tr) @ j ==> S = R
  "
end
```

D Data flow — server discovery and DNS

This appendix shows the data flow in Figure 5.

Figure 5: Data flow of receiving information on the server's keys via DNS lookups.

E Data flow — key exchange and handshake

This appendix shows the dataflow of the key exchange and handshake in Figure 6

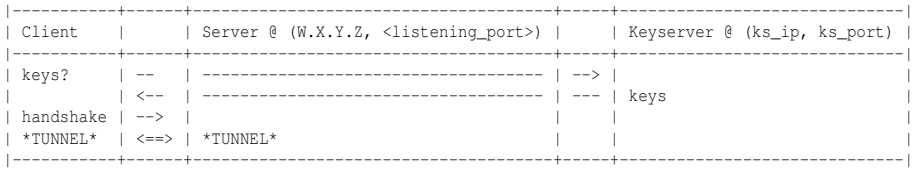

Figure 6: This figure shows the data flow between client and server establishing a connection