Artifact Evaluated NDSS
Available
Functional
Reproduced

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

Daniel J. Bernstein*‡, Tanja Lange†‡, Jonathan Levin‡† and Bo-Yin Yang‡
*University of Illinois at Chicago; Email: authorcontact-d@box.cr.yp.to
†Eindhoven University of Technology; Email: tanja@hyperelliptic.org
‡Academia Sinica; Email: j@jlev.in, byyang@iis.sinica.edu.tw
Date: 2024-12-04. Author list in alphabetical order;
see https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf.

***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.

The foundation of security in PQConnect is the server's long-term public key used to encrypt and authenticate all PQConnect packets. PQConnect makes a conservative choice of post-quantum KEM for this public key. 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 handshake component of PQConnect has been symbolically proven using Tamarin.

I. Introduction

CVEs mentioning TLS within the past 12 months include CVE-2024-5261 (in "LibreOffice, when used in LibreOfficeKit mode only, then curl's TLS certification verification was disabled"); CVE-2024-37309 (in CrateDB, a "high-risk vulnerability has been identified in versions prior to 5.7.2 where the TLS endpoint (port 4200) permits client-initiated renegotiation"); CVE-2024-37305 ("malformed input can lead to crashes or information leakage" in oqs-provider for "TLS, X.509, and S/MIME"); CVE-2024-32973 (in the Pluto language, "an attacker with the ability to actively intercept network traffic would be able to use a specifically-crafted certificate to fool Pluto into trusting it to be the intended remote for the TLS session"); CVE-2024-30166 (in the Mbed TLS library, "a malicious client can cause information disclosure or a denial of service because of a stack buffer over-read (of less than 256 bytes) in a TLS 1.3 server via a TLS 3.1 ClientHello"); CVE-2024-29963 (Brocade SANnav OVA has "hardcoded TLS keys used by Docker"); CVE-2024-29733 ("Improper Certificate Validation vulnerability in Apache Airflow FTP Provider"); CVE-2024-29209 (Phish Alert Button for Outlook "does not enforce strict SSL/TLS verification"; this "could allow an attacker to remotely execute arbitrary code on the host machine"); CVE-2024-28161 ("In Jenkins Delphix Plugin 3.0.1, a global option for administrators to enable or disable SSL/TLS certificate validation for Data Control Tower (DCT) connections is disabled by default"); 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-2023-4420, regarding TLS not being used in the SICK LMS5xx laser sensors; CVE-2023-4586, regarding a client for the Hot Rod data-access protocol not enabling hostname validation; CVE-2023-4331, saying that the Broadcom RAID Controller web interface has an insecure default TLS configuration; and many more.

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 [30], the percent of all web-page loads using TLS has increased from around

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241879
www.ndss-symposium.org

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 does not support TLS at all (as in CVE-2023-4420), 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 2.4 million pull requests; a skim of the most recent 50 (as of 2024-07-08) found 44 that were clearly 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 decrypting it later; see, e.g., [35]. Plenty of data 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., [61], [18], [37]. Various popular browsers, starting with Chrome version 116 released 2023-08-15, automatically encrypt 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 does not have it already. Everything then has to be repeated for clients and servers for SMTP, IMAP, Hot Rod, the Delphix DCT protocol, RADIUS (see the new attack [34]), and a very long list of further application-layer protocols. This is a clear path, but also a slow, labor-intensive path.

A. *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 [67], [68], the new Rosenpass [65], and VPNs based on OpenSSH, which has been using Streamlined NTRU Prime by default [52] 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 running PQConnect 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 III for further explanation of PQConnect's automatic peer discovery and authentication options.

This paper focuses on the PQConnect design, but we have also implemented PQConnect for GNU/Linux. See Section VII for a short summary and Appendix A for more details.

B. *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, 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 [31], [46]), but RC4 was not completely removed from TLS until version 1.3 in 2018 [56], 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 (quantified in [12]) 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 X25519 [10], a 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.

Most of our security analysis is manual, but symbolic security analysis of one component of PQConnect, namely the handshake, is within reach of existing automated tools and has been carried out using an existing prover. See Section V.

C. Performance

The McEliece system has 1MB public keys at the high security level we chose. Readers with concerns about the cost of transmitting 1MB might wonder whether it is better to take a less conservative post-quantum system, or, for environments not worried about future quantum threats, just ECDH without a post-quantum system. Either way would still add value beyond TLS as explained above, creating a second layer of defense with time-based key erasure and header encryption without touching application software.

However, our performance analysis in Section VI indicates that the McEliece system is already affordable. The critical point here is that, after using this KEM to create a tunnel, a PQConnect client automatically reuses the tunnel for any number of connections. The client also precomputes and caches KEM ciphertexts—which are just 194 bytes for the McEliece system—to efficiently build new tunnels if necessary. Either way, a single transmission of the server's long-term public key to the client protects any amount of traffic between the client and the server.

As an analogy, public-key cryptography is well known to be a negligible fraction of the cost of traditional VPNs with manually configured tunnels. The Mullvad and Rosenpass VPNs mentioned above already use the McEliece system; see [68] and [65]. A PQConnect client handles more public keys than a VPN client does, since a PQConnect client automatically retrieves long-term public keys from multiple servers, but these are still long-lasting keys that protect arbitrarily large volumes of user data. For TLS, applications are pressured to keep sessions short (see, e.g., [33], [38], and [66]), incurring frequent public-key operations for the sake of forward secrecy; PQConnect uses time-based key erasure to eliminate this tension. These structural features of PQConnect allow the choice of long-term KEM to skip performance-driven compromises and jump directly to what is best for security.

II. 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.

Given that attackers are storing data today for decryption by future quantum computers, what is most urgent today is adding post-quantum *encryption* to this data; but PQConnect is also designed so that its *authentication* will not need a subsequent post-quantum upgrade.

We also assume that the attacker can compromise a peer device in the future, for example by gaining physical access to the 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.

We assume that the attacker is powerful enough not just to compromise a device but to stay in control of the device, as in, e.g., [29], [55], and [48]. Under this assumption, any claims of "post-compromise security" are automatically broken. We focus on security *before* the device is compromised.[1]

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.

III. Data Flow for PQConnect Integration

This section explains PQConnect's application integration: how a PQConnect client recognizes PQConnect servers and arranges for applications on the same machine to send traffic through the PQConnect tunnel, *without* changes to the application software. We focus on GNU/Linux for concreteness.

The closest previous integration work that we are aware of is MinimaLT [54], which automatically creates end-to-end pre-quantum tunnels covering all application traffic. The applications in [54] were written for a new network API designed from the outset to use these tunnels, whereas in this section the applications are *unmodified* GNU/Linux programs unaware of PQConnect.

The high-level data flow is as follows. The administrator of a PQConnect client device installs and runs the PQConnect client software. This software automatically recognizes (see Section III-A) when application software on that device—for concreteness, imagine an XMPP client—is connecting to a server that supports PQConnect. This software then creates a PQConnect tunnel to that server (see Section IV for the protocol) if it does not have a recently used tunnel to that server. This software then captures packets from the application (see Section III-B), and routes those packets through that tunnel. See Sections III-C and III-D for attack analysis.

A. 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 there is a multi-part response: e.g., `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.[2] For example, a client looking up the address for the server `www.pqconnect.net` receives a `CNAME` pointing to `pq1u1hy1ujsuk258krx3ku6wd9rp96kfxm64mgct3s3j26udp57dbu1.pqconnect.net` and an `A` pointing this `pq1...bu1.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[3] from DNSCurve's `uz5`.

Instead of, or as a supplement to, distributing the short name `www.pqconnect.net`, server administrators can distribute the name `pq1u1h...bu1.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 III-C below.

B. 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.

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.

[1] Administrators and cryptographers considering weaker attack models, as in [22], often decide to rotate long-term keys, for example switching to a new long-term key every three months. PQConnect continues to work transparently in this scenario: the client sees a new DNS record with a new key hash, and all cryptographic operations are indexed by that key hash, as explained later.

[2] This is why we use CNAME; for simplicity, we focus on protecting names not at DNS zone cuts. For comparison, RFC 4025 [57] specifies a format for "IPSECKEY" records that are *not* naturally returned to clients: these records require a separate DNS lookup beyond looking up the server's IP address. Trying to use this for *automatic* tunnels would mean accompanying essentially every DNS lookup with an IPSECKEY lookup. This is not necessarily a performance problem, but it raises tricky questions such as (1) how to define "essentially" to avoid sometimes triggering combinatorial explosions and (2) how to handle timeouts from servers that drop unusual record types. Passively checking server names simplifies the engineering and lets PQConnect clients follow a simple rule of zero extra packets for non-PQConnect servers.

[3] 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`.

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.

C. 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 packets. 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 (see, e.g., [32]); the real responses are then ignored.

There has been some deployment of cryptographic add-ons to DNS, such as DNSSEC, DNSCurve, DNS over TLS, and DNS over HTTPS. Today these normally use pre-quantum cryptography—e.g., 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 `pq1u1h...bu1.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.

D. 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 `pq1u1h...bu1.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 III-C), 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. 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.

IV. Protocol Specification

A. Cryptographic Notation

We use the following notation:

- $a \leftarrow b$ means assigning the value of b to the variable a.
- $\mathbf{spk^X_p}, \mathbf{ssk^X_p}, \mathbf{epk^X_p}, \mathbf{esk^X_p}$ are static public, static private, ephemeral public, and ephemeral private keys for peer **p** and public-key cryptosystem **X**.
- **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 a pseudorandom function of the input ("implicit rejection").
- **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 [50]). The output is $\mathbf{c}^* = (\mathbf{c}, \mathbf{t})$ for a ciphertext **c** and 16-byte authentication tag **t**.
- **AEAD.Dec(k, n, 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**.
- **KDF**$_n$**(k, i)** Generates n 32-byte keys from key **k** and optional input **i**. The KDF uses ChaCha20 as the underlying pseudorandom function (PRF) and is described in detail in Appendix D.
- $\mathbf{C_p}, \mathbf{H_p}$ are "CipherState" and "HandshakeState" objects (see below) 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 server).
- **tunnelID** This is a 32-byte pseudo-random value that uniquely identifies a tunnel.

B. 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 of 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...u1.pqconnect.net`, there are public records `pq1...u1.pqconnect.net TXT "p=42424"` and `ks.pq1...u1.pqconnect.net TXT "ip=131.155.69.126;p=42425"` specifying the PQConnect listening port and keyserver address.

A long-term PQConnect public key is a mceliece6960119 key (along with an X25519 key as explained in Section I). This key is split 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 IV-C.

Once the long-term key is verified, the client requests an ephemeral key from the keyserver to compute a handshake message. The ephemeral key is an sntrup761 key (along with an X25519 key). Ephemeral keys have an issuing period of 30 seconds, and the private keys are erased after 120 seconds. That is, an ephemeral key distributed to a client at time t_0 might be issued again at $t_0 + 29$, but it will not be issued at time $t_0 + 30$, and any ciphertext created with that key will be unrecoverable by the server at time $t > t_0 + 120$.

The server uses its clock to rotate and erase keys. For all its connections it defines an epoch of 30 seconds during which an sntrup761 key should be used. To deal with slow networks, synchonization issues, and clock skew the server will also accept connections using the keys for the previous three epochs. The server erases the key 120 seconds after it first started using it.

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

C. 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 each packet's correctness 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 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 root and depth-1 packets and checks that the root equals the hash obtained from DNS and that the depth-1 node hashes to the root. It then requests and hashes the 26 depth-2 packets, verifying them against the depth-1 node. Finally it requests and verifies the 910 leaf packets.

Verifying a packet only requires that the parent packet containing its hash has already been received and verified. This allows for some level of parallelism when sending requests; request packets for one level of the tree can be sent before response packets for that level have been received. This creates a lower bound of 3-RTT for the key request, one round trip for each level. The client uses a congestion-control algorithm based on [19] and [15] to select transmission speeds.

D. The PQConnect Handshake

Once the client has obtained and verified the server's long-term keys, it 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. Of course, saying that the *handshake* is 0-RTT does not mean that the *pre-handshake* operations of receiving the server's keys are 0-RTT.

The 0-RTT handshake means that clients who already have a server's public keys can send tunneled packets immediately with the handshake message, reducing latency. On the other hand, all per-connection randomness comes from the client. We discuss protection against replay attacks in Section IV-I.

PQConnect's handshake protocol involves four public keys: the server's long-term and ephemeral KEM keys, plus long-term and ephemeral X25519 keys as explained in Section I. These keys are used in a "nested" manner: public key operations are performed sequentially, and each subsequent (inner) operation is cryptographically protected by a secret key derived from all previous (outer) operations. See Appendix C.

E. 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 [53], such as the use of CipherState and HandshakeState variables.

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 1.

The client $\mathcal{C}$ first generates a random 32-byte ephemeral X25519 key and its corresponding 32-byte public key. Then $\mathcal{C}$ encapsulates a random key $\mathtt{k}_0$ to the server's long term mceliece6960119 public key as $\mathtt{c}_0$. Then $\mathcal{C}$ encrypts its 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, it computes 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 sntrup761 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-byte "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 1299 bytes: the 2-byte prefix, 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 1055 bytes for $\mathtt{c}_3^*$ (1039-byte sntrup761 ciphertext + 16 byte authentication tag).

Upon receipt of $\mathcal{C}$'s message, the server $\mathcal{S}$ 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. Then $\mathcal{S}$ obtains the same shared values $\mathtt{tunnelID}$, $\mathtt{T}_\mathtt{c}$, and $\mathtt{T}_\mathtt{s}$.

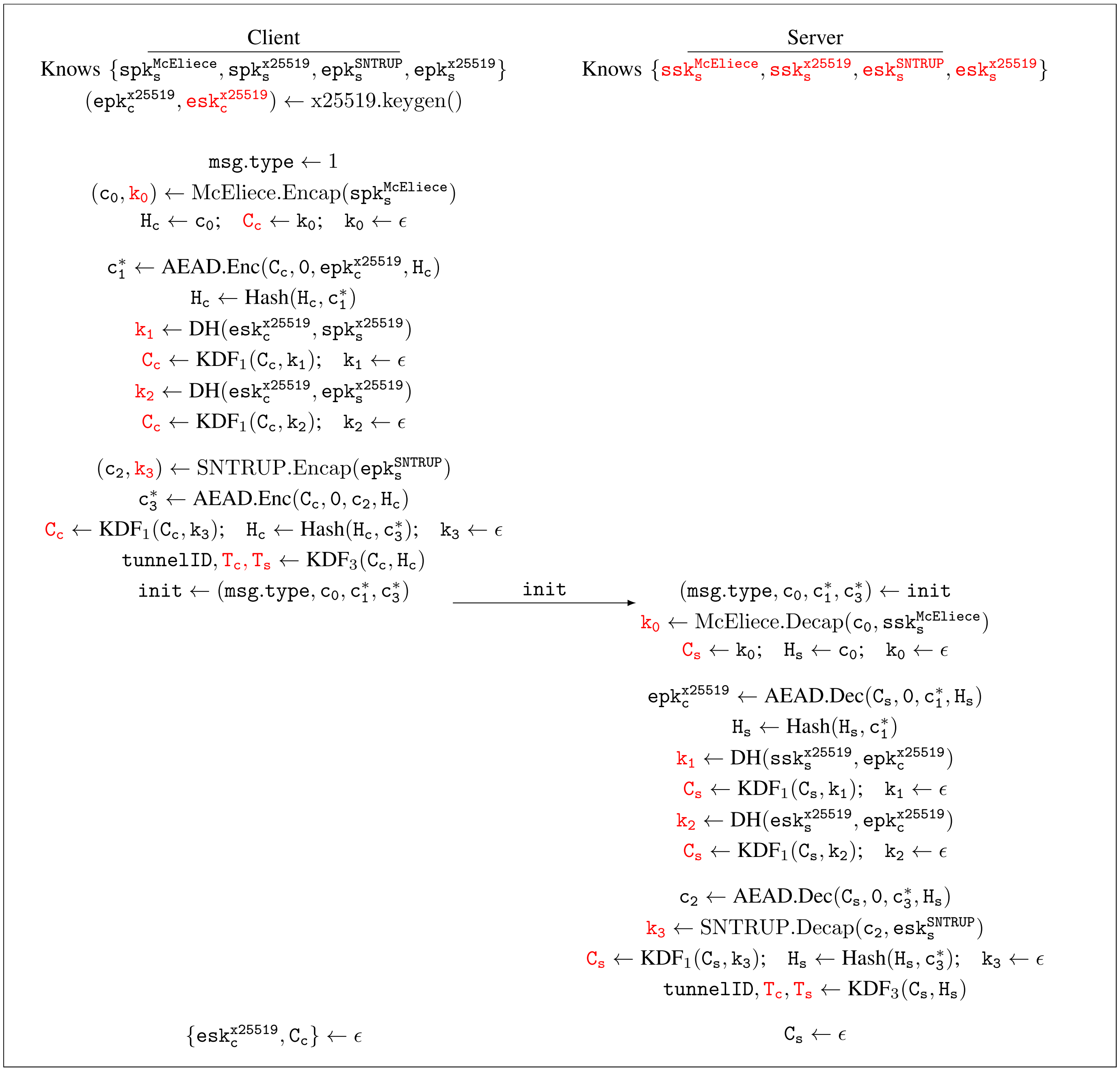
Client
Knows $\{\mathtt{spk}_s^{\mathtt{McEliece}}, \mathtt{spk}_s^{\mathtt{x25519}}, \mathtt{epk}_s^{\mathtt{SNTRUP}}, \mathtt{epk}_s^{\mathtt{x25519}}\}$
$(\mathtt{epk}_c^{\mathtt{x25519}}, \mathtt{esk}_c^{\mathtt{x25519}}) \leftarrow \mathrm{x25519.keygen}()$
$\mathtt{msg.type} \leftarrow 1$
$(\mathtt{c}_0, \mathtt{k}_0) \leftarrow \mathrm{McEliece.Encap}(\mathtt{spk}_s^{\mathtt{McEliece}})$
$\mathtt{H}_c \leftarrow \mathtt{c}_0; \quad \mathtt{C}_c \leftarrow \mathtt{k}_0; \quad \mathtt{k}_0 \leftarrow \epsilon$
$\mathtt{c}_1^* \leftarrow \mathrm{AEAD.Enc}(\mathtt{C}_c, 0, \mathtt{epk}_c^{\mathtt{x25519}}, \mathtt{H}_c)$
$\mathtt{H}_c \leftarrow \mathrm{Hash}(\mathtt{H}_c, \mathtt{c}_1^*)$
$\mathtt{k}_1 \leftarrow \mathrm{DH}(\mathtt{esk}_c^{\mathtt{x25519}}, \mathtt{spk}_s^{\mathtt{x25519}})$
$\mathtt{C}_c \leftarrow \mathrm{KDF}_1(\mathtt{C}_c, \mathtt{k}_1); \quad \mathtt{k}_1 \leftarrow \epsilon$
$\mathtt{k}_2 \leftarrow \mathrm{DH}(\mathtt{esk}_c^{\mathtt{x25519}}, \mathtt{epk}_s^{\mathtt{x25519}})$
$\mathtt{C}_c \leftarrow \mathrm{KDF}_1(\mathtt{C}_c, \mathtt{k}_2); \quad \mathtt{k}_2 \leftarrow \epsilon$
$(\mathtt{c}_2, \mathtt{k}_3) \leftarrow \mathrm{SNTRUP.Encap}(\mathtt{epk}_s^{\mathtt{SNTRUP}})$
$\mathtt{c}_3^* \leftarrow \mathrm{AEAD.Enc}(\mathtt{C}_c, 0, \mathtt{c}_2, \mathtt{H}_c)$
$\mathtt{C}_c \leftarrow \mathrm{KDF}_1(\mathtt{C}_c, \mathtt{k}_3); \quad \mathtt{H}_c \leftarrow \mathrm{Hash}(\mathtt{H}_c, \mathtt{c}_3^*); \quad \mathtt{k}_3 \leftarrow \epsilon$
$\mathtt{tunnelID}, \mathtt{T}_c, \mathtt{T}_s \leftarrow \mathrm{KDF}_3(\mathtt{C}_c, \mathtt{H}_c)$
$\mathtt{init} \leftarrow (\mathtt{msg.type}, \mathtt{c}_0, \mathtt{c}_1^*, \mathtt{c}_3^*)$
$\xrightarrow{\mathtt{init}}$
$\{\mathtt{esk}_c^{\mathtt{x25519}}, \mathtt{C}_c\} \leftarrow \epsilon$
Server
Knows $\{\mathtt{ssk}_s^{\mathtt{McEliece}}, \mathtt{ssk}_s^{\mathtt{x25519}}, \mathtt{esk}_s^{\mathtt{SNTRUP}}, \mathtt{esk}_s^{\mathtt{x25519}}\}$
$(\mathtt{msg.type}, \mathtt{c}_0, \mathtt{c}_1^*, \mathtt{c}_3^*) \leftarrow \mathtt{init}$
$\mathtt{k}_0 \leftarrow \mathrm{McEliece.Decap}(\mathtt{c}_0, \mathtt{ssk}_s^{\mathtt{McEliece}})$
$\mathtt{C}_s \leftarrow \mathtt{k}_0; \quad \mathtt{H}_s \leftarrow \mathtt{c}_0; \quad \mathtt{k}_0 \leftarrow \epsilon$
$\mathtt{epk}_c^{\mathtt{x25519}} \leftarrow \mathrm{AEAD.Dec}(\mathtt{C}_s, 0, \mathtt{c}_1^*, \mathtt{H}_s)$
$\mathtt{H}_s \leftarrow \mathrm{Hash}(\mathtt{H}_s, \mathtt{c}_1^*)$
$\mathtt{k}_1 \leftarrow \mathrm{DH}(\mathtt{ssk}_s^{\mathtt{x25519}}, \mathtt{epk}_c^{\mathtt{x25519}})$
$\mathtt{C}_s \leftarrow \mathrm{KDF}_1(\mathtt{C}_s, \mathtt{k}_1); \quad \mathtt{k}_1 \leftarrow \epsilon$
$\mathtt{k}_2 \leftarrow \mathrm{DH}(\mathtt{esk}_s^{\mathtt{x25519}}, \mathtt{epk}_c^{\mathtt{x25519}})$
$\mathtt{C}_s \leftarrow \mathrm{KDF}_1(\mathtt{C}_s, \mathtt{k}_2); \quad \mathtt{k}_2 \leftarrow \epsilon$
$\mathtt{c}_2 \leftarrow \mathrm{AEAD.Dec}(\mathtt{C}_s, 0, \mathtt{c}_3^*, \mathtt{H}_s)$
$\mathtt{k}_3 \leftarrow \mathrm{SNTRUP.Decap}(\mathtt{c}_2, \mathtt{esk}_s^{\mathtt{SNTRUP}})$
$\mathtt{C}_s \leftarrow \mathrm{KDF}_1(\mathtt{C}_s, \mathtt{k}_3); \quad \mathtt{H}_s \leftarrow \mathrm{Hash}(\mathtt{H}_s, \mathtt{c}_3^*); \quad \mathtt{k}_3 \leftarrow \epsilon$
$\mathtt{tunnelID}, \mathtt{T}_c, \mathtt{T}_s \leftarrow \mathrm{KDF}_3(\mathtt{C}_s, \mathtt{H}_s)$
$\mathtt{C}_s \leftarrow \epsilon$

Fig. 1. The handshake component of PQConnect. Red text denotes that the variable is secret. The assignment $v \leftarrow \epsilon$ denotes erasure of variable v.

For each AEAD operation, the HandshakeState variable is used as authenticated data, ensuring that the handshake succeeds only 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`, $\mathtt{T}_c$, and $\mathtt{T}_s$.

F. The PQConnect Key Ratchet

PQConnect tunnels encrypt each packet with a one-time key that the *sender* erases immediately 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. If decryption keys were to remain on the recipient's device until their corresponding packets arrive, this would also have the effect of preventing their erasure. Our model 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 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. [4]). PQConnect's ratchet, depicted in Figure 2, is designed to be able to handle large packet volumes including out-of-order and delayed packets; compared to Signal's symmetric-key ratchet [47], there is an extra dimension in Figure 2.

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`, $\mathtt{T_c}$, and $\mathtt{T_s}$. 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). $\mathtt{T_c}$ is the root of the client's sending chain, and $\mathtt{T_s}$ is the root of the server's sending chain.

Figure 2 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 $e_1, c_{0,0} \leftarrow \text{KDF}_2(\mathtt{T_c})$ and immediately erases $\mathtt{T_c}$ (assuming this is the client; the server does the same operation but starting with $\mathtt{T_s}$). 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 $c_{0,1}, p_{0,0} \leftarrow \text{KDF}_2(c_{0,0})$ to obtain the 0^{th} packet key $p_{0,0}$ and the next chain key $c_{0,1}$. For the next 30 seconds, for each new outgoing packet $P_{0,i}$, the sender computes $c_{0,i+1}, p_{0,i} = \text{KDF}_2(c_{0,i})$.

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 $e_2, c_{1,0} \leftarrow \text{KDF}_2(e_1)$. The same pattern continues through all epochs.

On the receiver side, if the receiver initially has keys $p_{0,0}$ and $c_{0,1}$ ($c_{0,0}$ is erased after being used as input by the KDF), and the first packet $P_{0,0}$ arrives, the receiver decrypts it with $p_{0,0}$ and ratchets the chain forward, overwriting $c_{0,1}$ with $c_{0,2}$ (and generating $p_{0,1}$). If $P_{0,1}$ instead arrives first, then the receiver computes $c_{0,2}$ and $p_{0,1}$ from $c_{0,1}$, decrypts $P_{0,1}$, and keeps $p_{0,0}$ to be able to subsequently decrypt $P_{0,0}$ if that appears. After two minutes all keys stemming from e_0 are deleted. 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.

$$\begin{tikzcd}
e_0 \arrow[dd] \arrow[dr] & & & & \\
& c_{0,0} \arrow[d] \arrow[dr] & & & \\
e_1 \arrow[dd] \arrow[dr] & p_{0,0} & c_{0,1} \arrow[d] \arrow[dr] & & \\
& c_{1,0} \arrow[d] \arrow[dr] & p_{0,1} & c_{0,2} \arrow[d] \arrow[dr] & \\
e_2 \arrow[dd] \arrow[dr] & p_{1,0} & c_{1,1} \arrow[d] \arrow[dr] & p_{0,2} & \ddots \\
& c_{2,0} \arrow[d] \arrow[dr] & p_{1,1} & c_{1,2} \arrow[d] \arrow[dr] & \\
\vdots & p_{2,0} & c_{2,1} \arrow[d] \arrow[dr] & p_{1,2} & \ddots \\
& & p_{2,1} & c_{2,2} \arrow[d] \arrow[dr] & \\
& & & p_{2,2} & \ddots
\end{tikzcd}$$

Fig. 2. The PQConnect key ratchet. Keys are erased as soon as they are used, and in any case within two minutes. Key $p_{0,i}$ 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 $p_{1,i}$ 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.

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 also 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.

G. Message Format

When $\mathcal{C}$ wishes to send packet j in epoch i of a PQConnect tunnel, they encrypt the packet using key $p_{i,j}$, then prepend the 32-byte `tunnelID`, 2-byte epoch number i, and 4-byte packet number j, and authenticate the encrypted data along

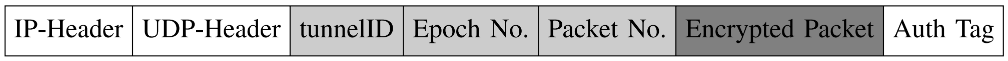

Fig. 3. Structure of a PQConnect packet: Light gray areas are authenticated but not encrypted. Dark gray is authenticated and encrypted

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 $p_{i,j}$ from their receiving ratchet to decrypt and verify the packet. The decrypted inner packet is then routed to the specified IP and port, and $p_{i,j}$ is deleted. The structure of a PQConnect message is shown in Figure 3.

H. 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.

I. Replay Protection

By virtue of being 0-RTT, client handshake messages are replayable, since the server does not contribute randomness. Processing the same handshake message twice will result in the same `tunnelID` and session keys. This raises two concerns. The first is at the protocol level. Replaying a handshake message that the server has already seen must not affect an existing tunnel with a client. The second is at the cryptographic level and relates to AEAD security problems under nonce reuse. Replaying a handshake must not give the attacker the ability to forge ciphertexts or break confidentiality.

PQConnect handles replays as follows. Recall that, for forward secrecy, each handshake is bound to an sntrup761 key that is erased after 120 seconds. The server keeps a list of McEliece ciphertexts used in successful handshake messages for each of the four currently valid ephemeral keys (simply indexed $0, 1, 2, 3$ and used round robin). The server checks each received ciphertext against the list, discarding any replay without performing a cryptographic operation. When an ephemeral key is erased, the corresponding list of used ciphertexts is also erased; those ciphertexts are invalid, making replays ineffective.

Replay of the symmetrically encrypted messages of the PQConnect tunnel will have no effect as each party erases keys as soon as they are used, hence, the replayed message cannot be decrypted and will be dropped.

J. Denial-of-Service Mitigation

Attackers with the powers considered in our threat model (see Section II) can trivially deny service for all network protocols by simply dropping all packets. Less powerful attackers can deny service at low cost by flooding the network. Any claims of protection against denial of service are necessarily against weak attackers and should be accompanied by specification of the costs for breaking the protection. We make a few comments in this section on specific attack avenues, but a full analysis of denial-of-service attacks is outside the scope of this paper.

Given the damage of amplification attacks (see, e.g., [58] and [28]) and the evidence of 1MB long-term public keys being affordable in context (see Section VI-B), we choose to spend another 1MB to have the server require the client's key-request packets to be as large as the server's responses, eliminating amplification. Clients simply zero-pad key-request packets to the required length.[4]

It is not as easy to prevent attackers from flooding a server with bogus handshake messages. In PQConnect, unlike a closed network such as 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.

Each handshake message triggers computation for four public-key operations: mceliece6960119 decapsulation, sntrup761 decapsulation, and two X25519 operations. Overall these take about 2^{19} cycles on a typical CPU core (see Section VI-C). There can be many more cycles for non-cryptographic operations (depending on software details), and each tunnel consumes RAM on the server.

We have a preliminary implementation of a hashcash-based challenge mechanism that servers can use if they are under load. This mechanism avoids keeping state on the server relating to who has been issued a challenge, and avoids amplifying client requests. The mechanism works as follows.

The server can respond to a handshake by sending a challenge message to the client instead of setting up a tunnel. The challenge consists of an authentication tag under a secret key held by the server. The authenticated data includes a timestamp value (indicating freshness), the client's IP address

[4]We could reduce the risk of network-layer compression by having the server require padding that is more expensive for networks to recognize, such as a ChaCha20 key followed by output from that key.

and port (to bind the challenge to the client), and a difficulty level `hardness_bits` selected by the server. The server sends `hardness_bits`, the timestamp, and the challenge to the client.

When the client receives the challenge, if it accepts the difficulty level, it computes a byte string that, when hashed together with the constant string `PQConnectChallenge` and the challenge, produces a hash with a leading number of zero bits at least as large as the difficulty level. The client then resends its handshake message along with the original timestamp, hardness, original authentication tag, and solution.

When the server receives the solution, it checks that the timestamp is fresh, that the hash begins with enough zero bits, and that the original authentication tag matches. If these tests pass, the server processes the accompanying handshake message.

V. Formal Verification of the PQConnect Handshake

The security analysis throughout this paper was generally carried out by hand, as noted in Section I. Computer-checked analyses reduce the risk of error to the extent that they can be carried out, and they turn out to be feasible for the handshake inside PQConnect. This section explains how we generated symbolic proofs for the handshake using Tamarin.

For a short introduction to Tamarin, see Appendix E. For more extensive background on Tamarin we direct the reader to [49]. See [2] for our full model and lemmas. It would also be possible to build a more complex model that accounts for further structure of elliptic curves as in [24]; as in [10], we are using plain ECDH and choosing private keys as multiples of 8, which is called "ClearPoint" in [24] and stops small-subgroup attacks. The structure of X25519 stops invalid-curve attacks; see [10].

In this section, 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 $\mathtt{T_I}$, and $\mathtt{T_R}$ are synonymous with $\mathtt{T_c}$ and $\mathtt{T_s}$, respectively.

A. Security Properties

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

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.

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.

3) Quantum confidentiality: An attacker who only obtains the long-term and ephemeral X25519 private keys of the Responder cannot recover the keys.

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) Quantum forward secrecy: A FS attacker who obtains the Responder's long-term private keys and ephemeral X25519 private key, but not the sntrup761 private key, does not break the confidentiality of the transport keys.

6) Responder to initiator authentication: The only party capable of decrypting I's messages is R. If I and R complete the handshake with matching secrets, then I is communicating with R.

B. 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. See [2] for the full model.

1) Protocol executability and correctness: First we check that the modeled protocol executes 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
  "
```

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.

The Responder's ephemeral public keys for the PQConnect handshake 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 sntrup761 key is never known to anyone besides R:

```
lemma 0_RTT_FS_confidential:
  /* For all handshakes occuring at time i */
  "
    All S id ti tr #i #j #k.
      (
        InitiatorTunnel(S,id,ti,tr) @ #i
      /* if long term key compromise occurs after time i */

        & NpqSskReveal(S) @ #j
        & (i < j)
        & PqSskReveal(S) @ #k
        & (i < k)
      /* and there is never also a compromise of
      the server's Post-Quantum ephemeral keys */

        & not(Ex #l. PqEskReveal(S) @ #l )
      )
      ==>
      /* then at no time does an adversary learn ti or tr */
      (
        not(Ex #r. K(ti) @ #r)
        &not(Ex #s. K(tr) @ #s)
      )
  "
```

3) Responder to initiator authentication: Finally we show that, 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,tr,ti) @ j
          ==> S = R
  "
```

VI. Cost Analysis

There is a long history of concerns being raised about the bandwidth and/or CPU time consumed by cryptography. Often these concerns delay deployment of safer cryptography.

Consider, for example, RSA-512, RSA-1024, and RSA-2048, which are typically estimated to have (pre-quantum) security levels around 2^{64}, 2^{80}, and 2^{112} respectively; see [8, Section 6.2] for a survey covering these estimates and smaller estimates. The first public report of an RSA-512 factorization [21], at the turn of the century, noted that "512–bit RSA keys protect 95% of today's E-commerce on the Internet". A 2007 study [44] found 88% of TLS servers using RSA-1024, and a 2011 study [36, Figure 11] found 50% of TLS servers still using RSA-1024; unlike the usage of RSA-512, the usage of RSA-1024 cannot be explained by export controls.

Today the dollar cost of cryptography is much lower: networks send much more data per dollar, CPUs carry out much more computation per dollar, and cryptography has been streamlined. For example, X25519—which is used for the "vast majority" of TLS connections monitored in [45]; see also [64], [69], [5] for confirming measurements—accounts for just $1/2000$ of Meta's total CPU cycles, according to [70]. A recent estimate [11] is that the costs of a server CPU cycle and of sending a byte through the Internet have dropped to about 2^{-51} dollars and about 2^{-40} dollars respectively; the low costs have enabled continuing increases in video traffic, which according to [59] is now the bulk of all Internet traffic. Meanwhile there is more awareness of the damage caused by security failures, and there are correspondingly large budgets available for security; see, e.g., [40] and [6]. But users might still be concerned about the deployability of a 1MB public key as part of a network-security mechanism.

This section analyzes from first principles the bandwidth and CPU time used by PQConnect—including, but not limited to, the McEliece costs. Section VI-A analyzes the pre-handshake cost of clients retrieving public keys from a server; this is where the McEliece public-key size appears. Section VI-B puts this cost in context, considering the cost of transmitting user data. Section VI-C analyzes the handshake cost, the cost of clients setting up tunnels to the server. Section VI-D analyzes the PQConnect costs per byte of user data. Section VI-E analyzes the costs that PQConnect incurs for non-PQConnect devices. Appendix B describes measurements of total costs of our implementation in end-to-end experiments.

A. Pre-Handshake Cost

The following paragraphs investigate the costs incurred for N PQConnect clients to retrieve public keys from a PQConnect server. Of course, the network-wide costs multiply this by the number of servers, with N averaged appropriately.

We begin by reviewing relevant microbenchmarks for public-key cryptography. Public keys are 1047319 bytes for mceliece6960119; 1158 bytes for sntrup761; and 32 bytes for X25519. For speeds, we consider CPU cycles on Intel's Skylake microarchitecture; this microarchitecture was introduced in 2015, is shared by Kaby Lake, Coffee Lake, Comet Lake, etc., and was not superseded until Ice Lake. The cryptographic libraries that we use report the following cycles for key generation on Skylake, specifically on a 3GHz Intel Xeon E3-1220 v5 with Turbo Boost disabled: 344836760 for mceliece6960119; 831462 for sntrup761; 29692 for X25519. Full digits of the numbers here are provided to support spot-checks against [14] and [17], not to suggest that the cycle counts are stable down to the last digit.

We next review what these microbenchmarks say about PQConnect. Key generation for mceliece6960119 takes a user-perceptible amount of time, for example 0.1 seconds on a 3GHz Skylake core, but is used in PQConnect only to compute the server's long-term key. Even if the administrator decides to generate a new long-term key every three months (rather than just once when PQConnect is installed on the server), the CPU time investment here is negligible. Similarly, the server generates a new sntrup761 key (and X25519 key) every 30 seconds, but this consumes only 1 out of every 100000 CPU cycles on a single 3GHz Skylake core.

What is repeated across all N clients is the network traffic to retrieve these keys from the server. Transmitting the server's key means transmitting a 1MB mceliece6960119 public key, plus negligible costs for the sntrup761 public key, the X25519 keys, and miscellaneous overhead. Recall from Section IV-J that we also send 1MB from the client to the server to eliminate amplification of key-request packets.

B. Cost in Context

It is important to note that the cost of transmitting the server's 1MB key, plus 1MB of defense against amplification, is a *per-client* cost, not a *per-connection* cost. This difference is critical for a cost analysis.

Consider the measurements from [45] of the TLS traffic generated by the most popular Android apps, specifically the

45 most popular general-purpose apps and the 45 most popular games. An average app in just *five minutes* of usage generated 13.6MB of traffic to 45 servers using 144 TLS handshakes with 36 TLS resumptions.

Seeing multiple TLS handshakes per server is not surprising. TLS software is organized around application-layer connections; sharing handshakes across connections takes extra work. TLS applications are also encouraged to start new sessions frequently for forward secrecy; see the references in Section I.

PQConnect is different. Tunnels to each server are handled centrally by the PQConnect software on the client device, and PQConnect's time-based key erasure achieves forward secrecy *without* new handshakes. Consequently, 45 servers (with separate PQConnect keys) will produce exactly 45 PQConnect tunnels.

Splitting 13.6MB across 45 PQConnect tunnels produces 0.3MB of data in an average tunnel, which is still below the 2MB cost of setting up the tunnel—but, again, the 0.3MB is for just five minutes of usage, whereas PQConnect tunnels can continue handling much more than five minutes of traffic. In just two hours of usage of the app (spread over whatever amount of real time) one expects 24 times as much traffic, presumably to the same set[5] of servers, at which point the 2MB cost of setting up each tunnel has to be compared to 7.2MB of traffic sent through each tunnel. The 7.2MB keeps growing with usage while the 2MB does not.

As another data point, [7, "Total Kilobytes" and "TCP Connections Per Page"] shows that the average web page has grown beyond 2MB and that the number of connections per web page has dropped to 10. Clickstream studies such as [51] show that, for most users, the majority of web-page visits are to the user's top 10 sites, and the number of servers visited is $12\times$ smaller than the number of visits.

A different way to understand the affordability of 2MB is to consider the estimate cited above of about 2^{-40} dollars to send a byte of data through the Internet, implying that sending 2MB costs about 2^{-19} dollars. For a user to spend a dollar on this would require the user to contact about 2^{19} servers. Of course, a smartphone is sometimes on a more expensive mobile network; if the mobile data plan has a 10GB-per-month data cap, then it is not possible for the phone to start PQConnect tunnels to more than 10000 different servers in a month without getting on WiFi.

C. Handshake Cost

After N PQConnect clients retrieve public keys from a PQConnect server, they carry out N handshakes to set up N tunnels to the server. The following paragraphs investigate the costs of these handshakes.

As in Section VI-A, we begin by reviewing relevant microbenchmarks. Ciphertexts are 194 bytes for mceliece6960119 and 1039 bytes for sntrup761, as mentioned in Section IV-E. (The reversal is not a typo: mceliece6960119 has a larger public key than sntrup761 but has smaller ciphertexts.) Regarding speeds on the same CPU described above, encapsulation takes 116636 cycles for mceliece6960119 and 41784 cycles for sntrup761; decapsulation takes 272042 cycles for mceliece6960119 and 61793 cycles for sntrup761; computing an X25519 shared secret takes 87876 cycles.

Each handshake involves, for the client, encapsulation for both mceliece6960119 and sntrup761, generating an X25519 key, and computing two X25519 shared secrets; the total of microbenchmarks is 334172 cycles. Each handshake also involves, for the server, decapsulation for both mceliece6960119 and sntrup761, and computing two X25519 shared secrets; the total of microbenchmarks is 509587 cycles. The ciphertext data communicated is $1039 + 194 + 32 = 1265$ bytes. (The full packet size is 1299 bytes; see Section IV-E for details.)

The dollar-cost estimates mentioned above imply that the handshake costs are, for each side, about 2^{-32} dollars for computation and about 2^{-30} dollars for communication.

Clients also precompute more mceliece6960119 ciphertexts in case they want to regenerate tunnels; caching some ciphertexts is less space than caching the public key. This increases the initial computation costs in the obvious way, depending on how many ciphertexts are computed.

D. Cost per Byte of User Data

Once a tunnel has been established, PQConnect uses authenticated encryption to protect the packets sent through the tunnel. This involves purely symmetric cryptography; there are no public-key operations here.

Regarding traffic, each packet expands by 56 bytes: a 2-byte message type, a 32-byte tunnel identifier, 6 bytes for the position within the tunnel, and a 16-byte authenticator. Average Internet packet sizes have been growing towards a kilobyte (for example, [41, Table 5] reported 870 bytes in 2021, while [39, Figure 1] showed less in 2007), so this 56-byte packet expansion adds only about 7% to traffic.

Regarding speed, [16] reports ChaCha20 running at, e.g., 1.23 cycles/byte on Skylake for 1536-byte packets and 1.71 cycles/byte for 576-byte packets; i.e., roughly 2^{-50} dollars/byte with the server hardware described in [11]. This is negligible cost on an absolute scale and compared to communicating the encrypted data. The same comment applies to Poly1305 and per-packet cryptographic operations such as ratcheting.

E. Costs for Non-PQConnect Devices

We close by considering the question of what costs are imposed on devices that do not support the protocol. Note that, when a protocol is new, there are many more devices *not* supporting the protocol than supporting the protocol.

PQConnect is designed to add zero costs for non-PQConnect servers. A PQConnect client does not incur or modify traffic to non-PQConnect servers. In particular, a client *passively* detects PQConnect support, rather than sending out probes to check for PQConnect support.

[5]Often it is useful to generate many names for the same server, but if they are sharing the same PQConnect public key then the public key does not need to be transmitted repeatedly. Note that [45] says "The Servers column indicates the number of unique server names".

There is, however, a tiny part of the PQConnect cost visible to non-PQConnect clients. Specifically, announcing a server's PQConnect support adds bytes in DNS responses to supply a CNAME to a server name that includes a 56-byte `pq1` announcement. (The number of added bytes will vary; in some cases the announcement will replace, rather than supplement, an existing server name and possibly a CNAME.) This sends extra bytes of data to each non-PQConnect client looking up that DNS record. It would not be surprising if this occasionally incurs extra packets, for example because of a DNS response having to be split across packets or because of higher packet-loss rates for larger packets.

VII. PQConnect Software

Our PQConnect software is available from [1]. The software package contains code for setting up and running PQConnect servers, keyservers, and clients. See Appendix A for further software considerations, Appendix B for measurements of network traffic and CPU time for end-to-end experiments with PQConnect, and Appendix F for instructions to reproduce this paper's main software claims.

Acknowledgments

Thanks to Jacob Appelbaum and Jon Solworth for their comments. This work was funded in part by the U.S. National Science Foundation under grant 2037867; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy–EXC 2092 CASA–390781972 "Cyber Security in the Age of Large-Scale Adversaries"; the European Commision through the Horizon Europe program under project number 101135475 (TALER); the Dutch Ministry of Education, Culture, and Science through Gravitation project "Challenges in Cyber Security - 024.006.037"; the Taiwan's Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP); and by the Academia Sinica Grand Challenge Seed Project AS-GCS-113-M07. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation" (or other funding agencies).

References

[1] PQConnect software. https://www.pqconnect.net/pqconnect-20241202.tar.gz, 2024.
[2] PQConnect handshake analysis. https://www.pqconnect.net/pqconnect-handshake-20241202.spthy, 2024.
[3] PQConnect measurements. https://www.pqconnect.net/pqconnect-experiment-graph-20241204.pdf, 2024.
[4] 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. DOI 10.1007/3-540-44448-3_42.
[5] David Adrian. Curve-popularity data?, 2024. https://mailarchive.ietf.org/arch/msg/tls/vWAEg7E3jeLZjLABVaMVLR0flX4/.
[6] Bharath Aiyer, Jeffrey Caso, Peter Russell, and Marc Sorel. New survey reveals $2 trillion market opportunity for cybersecurity technology and service providers, 2022. https://tinyurl.com/y8amhejx.
[7] HTTP Archive. State of the web, 2024. Accessed: 2024-12-04. https://httparchive.org/reports/state-of-the-web.
[8] Steve Babbage, Dario Catalano, Carlos Cid, Benne de Weger, Orr Dunkelman, Christian Gehrmann, Louis Granboulan, Tanja Lange, Arjen Lenstra, Chris Mitchell, Mats Näslund, Phong Nguyen, Christof Paar, Kenny Paterson, Jan Pelzl, Thomas Pornin, Bart Preneel, Christian Rechberger, Vincent Rijmen, Matt Robshaw, Andy Rupp, Martin Schläffer, Serge Vaudenay, and Michael Ward. ECRYPT2 yearly report on algorithms and keysizes (2008-2009). 2009. https://web.archive.org/web/20110527055638/http://www.ecrypt.eu.org/documents/D.SPA.7.pdf.
[9] Davide Balzarotti and Wenyuan Xu, editors. *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. https://www.usenix.org/conference/usenixsecurity24.
[10] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. DOI 10.1007/11745853_14.
[11] Daniel J. Bernstein. Predicting performance for post-quantum encrypted-file systems, 2024. https://cr.yp.to/papers.html#pppqefs.
[12] Daniel J. Bernstein. Quantifying risks in cryptographic selection processes, 2024. https://cr.yp.to/papers.html#qrcsp.
[13] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in Kyber implementations, 2024. https://eprint.iacr.org/2024/1049.
[14] Daniel J. Bernstein and Tung Chou. libmceliece: speed, 2024. Accessed: 2024-12-04. https://lib.mceliece.org/speed.html.
[15] Daniel J. Bernstein and Tanja Lange. McTiny: Fast high-confidence post-quantum key erasure for tiny network servers. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1731–1748. USENIX Association, 2020. https://www.usenix.org/conference/usenixsecurity20/presentation/bernstein.
[16] Daniel J. Bernstein and Tanja Lange (editors). Measurements of stream ciphers on one machine: amd64; Skylake (506e3); 2015 Intel Xeon E3-1220 v5; 4 x 3000MHz; samba, supercop-20240625. https://bench.cr.yp.to/results-stream/amd64-samba.html. Accessed: 2024-07-08.
[17] Daniel J. Bernstein and Kaushik Nath. lib25519: speed, 2024. Accessed: 2024-12-04. https://lib25519.cr.yp.to/speed.html.
[18] 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. DOI 10.1109/SP.2015.40.
[19] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: congestion-based congestion control. *Commun. ACM*, 60(2):58–66, 2017. DOI 10.1145/3009824.
[20] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. FaCT: A flexible, constant-time programming language. In *IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017*, pages 69–76. IEEE Computer Society, 2017. DOI 10.1109/SECDEV.2017.24.
[21] Stefania Cavallar, Bruce Dodson, Arjen K. Lenstra, Walter M. Lioen, Peter L. Montgomery, Brian Murphy, Herman J. J. te Riele, Karen I. Aardal, Jeff Gilchrist, Gérard Guillerm, Paul C. Leyland, Joël Marchand, François Morain, Alec Muffett, Chris Putnam, Craig Putnam, and Paul Zimmermann. Factorization of a 512-bit RSA modulus. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2000. DOI 10.1007/3-540-45539-6_1.
[22] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 164–178. IEEE Computer Society, 2016. DOI 10.1109/CSF.2016.19.
[23] 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 Com-*

puter and Communications Security, CCS '17, page 1773–1788, New York, NY, USA, 2017. Association for Computing Machinery. DOI 10.1145/3133956.3134063.

[24] Cas Cremers and Dennis Jackson. Prime, order please! Revisiting small subgroup and invalid curve attacks on protocols using Diffie-Hellman. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 78–93. IEEE, 2019. DOI 10.1109/CSF.2019.00013.

[25] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983. DOI 10.1109/TIT.1983.1056650.

[26] 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.

[27] Constantine Dovrolis and Matthew Roughan, editors. *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference, IMC 2007, San Diego, California, USA, October 24-26, 2007*. ACM, 2007.

[28] Huayi Duan, Marco Bearzi, Jodok Vieli, David A. Basin, Adrian Perrig, Si Liu, and Bernhard Tellenbach. CAMP: compositional amplification attacks against DNS. In Balzarotti and Xu [9]. https://www.usenix.org/conference/usenixsecurity24/presentation/duan.

[29] Shawn Embleton, Sherri Sparks, and Cliff Changchun Zou. SMM rootkit: a new breed of OS independent malware. *Secur. Commun. Networks*, 6(12):1590–1605, 2013. DOI 10.1002/SEC.166.

[30] SSL Ratios (public) - Mozilla Data Documentation. https://docs.telemetry.mozilla.org/datasets/other/ssl/reference. Accessed: 2024-12-04.

[31] 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. DOI 10.1007/3-540-45537-X_1.

[32] Fox IT. Deep dive into QUANTUM INSERT, 2015. https://blog.fox-it.com/2015/04/20/deep-dive-into-quantum-insert/.

[33] Phillipa Gill, John S. Heidemann, John W. Byers, and Ramesh Govindan, editors. *Proceedings of the 2016 ACM on Internet Measurement Conference, IMC 2016, Santa Monica, CA, USA, November 14-16, 2016*. ACM, 2016. DOI 10.1145/2987443.

[34] Sharon Goldberg, Miro Haller, Nadia Heninger, Mike Milano, Dan Shumow, Marc Stevens, and Adam Suhl. RADIUS/UDP considered harmful. In Balzarotti and Xu [9]. https://www.usenix.org/conference/usenixsecurity24/presentation/goldberg.

[35] Andy Greenberg. Leaked NSA doc says it can collect and keep your encrypted data as long as it takes to crack it. https://www.forbes.com/sites/andygreenberg/2013/06/20/leaked-nsa-doc-says-it-can-collect-and-keep-your-encrypted-data-as-long-as-it-takes-to-crack-it/.

[36] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. The SSL landscape: a thorough analysis of the x.509 PKI using active and passive measurements. In Patrick Thiran and Walter Willinger, editors, *Proceedings of the 11th ACM SIGCOMM Internet Measurement Conference, IMC '11, Berlin, Germany, November 2-, 2011*, pages 427–444. ACM, 2011. DOI 10.1145/2068816.2068856.

[37] 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. DOI 10.1109/SP40001.2021.00030.

[38] Sara Islet. Stupid TLS facts: TLS resumption, 2023. https://keymaterial.net/2023/04/23/stupid-tls-facts-tls-resumption/.

[39] Wolfgang John and Sven Tafvelin. Analysis of Internet backbone traffic and header anomalies observed. In Dovrolis and Roughan [27], pages 111–116. DOI 10.1145/1298306.1298321.

[40] Connor Jones. Cancer patient forced to make terrible decision after Qilin attack on London hospitals, 2024. https://www.theregister.com/2024/07/05/qilin_impacts_patient/.

[41] Piotr Jurkiewicz, Grzegorz Rzym, and Piotr Boryło. Flow length and size distributions in campus Internet traffic. *Computer Communications*, 167:15–30, 2021.

[42] Dusan Kostic, Hanno Becker, John Harrison, Juneyoung Lee, Nevine Ebeid, and Torben Hansen. Adoption of high-assurance and highly performant cryptographic algorithms at AWS, 2024. https://iacr.org/submit/files/slides/2024/rwc/rwc2024/38/slides.pdf.

[43] 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. DOI 10.1007/3-540-45537-X_17.

[44] Homin K. Lee, Tal Malkin, and Erich M. Nahum. Cryptographic strength of SSL/TLS servers: current and recent practices. In Dovrolis and Roughan [27], pages 83–92. DOI 10.1145/1298306.1298318.

[45] Dimitri Mankowski, Thom Wiggers, and Veelasha Moonsamy. TLS $\rightarrow$ post-quantum TLS: inspecting the TLS landscape for PQC adoption on Android. In *IEEE European Symposium on Security and Privacy, EuroS&P 2023 - Workshops, Delft, Netherlands, July 3-7, 2023*, pages 526–538. IEEE, 2023. DOI 10.1109/EUROSPW59978.2023.00065.

[46] 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. DOI 10.1007/3-540-45473-X_13.

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

[48] Daniel Marth, Clemens Hlauschek, Christian Schanes, and Thomas Grechenig. Abusing trust: Mobile kernel subversion via TrustZone rootkits. In *43rd IEEE Security and Privacy, SP Workshops 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 265–276. IEEE, 2022. DOI 10.1109/SPW54247.2022.9833891.

[49] 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. DOI 10.1007/978-3-642-39799-8_48.

[50] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF protocols. *RFC*, 8439:1–46, 2018. DOI 10.17487/RFC8439.

[51] Hartmut Obendorf, Harald Weinreich, Eelco Herder, and Matthias Mayer. Web page revisitation revisited: implications of a long-term click-stream study of browser usage. In Mary Beth Rosson and David J. Gilmore, editors, *Proceedings of the 2007 Conference on Human Factors in Computing Systems, CHI 2007, San Jose, California, USA, April 28 - May 3, 2007*, pages 597–606. ACM, 2007. DOI 10.1145/1240624.1240719.

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

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

[54] 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. DOI 10.1145/2508859.2516737.

[55] Julian Rauchberger, Robert Luh, and Sebastian Schrittwieser. Longkit - A universal framework for BIOS/UEFI rootkits in system management mode. In Paolo Mori, Steven Furnell, and Olivier Camp, editors, *Proceedings of the 3rd International Conference on Information Systems Security and Privacy, ICISSP 2017, Porto, Portugal, February 19-21, 2017*, pages 346–353. SciTePress, 2017. DOI 10.5220/0006165603460353.

[56] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018. DOI 10.17487/RFC8446.

[57] Michael C. Richardson. A method for storing IPsec keying material in DNS. *RFC*, 4025:1–12, 2005. DOI 10.17487/RFC4025.

[58] Christian Rossow. Amplification hell: Revisiting network protocols for DDoS abuse. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. https://www.ndss-symposium.org/ndss2014/amplification-hell-revisiting-network-protocols-ddos-abuse.

[59] Sandvine. The global Internet phenomena report: January 2023, 2023. https://www.sandvine.com/hubfs/Sandvine_Redesign_2019/Downloads/2023/reports/Sandvine%20GIPR%202023.pdf.

[60] 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. DOI 10.1109/CSF.2012.25.

[61] 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. DOI 10.1145/3372297.3423350.

[62] 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. DOI 10.1007/978-3-642-25405-5_4.

[63] Laurent Simon, David Chisnall, and Ross J. Anderson. What you get is what you C: controlling side effects in mainstream C compilers. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 1–15. IEEE, 2018. DOI 10.1109/EUROSP.2018.00009.

[64] Martin Thomson. Curve-popularity data?, 2024. https://mailarchive.ietf.org/arch/msg/tls/pQRDJ9MBwnmLHp86Zvs_CfYUFaY/.

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

[66] Veracode. Harden TLS session resumption, 2024. https://docs.veracode.com/r/harden-tls-session-resumption.

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

[68] Mullvad VPN. Stable quantum-resistant tunnels in the app!, 2023. https://mullvad.net/en/blog/stable-quantum-resistant-tunnels-in-the-app.

[69] Bas Westerbaan. Curve-popularity data?, 2024. https://mailarchive.ietf.org/arch/msg/tls/lWh_uimMIgQ6SMV_BSkJDh34eQM/.

[70] Bas Westerbaan. "One in every two thousand CPU cycles of Meta is used for X25519" (tweet), 2024. https://twitter.com/bwesterb/status/1771958142147973390.

Appendix A
PQConnect Software – Language Considerations

Our software is several thousand lines of Python. This language choice imposes performance costs. One experiment showed that the software (running on one core of the Skylake CPU described in Section VI) was able to handle slightly above 20Mbps, which is acceptable for some environments but not others. We already call C libraries for all of the cryptographic computations (libmceliece for mceliece6960119, libntruprime for sntrup761, lib25519 for X25519, and libsodium for symmetric cryptography) but moving more of the packet processing into C will be beneficial.

Using high-level languages also raises security concerns. Avoiding timing attacks is already tricky in C (see, e.g., [13]) and becomes more difficult in higher-level languages. The same comment applies to securely erasing keys for forward secrecy. Responses include rewriting software that handles secrets in assembly language (see, e.g., [42]) and teaching higher-level languages about secrets (see, e.g., [20] and [63]).

What drove our choice of Python was that Python allows rapid development of software with performance that *often* suffices. Recall from Section I that the goal here is to expand the usage of post-quantum cryptography as rapidly as possible; having Python software already available for the situations where it suffices is better than having all users wait for higher-performance software.

Appendix B
Experiments

We considered, for various values of x, the total cost of creating a tunnel and using the tunnel to transmit x bytes of user data from our `www.pqconnect.net` server in Europe to a client in the US using our Python implementation of PQConnect. Speeds of other connections will vary depending on the latency and bandwidth between clients and servers. Note that PQConnect tunnels, like VPN tunnels, can last any amount of time, so very large values of x are of interest.

To ensure that all PQConnect costs are included, we restarted the PQConnect client software, forcing new downloads of the server keys along with creation of a new tunnel. We then ran `curl` on the client (an Intel Core i7-1370P) to download an x-byte file via HTTPS from the server (see, e.g., https://www.pqconnect.net/bytes/16777216). We monitored network traffic using `tcpdump`. We ran one experiment at each size. The network was not idle, so slight variations are unsurprising.

The resulting measurements are graphed in [3]. As expected, the number of bytes sent from the server to the client for $x = 2^{30}$ is slightly above 2^{30} (because of TCP packet overhead and PQConnect packet overhead), while the number of bytes sent from the client to the server is a fraction of this for TCP acknowledgments. The left side of the graph in [3] shows, independently of x, the expected initial 2MB of traffic.

Appendix C
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 do not 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 is more than twice as fast as decapsulating a mceliece6960119 ciphertext; compare, e.g., [17] and [16]. 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.

As another example, consider the scenario of pre-quantum attackers passing 2^{128} operations before quantum computers are built. Breaking the long-term X25519 keys for t servers is only about $\sqrt{t}$ times as much work as breaking a single key [43]: for example, only about 100 times as much work for $t = 10000$. Putting X25519 in the outer layer would allow such attackers to peel back this layer on all handshakes at once for many servers.

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. The "DOOM" (decoding one out of many) attacks from [62] reduce the cost of attacking many ciphertexts, but they do not recover all plaintexts encapsulated to the same public key for free. Furthermore, mceliece6960119 is far beyond a 2^{128} security level to start with.

Taking the above considerations into account, the nesting order for PQConnect, from outermost to innermost, is Classic McEliece (long-term) $\rightarrow$ Curve25519 (long-term–ephemeral) $\rightarrow$ X25519 (ephemeral–ephemeral) $\rightarrow$ Streamlined NTRU prime (ephemeral). This places the oldest and most confidently quantum-resistant primitive as the first line 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. As an example of how this nesting reduces the attack surface, a bug in the client's X25519 software that leaks secret client memory through X25519 public keys is not a problem here: the attacker cannot see the client's ephemeral X25519 public key without first recovering the secret McEliece key.

Appendix D
ChaCha20-based KDF

PQConnect uses a deterministic Key Derivation Function (KDF) for the following purposes: during the handshake, deriving a 32-byte session key from two 32-byte session keys (such as a mceliece6960119 session key and an X25519 session key); later in the handshake, deriving three 32-byte quantities `tunnelID`, $\mathtt{T_c}$, $\mathtt{T_s}$ from a 32-byte session key and a ciphertext hash; in the key ratchet, deriving a 32-byte next-epoch key and a 32-byte chain key from a 32-byte root key; also in the key ratchet, deriving many 32-byte packet keys and a new 32-byte chain key from a 32-byte chain key; and deriving a new 32-byte cookie key from a previous cookie key and a random input.

Recall from Section IV-F that there is a new packet key for each packet sent or received. A slow KDF might raise concerns regarding per-packet costs in the key ratchet. To address such concerns, PQConnect reduces the CPU cost of key derivation in two ways. First, PQConnect makes an efficient choice of KDF; the KDF is defined below. Second, PQConnect batches packet-key derivation (the word "many" in the previous paragraph) to amortize the number of cycles used per byte of KDF output.

The KDF is based on ChaCha20. ChaCha20 maps a 32-byte key and a 16-byte input to a 64-byte output block. Typically the 16-byte input is viewed as a concatenation of a nonce and a counter, so ChaCha20 is a stream cipher producing many output blocks from each nonce.

The KDF works as follows. It takes as input (1) an integer n, (2) a secret key k, and (3) optionally an additional 32-byte value i, which may or may not be secret. It outputs n 32-byte keys derived from k (and from i when i is present). KDF usage follows three restrictions: first, the value n is always small; second, if i is provided and $n > 2$ then i is obtained as a hash; third, each key k is used in the KDF in only one way, and in particular if it is used with i absent then it is not reused with i present.

If i is absent (this is the situation for the key ratchet), then the KDF is simply the ChaCha20 stream cipher on nonce 0 and counter 0, used to produce $32n$ bytes of output (which are then viewed as n separate 32-byte keys). If ChaCha20 meets its PRF goal then these $32n$ bytes are indistinguishable from random. This also implies infeasibility of recovering the original key from these $32n$ bytes.

If the 32-byte input i is present, the KDF first divides i into two 16-byte chunks, i_0 and i_1, and derives a subkey k' from the first 32 bytes output when ChaCha20 is initialized with key k and input i_0. It then re-initializes the ChaCha20 stream cipher with key k' and counter/nonce i_1, and returns $32n$ bytes of output. The use of subkeying is similar to the way XChaCha20 uses a 192-bit nonce, but for simplicity uses ChaCha20 rather than HChaCha20 for the subkey computation.

The security analysis when i is present relies on more than the PRF property of ChaCha20. For $n > 2$, there are multiple ChaCha20 output blocks under key k' using inputs $i_1, i_1 + 1, \ldots$; the KDF results for (k, i_0, i_1+1) thus overlap the KDF results for (k, i_0, i_1). Recall that i is required to be generated as a hash in this case; hash inputs producing small-difference outputs would be an example of "near-collisions", a standard topic of hash-function cryptanalysis.

A separate issue is that, inside a forward-secrecy analysis, one wants to know that a key derived from a secret i is secret even when the first key k is not secret. Here a PRF assumption at a 256-bit security level does appear sufficient, as long as k is approximately uniformly distributed. The point is that if such a key k has a noticeable probability of mapping 32-byte inputs i to, e.g., only 2^{200} possible 32-byte outputs, then a secret key k would also have approximately that probability, which would allow a time-2^{200} PRF attack that looks for colliding values of i.

Appendix E
The Tamarin Prover

We give a brief introduction to Tamarin and its semantics.

The Tamarin prover is a formal-verification tool for proving properties of cryptographic protocols, such as confidentiality, peer-to-peer authentication, and forward secrecy [60]. It has been used to analyze security properties of widely deployed cryptographic network protocols, such as TLS 1.3 [23] and the WireGuard protocol handshake [26]. Protocol properties in Tamarin are proven (or disproven) in the Dolev–Yao model; the adversary has full power to eavesdrop, intercept, modify, and insert messages into the channel [25]. 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.

A. 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. 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) ] // ! says persists
  --[ 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 in 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.

C. 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 use the following 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 knows 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 [60]. When a lemma is intended to prove a universally quantified statement $\forall_x P(x)$ 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. Statements in first-order logic are undecidable in general, so there exist lemmas for which the algorithm will not terminate. However, when the prover does terminate, it either proves the statement to be true over unbounded executions or derives a contradiction.

Appendix F
Artifact Appendix

This appendix explains in detail how to use our PQConnect software to check the main software-related claims in this paper. The primary goal is to see the PQConnect client software transparently protecting connections to an existing PQConnect server while continuing to allow connections to non-PQConnect servers. A stretch goal (which has also been tested by reviewers) is to similarly see the PQConnect server software in operation.

This appendix also explains how to verify the security proofs from Section 5 of the paper using Tamarin.

A. Description & Requirements

Here is what you need to recreate a setup suitable for carrying out the experiments below.

1) How to access: https://doi.org/10.5281/zenodo.14253085 is a permanently archived version of the PQConnect software, `pqconnect-20241202.tar.gz`.

2) Hardware dependencies: We expect that approximately 100% of laptops, desktops, and servers are capable of running the PQConnect software. Example: we rechecked all steps below on a computer named `jasper3`, which has a 4-core 2GHz Intel Celeron N5105 CPU with 8GB of RAM.

3) Software dependencies: Our PQConnect installation scripts currently support Arch, Debian (including variants such as Raspberry Pi OS and Ubuntu), and Gentoo. Example: `jasper3` runs Debian 11. Adjusting the scripts for another GNU/Linux distribution should be straightforward.

The stretch goal of trying the server software (not just the client software) requires you to have a second computer already working as a server (as a baseline for comparison to what happens when PQConnect is installed), and requires the ability to modify DNS entries for the server.

4) Benchmarks: The scope of this artifact appendix does not include experimental cost measurements. The cost analysis in the main body of this paper (as opposed to Appendix B) is a manual analysis of bottlenecks from first principles.

B. Artifact Installation & Configuration

As root, in `bash`, in the `/root` directory, download `pqconnect-20241202.tar.gz`.

Unpack into `/root/pqconnect-20241202` and switch to that directory:

```
tar -xf pqconnect-20241202.tar.gz
cd pqconnect-20241202
```

Install (this uses under 3GB of disk space for some OS packages, some further libraries, `/home/linuxbrew`, `/home/tamarin`, and `/etc/pqconnect`):

```
./install-pqconnect # jasper3: 390 seconds
./install-tamarin # jasper3: 492 seconds
```

Optionally, install a network-packet sniffer such as `tcpdump`. (Checking cryptographic security is outside the scope of this appendix, but a sniffer shows the difference between a readable HTTP connection outside PQConnect and a not-obviously-readable HTTP connection protected by PQConnect.)

For the stretch goal of trying the server software, repeat the above steps on your existing server. Then create a PQConnect key for the server:

```
cd /root/pqconnect-20241202
./create-first-server-key
```

This also prints out instructions for announcing the server key in DNS. Follow those instructions.

C. Major Claims

The paper's major software-related claims are as follows, demonstrated by the experiments below with the same numbers:

- (C0): The lemmas in `handshake.spthy` pass verification by Tamarin.
- (C1): The client software recognizes PQConnect servers and creates end-to-end tunnels to those servers, with no per-server configuration on the client.
- (C2): The client software works with a wide range of existing applications, with no changes to the application software.
- (C3): The client software continues to allow connections to non-PQConnect servers.
- (C4): The server software accepts end-to-end tunnels from PQConnect clients, with no per-client configuration on the server.
- (C5): The server software works with a wide range of existing applications, with no changes to the application software.
- (C6): The server software continues to accept connections from non-PQConnect clients.

D. Evaluation

This section explains how to carry out experiments with the PQConnect software.

1) Experiment (E0) for Claim (C0): running Tamarin-Prover with the included PQConnect model and lemmas.

[Preparation] None.

[Execution] Load and verify the proofs using Tamarin:

```
cd /root/pqconnect-20241202
./run-tamarin # jasper3: 7 seconds
```

[Results] Tamarin will prove the three lemmas. The final section of output will show that all three lemmas have indeed been verified:

```
0_RTT_executable (exists-trace):
  verified (13 steps)
0_RTT_FS_confidential (all-traces):
  verified (24 steps)
responder_client_auth (all-traces):
  verified (7 steps)
```

2) Experiment (E1) for Claim (C1): seeing the client software creating an end-to-end tunnel to a PQConnect server.

[Preparation] Start the PQConnect client software:

```
cd /root/pqconnect-20241202
./run-client-verbose &
```

This continues to run, while saving (initially) 6 lines in a new file `pqconnect-log`, the last line ending `Listening on port 42423`. This also creates a `pqccli0` network interface.

Optionally, run `tail -f pconnect-log` in another terminal. You can also look later at `pqconnect-log`.

Optionally, start a sniffer for packets involving IP address 131.155.69.126 (our `pqconnect.net` server), saving results in a file to peruse later. For example, if your external network interface is named `enp3s0`:

```
tcpdump -Xlnei enp3s0 host 131.155.69.126 \
> tcpdump-log &
```

Later, when the experiment is done, stop the client software by killing the `run-client-verbose` job, and stop the sniffer if you started it.

[Execution] While PQConnect is running, look up the address of `www.pqconnect.net`:

```
dig www.pqconnect.net
```

[Results] The `dig` command will show a 10.* address instead of 131.155.69.126. The `pqconnect-log` file will show various lines for a key exchange.

3) Experiment (E2) for Claim (C2): seeing the client software protecting unmodified application software.

[Preparation] Start (and later stop) the PQConnect client software as above.

[Execution] While PQConnect is running, use `wget` to retrieve a web page via HTTP:

```
wget -O test.html \
http://www.pqconnect.net/test.html
```

HTTPS also works here, but HTTP is more interesting if you're sniffing the network.

[Results] The `pqconnect-log` file will show various "Message sent" and "Message received" lines for incoming and outgoing PQConnect packets. The `test.html` file will include "Looks like you're connecting with PQConnect. Congratulations!" If you're sniffing the network: the sniffer output will show UDP packets to and from port 42424 of 131.155.69.126 instead of unencrypted HTTP packets over TCP to and from port 80.

4) Experiment (E3) for Claim (C3): seeing the client software continuing to allow connections to non-PQConnect servers.

[Preparation] Start (and later stop) the PQConnect client software as above.

[Execution] While PQConnect is running, use `wget` to retrieve a web page via HTTP from `testwithout.pqconnect.net`:

```
wget -O test.html \
http://testwithout.pqconnect.net/test.html
```

Optionally, try your favorite tests of normal network operation with other servers.

[Results] The `pqconnect-log` file will not show any PQConnect packets for the `wget` run. The `test.html` file will include "Looks like you aren't connecting with PQConnect." If you're sniffing the network: the sniffer output will show unencrypted HTTP to and from port 80 of 131.155.69.126, including a server-name indication from the client ("Host: testwithout.pqconnect.net") and the web page from the server.

5) Experiment (E4) for Claim (C4): seeing the server software accepting an end-to-end tunnel from a PQConnect client.

[Preparation] On your client computer, start the PQConnect client software as above. On your server computer, start the PQConnect server software:

```
cd /root/pqconnect-20241202
./run-server-verbose &
```

Later stop the PQConnect client software as above, and similarly stop the PQConnect server software by killing the `run-server-verbose` job.

[Execution] On your client computer, use `dig` to look up the address of your server, while PQConnect is running on both computers.

[Results] The `dig` command will show a 10.* address instead of the actual server address. The `pqconnect-log` files on both computers will show various lines for a key exchange.

6) Experiment (E5) for Claim (C5): seeing the server software protecting unmodified application software.

[Preparation] Start (and later stop) the PQConnect client software and server software as above.

[Execution] On your client computer, use whatever tools you would normally use to contact your server, while PQConnect is running on both computers.

[Results] The client-server connection will function normally. The `pqconnect-log` files (and sniffer results if you are running a sniffer) will show that the connection is being tunneled via PQConnect.

PQConnect is not able to intercept *all* applications; if you are trying an application that dodges PQConnect, just try another application (and please let us know what didn't work).

7) Experiment (E6) for Claim (C6): seeing the server software continuing to allow connections from non-PQConnect clients.

[Preparation] Start (and later stop) the PQConnect server software as above.

[Execution] From any client machine not running PQConnect, connect to the server as usual, while PQConnect is running on the server.

[Results] The client-server connection will function normally, and the server's `pqconnect-log` file will remain idle.