PQConnect

An Automated Boring Protocol for Quantum-Secure Tunnels

Levin, Jonathan

Award date:
2021

PQConnect: An Automated Boring Protocol for Quantum-Secure Tunnels

Jonathan Levin

August 17, 2021

Contents

Acknowledgments

I would like to take this opportunity to thank a number of people, without whose support this thesis would not have been nearly as successful or as much fun to work on. I must start by expressing my extreme gratitude to Tanja Lange who supervised me on this project. She has been a wonderful mentor since I became her student and has given me many opportunities to learn as much as I can about cryptography. Because of her I was able to attend summer school in Croatia and come to Taiwan to complete my degree. She has been academically and personally supportive to an extent that is difficult to describe, but that I appreciate immensely.

I would like to say thank you to Boris Škorić and Bo-Yin Yang for taking the time to serve on my committee and give their feedback on my work. Bo-Yin was also instrumental in bringing me to Taiwan to write my thesis and has been very helpful throughout my stay. Given the pandemic and the difficulty in many places of collaborating with others in person, having the opportunity to come here was a real privilege.

I want to extend my thanks as well to Daniel J. Bernstein, both for planting the `PQConnect` seed that made this project possible and also for answering many of my questions.

Tung Chou showed a lot of interest in this project and took time to give helpful feedback on early drafts of my thesis. Additionally, he always has really good questions that lead to useful and enjoyable discussions.

Thank you to Matthias and Lorenz for being great office and lunch companions and for occasionally letting me spam you with my frustration about various things I could not get to work.

This thesis would have been much harder to complete without the constant supply of incredibly delicious vegetarian food cooked by the friendly old woman who runs my favorite Buddhist takeout place in Xizhi. Sadly, I do not know her name.

Finally, I am fortunate to have very loving parents and friends, many of whom I have not been able to see in a long time. I hope we can all be together again in person soon.

Chapter 1

Introduction

Transport Layer Security (TLS) is the predominant cryptographic protocol for encrypting and authenticating Internet traffic at the transport layer of the network stack. Unfortunately, every public key cryptosystem used in TLSv1.3 is broken by a quantum computer with enough qubits to run Shor's Algorithm [Sho94]. While no such quantum computer is known to exist, a reasonable assumption is that any attacker possessing one would keep its existence a secret.

In [Mos18], Michele Mosca straightforwardly argues that if x is time before post-quantum cryptography is widely deployed, and y is the duration you need information to be kept secret starting from the time of its creation, then if a quantum computer materializes before the moment $x + y$, we have a crisis. It becomes immediately obvious then that widely deploying post-quantum public key cryptography is a high priority if we want to ensure protection against future quantum attacks.

In the last decade there have been significant efforts to research the efficiency and security of post-quantum systems and on how to secure existing cryptographic infrastructure and protocols, including TLS, against quantum attacks (see, e.g., [SSW20] [Bos+15] [Hül+21]). Getting these updates implemented and deployed quickly, however, is challenging. The process for updating existing standards like TLS is historically a very slow one that has allowed broken algorithms to remain included for backwards compatibility purposes, long after attacks against them were published. Some of the first attacks against RC4, for example, were published already in 2001 ([FMS01], [MS01]), but RC4 was not completely removed from TLS until version 1.3 in 2018 [Res18], 17 years later.

Putting the issue of deployment speed aside for a moment, there are also noteworthy gaps in cryptographic protection on the Internet that TLS is incapable of addressing. For example, while TLS encrypts data above layer 4 of the OSI stack, it does not protect the confidentiality of transport header information such as source and destination ports, which can reveal information to eavesdroppers about the kinds of information being sent. There are also widely deployed network protocols, such as DNS, with *no* current cryptographic protections.

These problems together point to the need for new cryptographic network protocols that can provide post-quantum security, can be deployed quickly, and that add security properties to Internet traffic that do not exist with TLS.

1.1 Tunneling Protocols and VPNs

Protocols that provide cryptographic protections to transport headers must logically perform their encryption below the transport layer. One way to do this is through the use of tunneling protocols that have encrypted payloads. A tunneling protocol is a protocol that encapsulates one network packet as the payload of another packet, allowing machines in different networks (connected by the outer packet) to communicate as if they were part of the same virtual network (inner packet). When the inner packet is encrypted, the tunneling protocol becomes a virtual private network, or VPN. Some common VPN protocols include OpenVPN (`https://www.openvpn.net`), IPSec (`https://datatracker.ietf.org/wg/ipsec/documents/`), and more recently WireGuard [Don17]. The "private" in VPN refers to the fact that the inner packet is encrypted, and so anyone outside the VPN cannot see how packets are being routed or read their contents. Furthermore, because VPNs (usually) encrypt the entire inner IP packet, they provide cryptographic protection for all traffic within the network, not just a subset that supports TLS.

VPNs have two major use cases. One is connectivity and firewall traversal: company networks often use VPNs to connect their internal network to the internet, and many people who work from home must use VPN software to access their corporate network. The second use case is privacy against local attackers such as ISPs: Because packets can be encrypted and encapsulated, only the VPN server and subsequent Internet hops can see the destination and content of the inner packet, but they cannot link it to its origin. Without doing traffic analysis, an outside observer can only see that a person is communicating with a VPN provider, but not what was communicated.

One limitation of VPNs, however, is that they require knowing everyone on the private network in advance. With WireGuard, for example, tunnels between two peers are created using a single round trip time (1-RTT) handshake based on each peer's public keys and an optional pre-shared secret key. How these keys are exchanged is outside the scope of the protocol, however. Another is that VPN tunnels are frequently not end-to-end. That is, Party A wants to connect to party B through VPN V. Rather than forming a tunnel directly with B, A forms a tunnel with V, and V forwards communication between A and B.

1.2 `PQConnect`: A Boring Private Network

The requirement of knowing in advance the public keys of everyone you wish to talk to on the Internet makes existing VPNs impractical to deploy for general Internet traffic. However, if there were a way to establish encrypted tunnels with strangers, the same way that TLS facilitates establishing encrypted sessions, then this would solve our problem. We need then a protocol to bore encrypted tunnels: a boring private network (BPN).

In 2018, the PQCRYPTO project published the document "D 2.5 Internet: Integration", which provides a brief outline of a protocol called `PQConnect` that functions as a BPN [Ber18]. Unlike TLS, `PQConnect` operates at layer 3 of the network stack, establishing an authenticated, encrypted tunnel between communicating nodes on a network, and using post-quantum cryptography for key agreement. `PQConnect` should create tunnels automatically between clients and servers using long-term public keys stored in a public directory, such as a server's DNS record. Further, `PQConnect` can be deployed on the existing Internet. Peers that support `PQConnect` discover each other automatically and transparently establish an encrypted tunnel

through which they communicate.

Automatically creating tunnels using post quantum cryptography poses some new challenges. Due to the young age of many post-quantum schemes, the difficulty of their underlying hardness assumptions is very much an open area of research. Post-quantum cryptography also does not provide a 1-to-1 replacement for current schemes, either. For example, aside from cryptography based on isogenies between supersingular elliptic curves, there is not currently a post-quantum equivalent to Diffie-Hellman (DH) key exchange [Cas+18]. The security of isogenies is also not well understood yet, and their performance is quite slow. Establishing a shared key using key encapsulation mechanisms (KEMs) seems then like the obvious alternative. However, all KEMs have significantly larger key sizes than and perform more slowly than `x25519` elliptic curve Diffie-Hellman (ECDH).

Key size needs to be taken into account both before and during key agreement. To achieve perfect forward secrecy, fresh ephemeral keys must be generated and/or exchanged for each connection. If the key agreement phase of the protocol requires sending a new 1 MB `mceliece6960119` [Alb+20] public key every time a new tunnel is created, the protocol might be too slow to be used in practice. KEMs with smaller keys, such as `sntrup4591761` [Ber+20], could be used instead, for example, but this also means that the security of `PQConnect` potentially depends on the security of multiple post-quantum schemes.

Adding long-term post-quantum public keys to DNS records also potentially creates issues in performance. Size limits of both records and DNS caches can mean that storing and returning large public keys could cause performance penalties during DNS lookup requests.

Lastly, network protocols are subject to many different availability attacks. The larger the size of a protocol message and the more expensive the computation needed to generate it, the easier it is potentially to exhaust the resources of an endpoint, resulting in denial of service (DoS). To make sure that `PQConnect` is usable, these concerns must be addressed so that the cost of attacking the protocol is at least the cost of the necessary defense.

1.3 Contributions of this Thesis

This thesis takes the sketch of `PQConnect` provided in [Ber18] and develops the idea into a complete protocol. In chapter 2, I provide an introduction to the main ideas from cryptography and networking used in the protocol. Next, chapter 3 gives a detailed overview of the `PQConnect` protocol, including key distribution, the handshake that two parties perform to establish a shared secret, and symmetric cryptography and key management within the tunnel itself. Along the way I add many details to the outline of `PQConnect` provided in [Ber18]. In chapter 4 I discuss further improvements to the implementation of `PQConnect` that go beyond the basic protocol functionality described in chapter 3. Next, in chapter 5, I use the TAMARIN prover to formally verify that the `PQConnect` handshake achieves important security properties. Finally in chapter 6 I discuss the proof of concept (PoC) for `PQConnect` that I implemented and the steps required for widespread deployment.

Chapter 2

Cryptographic and Networking Preliminaries

This chapter provides background on cryptographic schemes and security notions used in `PQConnect`, focusing in particular on properties of key encapsulation mechanisms (KEMs) used during the `PQConnect` handshake. It also introduces the reader to networking concepts important to the design of `PQConnect`.

2.1 Cryptography

In this section I offer a brief introduction to the main cryptographic functions and algorithms that the reader will encounter in this thesis.

Key Encapsulation Mechanism

Public key cryptosystems are frequently inefficient at encrypting large amounts of data. For example, the RSA cryptosystem encrypts (resp. decrypts) data by first encoding it as an integer in a ring modulo the product of two large primes and then exponentiating it in the ring to the user's public (resp. private) exponent. Not only is it computationally expensive to exponentiate large numbers, the size of the modulus also limits the bit length of the data being computed on. In practice, RSA is frequently used as a hybrid encryption scheme, whereby it outputs the ciphertext c of a random message m, and the symmetric key $k = \text{KDF}(m)$, for some key derivation function KDF, is then used with a faster symmetric encryption scheme to encrypt the original data. To perform decryption, the recipient uses RSA to recover m, and then applies the KDF to retrieve k. The fast symmetric scheme makes this a much more computationally efficient method for encrypting large amounts of data than by using public key cryptography alone.

A public key cryptosystem designed specifically to encrypt symmetric keys in this fashion is referred to as a key encapsulation mechanism (KEM). A KEM is defined as a three-tuple of algorithms KEM = (`KeyGen, Encapsulate, Decapsulate`).

- `KeyGen` is a probabilistic algorithm that takes as input a security parameter λ and outputs a public-private keypair (pk, sk).

- `Encapsulate` is a probabilistic algorithm that takes a public key pk as input and outputs the pair $(k, c) \in \mathcal{K} \times \mathcal{C}$: a random key k from key space $\mathcal{K}$ and a ciphertext c, the encapsulation of k, from ciphertext space $\mathcal{C}$.
- `Decapsulation` is a deterministic algorithm that takes a secret key sk and encapsulated key c and outputs symmetric key k or failure symbol $\perp$.

Simple Symmetric Primitives

A **hash function** is a function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^n$ that takes arbitrary bit strings and maps them to a fixed-length output. A trivial hash function is the function that takes an input string and outputs its first n bits if the bit-length of the input is longer than n, and otherwise outputs the input padded by zeroes until it is n-bits long. In cryptography, however, we want hash functions that make it difficult to learn about the input, given only the function's output. A **cryptographic hash function** is a hash function that also has at least the following three properties:

1. Pre-image resistance. Given a hash function H and an output of the hash y, an attacker should not be able to compute a value x such that $H(x) = y$.
2. Second pre-image resistance. Given an output y of a hash function and a fixed given input x with $H(x) = y$, an attacker should not be able to find another $x' \neq x$ such that $H(x') = y$.
3. Collision resistance. It should be not be possible for an attacker to compute any two values x, x' with $x \neq x'$, such that $H(x) = H(x')$.

Often in cryptography we want to generate long streams of random-looking bits. The classical use case for this is the one-time pad symmetric encryption scheme, which takes an n-bit message m and an n-bit (truly) random key k and encrypts m under k to generate the ciphertext $c = m \oplus k$, where $\oplus$ is bitwise xor. The recipient who knows k can then decrypt c by computing $m = c \oplus k$.

While the one-time pad is information-theoretically secure (for any n-bit message m' there exists a key k' such that $c = m' \oplus k'$), it requires truly random keys that are as long as the message itself. In practice, we can achieve a computationally indistinguishable level of security using pseudorandomness. That is, using bit-strings that are not truly random but are generated from a short random seed.

A **pseudo-random generator** (PRG) is a function $\mathcal{G} : \{0,1\}^n \to \{0,1\}^{\ell(n)}$, with $\ell(n) > n$, such that any attacker trying to distinguish the output of $\mathcal{G}$ from a truly random $\ell(n)$-bit string has negligible advantage. A **stream cipher** is a PRG that is used to generate a long stream of pseudo-random bits, which are then used the same way as the key of the one-time pad for encrypting and decrypting messages.

The security guarantees of PRGs and stream ciphers imply that output bits are independent of each other. This property is important as it is relied on in the key derivation function (KDF) I use in `PQConnect`, which is based on the stream cipher `ChaCha20`.

Authenticated Encryption (with Associated Data)

When communicating with someone over an untrusted network, we frequently want our communication to be both confidential and authenticated. That is, we want the intended recipient

of our message to be the only one who can read it, and we want to also know that the messages sent to one another have not been changed by a third party in a way that we cannot detect. A so-called authenticated encryption (AE) scheme provides both of these properties. When sending a message, the message is both encrypted to ensure confidentiality, and it is also accompanied by an extra piece of information—a tag—that allows the recipient to verify its authenticity.

In some situations, we want to not only be able to authenticate our encrypted message, but also authenticate associated data that accompanies the message in plaintext. In a networking context this data could consist of header fields that are required for proper message routing or that are needed by the recipient to identify the correct decryption key, for example. An AE scheme that also supports authenticating unencrypted associated data is called an Authenticated Encryption with Associated Data (AEAD) scheme, and it consists of a three-tuple of algorithms (`Encrypt, Verify, Decrypt`).

- `Encrypt`. Given a key space $\mathcal{K}$, nonce space $\mathcal{N}$, message space $\mathcal{M}$, associated data space $\mathcal{AD}$, ciphertext space $\mathcal{C}$ and authentication tag space $\mathcal{T}$, `Encrypt` is a map $\mathcal{E} : \mathcal{K} \times \mathcal{N} \times \mathcal{M} \times \mathcal{AD} \to \mathcal{C} \times \mathcal{T}$ that takes a key `k`, nonce `n`, message `m`, and optional associated data `d`, and generates a ciphertext `c` and an authentication tag `t`, with $\texttt{Encrypt}(\texttt{k}, \texttt{n}, \texttt{m}, \texttt{d}) = (\texttt{c}, \texttt{t})$. In this thesis I will frequently abbreviate $(\texttt{c}, \texttt{t})$ as $\texttt{c}^*$.

- `Verify` takes a ciphertext `c`, tag `t`, key `k`, nonce `n`, and associated data `d`, and returns True if `t` is a proper authentication of `c` and `d` under `n` and `k`. Otherwise `Verify` returns False.

- `Decrypt` takes a ciphertext `c`, tag `t`, key `k`, nonce `n`, and associated data `d`, and returns m if and only if $\texttt{Verify}(\texttt{k}, \texttt{n}, \texttt{c}, \texttt{t}, \texttt{d}) = \text{True}$ and $\texttt{Encrypt}(\texttt{k}, \texttt{n}, \texttt{m}, \texttt{d}) = (\texttt{c}, \texttt{t})$.

2.2 Networking

To make later sections on server discovery, key distribution, and encapsulation clearer for the reader, I will give a brief overview in this section of DNS and the OSI model of the Internet, trying my best to include only the most relevant details.

DNS

The Domain Name System (DNS) is a decentralized infrastructure and set of protocols intended to ease navigating the internet. It does this by associating strings that are easy for humans to remember like "`www.zombo.com`" with IP addresses that are harder to remember like 50.28.52.163. Its function is much like a telephone book for the entire telephone network that associates human names with phone numbers.

DNS is organized hierarchically into a tree structure, with this hierarchy corresponding to the structure of the domain names themselves, called the domain name space. For example, the domain name "`www.zombo.com`" contains three labels: "www", "zombo", and "com". The latter, "com" is a top-level domain (tld). Every address ending with "com" is a child of this node in the domain name space. The domain "zombo" is then one of these children. Finally, the subdomain "www" is a child node of "zombo". The full tree of all names in the domain name space is divided into so-called zones of authority. Each zone is managed by a name server which contains DNS records for the domains in its zone.

When a user wishes to obtain the IP address of a website like "www.zombo.com", their browser will send a DNS query to a DNS name server that it knows, asking for an address record (A record) containing the IP address associated with the name "www.zombo.com". An A record simply matches this name to an IP address. If the name server has this record, it will return it to the user. If it does not, then it will ask a ".com" name server of the DNS tree for the A record or for a referral to another authoritative name server lower in the tree, and repeat this process until it obtains an A record. For example, the ".com" name server will refer the requester to the authoritative name server for "zombo.com" (which happens to be "ns.liquidweb.com"). The authoritative name server then returns the A record to the user, containing the website's IP address. User-facing DNS name servers will frequently cache records for a period of time, so that they can avoid performing lookups for the same record repeatedly in a short span of time.

OSI Model and Networking Layers

If two end points on a large network like the Internet wish to communicate with one another, they depend on a large number of intermediate parties to facilitate that communication. Each of these intermediate points might be running different hardware and software, or might use a completely different physical interface for communication. For example, if I call a friend on her cell phone from my laptop, the packets containing my voice data must travel from my network card to my wireless router, and then along an Ethernet connection to my ISP, and then through a series of backbone routers to eventually end up at a cell tower, and then finally to my friend's cell phone. This description still simplifies the actual process and completely ignores the question of how my computer discovered which IP address her phone had at the exact moment I called her in the first place, but it highlights the heterogeneity of interfaces and devices that must be compatible with one another for communication to function.

This is similar to the way in which two people who send letters to each other rely on several different postal systems, each with its own modes of transportation, sorting and routing procedures, addressing systems, etc., for letters to reach one another.

The way that all these different systems interface with each other is by organizing the network into a stack of layers. Each layer of the stack corresponds to a different level of abstraction in the network, and the layers provide services to the layers above them, and request services from the layers below. One of the most common layering models for telecommunication networks is the OSI model, which consists of seven layers. From lowest to highest, they are the

1. Physical Layer. The medium through which data flows, such as radio waves or a fiber-optic cable

2. Link Layer. The communication between adjacent nodes of a network

3. Network Layer. End-to-end communication in the network (IP addresses address nodes at this layer)

4. Transport Layer. Concerned with routing traffic to the correct port at each endpoint and enabling reliable data transfer. It is frequently combined with the following layer, since the TCP protocol contains aspects of both.

5. Session Layer. This layer models a complete conversation between two hosts instead of the routing of individual messages. I mention it for completeness but it is not important for the rest of the thesis.

6. Presentation Layer. This layer standardizes message formats. I also just include it for completeness.

7. Application Layer. Application-specific protocols, such as HTTP or the Signal Protocol.

When information is sent over the network, data from each layer is typically encapsulated by the layer beneath it. An IP packet on the network layer contains headers that include a source and destination IP address. The payload of an IP packet may be a UDP datagram (layer 4), which specifies the source and destination port numbers for the packet. The datagram then may encapsulate application data, such as a streaming video or a DNS query.

The concept of encapsulation of data plays an important role in this thesis. While network traffic typically respects the layer ordering in its encapsulation, it does not necessarily have to. For example, in tunnel protocols, a TCP segment at layer 4/5 may encapsulate an inner IP packet at layer 3. What is important is that the encapsulating layer is providing a service (in this case port routing) for the data it encapsulates.

Chapter 3

The PQConnect Protocol

PQConnect provides server to client authentication, post-quantum protection of both data and headers of the inner IP packet, packet authentication, and perfect forward secrecy. As a nice bonus, because PQConnect public keys are bound to servers via DNS records, DNS name servers supporting PQConnect automatically provide post-quantum security to DNS, which at the moment largely lacks even pre-quantum security and is vulnerable to serious network attacks.

This chapter offers a detailed specification of PQConnect that I developed based on the original sketch in [Ber18]. I discuss my design rationale, the various cryptographic algorithms I chose, and how my specification of PQConnect avoids various network attacks.

3.1 Cryptographic Design

PQConnect aims to make secure, post-quantum tunnels possible to deploy end-to-end across the entire Internet. However, the majority of post-quantum cryptosystems is relatively new, and the security of many underlying mathematical hardness assumptions is subject to ongoing analysis and debate, including for candidates in the third round of NIST's current post-quantum standardization process (see, e.g., [Ber20]). Novelty generally decreases confidence in a cryptosystem, since there has been less time to investigate potential attacks against it or scrutinize the underlying hardness assumption(s).

The goal of this work is to protect network data against a quantum adversary, while also not weakening pre-quantum security, even if all the post-quantum cryptosystems are later broken. PQConnect does this using a hybrid approach in the handshake phase that nests pre-quantum cryptographic algorithms inside of post-quantum schemes, and newer constructs inside of older ones. Nesting algorithms inside each other (as opposed to using both side by side) has the added benefit of forcing an attacker to work sequentially instead of in parallel: an attacker must first successfully attack the outer cryptosystem before being able to attack the inner one. By placing the most secure scheme in the outermost layer, we ensure the attacker is faced with this problem first, adding protection to newer and potentially weaker systems inside. Because the final shared secret depends on all the public key schemes (both pre- and post-quantum), even if an attacker does successfully break the post-quantum schemes, the combined system will at least not have *less* security than the pre-quantum algorithm does. Hybrid approaches like this have been used in other projects trying to incrementally deploy post-quantum cryptography to existing protocols or infrastructure, such as the development

of post-quantum X.509 PKI [Bin+17]. A discussion of the specific implementation of the nesting in `PQConnect` is given in section 3.3.1.

`PQConnect` largely assumes a client-server architecture, but this is not technically required. Any two peers can use `PQConnect` as long as the long term public keys of one peer are known to the other. In the following we refer to the peer whose long-term public keys are known as the server. When two peers establish a tunnel, they first perform a handshake to derive a shared secret derived in part from the server's long-term public keys. Because only the server possesses the corresponding private keys, the successful establishment of a tunnel implicitly authenticates the server to the client without requiring the use of signatures. This contrasts `PQConnect` with, say, TLS, where the server signs messages to the client with the private key belonging to its X.509 certificate public key.

The `PQConnect` handshake also provides forward secrecy using ephemeral keys with fast key erasure. Forward secrecy is the property that even if all keys in memory (including long term private keys) are compromised, no attacker can decrypt messages communicated prior to the moment of compromise. Fast key erasure is the approach by which keys are erased as soon as they are no longer needed. By immediately erasing ephemeral keys we prevent an adversary from later compromising one peer and recovering the secrets from the handshake. The handshake is described in detail in section section 3.3.

After the handshake is completed, both peers possess two shared secret keys, which they then use to send and receive encrypted transport data. Each key forms the root of a ratchet that generates one-time keys to symmetrically encrypt and authenticate packets. One ratchet is used for encrypting packets from client to server, and the other for encrypting from server to client. Fast key erasure again ensures that no keys are left on either device after they have been used. Additionally, `PQConnect` implements these ratchets in a novel way so that even an adversary who can prevent one peer from decrypting packets (for instance by blocking traffic on the network) cannot indefinitely prevent unused keys from being erased. The details of the transport phase are discussed in section section 3.4

3.1.1 Cryptographic Primitives

This section presents some background on specific cryptographic algorithms in `PQConnect` and the rationale for choosing them.

Public key cryptography

`PQConnect` uses Classic McEliece [Alb+20] (with the `mceliece8192128` parameter set) as its long-term post-quantum public key cryptosystem and uses Streamlined NTRU Prime [Ber+20] (with the `sntrup4591761` parameter set) for its ephemeral post-quantum keys. Both long-term and ephemeral `x25519` keys [Ber06] are used to perform ECDH key exchange for additional pre-quantum security. Classic McEliece is a third round candidate in the NIST PQC competition, based on the McEliece cryptosystem, one of the oldest and longest-surviving public key cryptosystems. It was published by Robert McEliece in 1978 [McE78], only one year after RSA. As an IND-CCA2 secure KEM based on error-correcting codes, it avoids the potential risks of future attacks against lattice-based schemes, making it a conservative choice from a security perspective. Streamlined NTRU Prime is a lattice-based scheme also in the third round of the NIST PQC competition. While it is a less conservative choice from a security perspective than McEliece, it has much smaller public keys, and is thus

better suited for sending public keys frequently over the network.

Authenticated encryption

Each tunnel packet is encrypted and authenticated using the `ChaCha20-Poly1305` authenticated encryption with associated data (AEAD) scheme [NL18]. This allows unencrypted `PQConnect` header information to be authenticated. During the handshake phase, `ChaCha20-Poly1305` is also used to protect the confidentiality of the handshake data and ensure that associated state variables are authenticated. In `PQConnect`, a key ratchet described in section section 3.4.1 is used to constantly derive per-packet secret keys.

Hash and KDF constructs

`PQConnect` uses BLAKE3 as its hash function due to fast performance [OCo+][Danb]. For key derivation, `PQConnect` uses the ChaCha20 stream cipher as pseudo-random generator (PRG) with fast key erasure under an input key $\mathcal{K}$ and zero nonce. This follows the construct described by Bernstein in [Ber17]. The first 32 bytes of output automatically overwrite $\mathcal{K}$ with a new key, while any further generated 32-byte chunks of stream can be used as more keys, if necessary. The KDF can also take an optional 32-byte input $\mathcal{D}$. If this input is present, then a new key $\mathcal{K}'$ is computed as the ChaCha20 encryption of $\mathcal{D}$ under $\mathcal{K}$. The KDF then outputs the required number of keys as 32-byte chunks of ChaCha20 stream under $\mathcal{K}'$. More details about how the KDF is used are given in the sections on the `PQConnect` handshake section 3.3 and transport data section 3.4.

3.2 Opportunistically Boring Tunnels

To establish a `PQConnect` tunnel with a server, a client must obtain the server's long-term public keys. Key distribution and public key infrastructure (PKI) are major challenges in real world cryptography in general. The approach that `PQConnect` takes is very similar to the one proposed in the MinimaLT protocol [Pet+13], in that it uses authenticated DNS records to bind a server's domain name to its `PQConnect` public keys. On the client side, `PQConnect` acts as a proxy between the client and the network. When a client machine connects to the Internet, `PQConnect` forms a tunnel (as described in the next sections) with a DNS name server that supports `PQConnect`. The public keys of that name server (or at least their hash) should exist pre-installed on the client device, similar to root CA certificates used in web PKI. Once this tunnel is created, the client uses it to make encrypted and authenticated DNS queries for other servers that it wishes to connect to.

Whenever software on the client performs a DNS query, `PQConnect` intercepts the DNS request and ensures that it is sent through the tunnel established with the trusted name server. Then it inspects the response to determine whether the host supports `PQConnect`. If the DNS query returns a CNAME record containing the magic string `pq1` followed by a 52-character Base32-encoded hash of the host's long-term public keys, then this signals support for `PQConnect`, and the client can proceed to establish a tunnel with the server. The benefit of using CNAME records is that they are automatically included in address record (A/AAAA) queries, allowing `PQConnect` to piggyback off of queries the client must perform anyway. Additionally, the authenticated response from the name server forms a trusted link

between the name server and the public key of the desired server, similar to X.509 certificate chains that link a server's public key to a trusted certificate authority.

When the client obtains the server's hash from DNS, `PQConnect` checks whether it already possesses the server's long-term `mceliece8192128` and `x25519` public keys in its local cache. The (hash, key) pairs are indexed by the server's apex domain name. This makes it straightforward to check if the keys have changed and notify the user of this change, if desired. If the client does not possess the keys matching the hash, it must obtain them either directly from the server or from an external public directory.

3.2.1 Streaming Verification of Long-term Keys

Because the `mceliece8192128` public keys are large (1,357,824 bytes), we do not want the client to have to store packets of key material before being able to verify that they are correct. Instead we want to be able to verify each packet of key material as it is received, and immediately reject those that do not verify. To accomplish this we transmit the keys as part of a Merkle tree, with the public hash of the long-term public keys in the root and with packet-sized chunks of the public keys stored in the leaves. Each internal node of the tree is the hash of its children, and the tree has variable arity that ensures that all the children of any given node can be transmitted in a single packet of roughly 1KB. This allows the client to perform streaming verification by hashing the contents of each received packet and comparing the digest with that packet's parent node, provided that they have already received the parent node of that packet in the tree. Any packet whose hash does not match its parent can be immediately discarded and re-requested.

The procedure for distributing and verifying `PQConnect` public keys is depicted in Figure 3.1. The `mceliece8192128` public key is divided into 1179 parts of 1152 bytes (the last part has fewer). The 32-byte long-term `x25519` public key is appended to the last part (which becomes 800 bytes in total). These parts form the leaves of the tree (at depth 3). Each leaf node is hashed independently to form 1179 nodes at depth 2 of the tree. The depth-2 nodes are grouped together into groups of 36 nodes (each group being 1152-bytes long, except for the last group which has fewer nodes), forming 33 groups in total. Each group is then hashed into the nodes at depth 1. The length of all 33 nodes at depth 1 is $33 \cdot 32 = 1056$ bytes. These nodes are finally hashed to obtain the 32-byte public hash provided by DNS. The full tree is thus a {33,36,1}-ary Merkle tree.

When a client wishes to obtain the server's long term keys, it requests the nodes in a breadth-first order, beginning with the depth 1 nodes, which are sent in a single packet. If the hash of this packet matches the server's public key hash, the client then proceeds to depth 2. Each depth 2 packet is hashed and verified against the appropriate node (32-byte slice) from the depth 1 packet. Finally, the parts of the `mceliece8192128` public key are requested, hashed, and verified against the depth 2 nodes.

3.3 The `PQConnect` Handshake

Once the client has obtained and verified the server's long-term keys, they can proceed to establish a tunnel. The first step of this process is to perform a handshake that results in a shared secret and authentication of the server to the client. If the server regularly uploads ephemeral public keys to a directory service, these can be pre-fetched by the client, allowing the client to perform a 0-RTT handshake and send data in the first message. Otherwise,

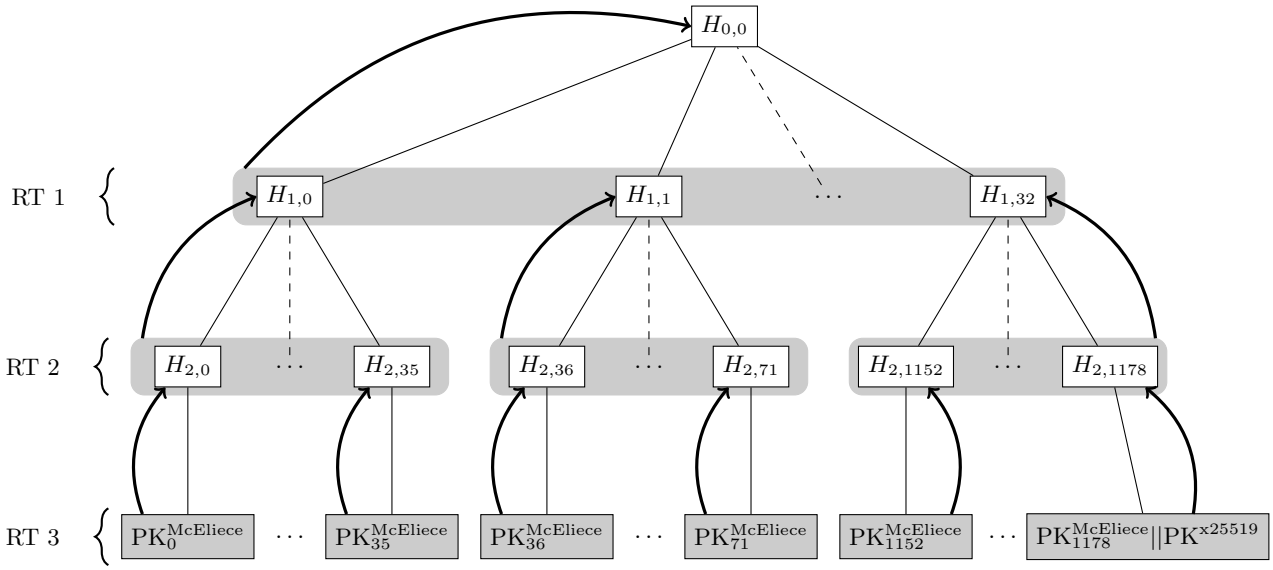

Figure 3.1: The structure and verification process for the `PQConnect` public key Merkle tree. Nodes of the tree are shown as rectangles. Leaf nodes are 1152 bytes, except for the last one, which is 800 bytes. Internal nodes are 32-byte hashes H indexed by tree depth and position within their given depth. Their values are equal to the hash of the total concatenated bytes of their children. The shaded regions indicate how children of internal nodes are grouped into packets, and so each shaded region represents a single packet. Arrows pointing upwards from packets to internal nodes represent verifying a packet by hashing its contents and comparing this hash with its parent. To obtain and verify the server's public keys, the client first requests the depth 1 packet and checks that its hash matches the hash $H_{0,0}$ which was obtained via DNS. They then request and verify the 33 depth 2 packets. Finally they request and verify the 1179 leaf packets, which comprise the public keys. Requests at each level can occur in parallel, so the entire process can be performed in three RTTs. Each round trip is indicated to the left of the tree.

the client and server must perform a 1-RTT handshake, and the client can send data in the second message to the server.

This section first describes each version of the handshake in detail, beginning with the 0-RTT version, then a first attempt at a 1-RTT adaptation, and then the final 1-RTT version, which improves on the first attempt. Because `PQConnect` is not specific to any particular network architecture and can be used to protect server-server communication as well, for instance, we now abstract away from the client-server terminology and instead speak of an Initiator and a Responder. The only requirement for the Responder is that they have generated and published their long term public keys as described in section 3.2.

The final 1-RTT handshake also takes inspiration from the pre-quantum Noise Protocol Framework [Per18], specifically the **NK** pattern, where the Initiator has **N**o static key, but **K**nows those of the Responder. However, we adapt it to include the use of post-quantum KEMs in addition to elliptic curve Diffie-Hellman (ECDH), which is the only defined public key operation in Noise. If the Responder also broadcasts its ephemeral public keys to a public directory, a client can look up these keys and perform a 0-RTT handshake, making it possible to send transport data in the first message.

Notation

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

$\leftarrow, \xleftarrow{\$}$. The arrow symbol $\leftarrow$ indicates assignment of a value on the right hand side of the arrow to a variable on the left hand side. The symbol $\xleftarrow{\$}$ represents assignment of a random element of a set to a variable. For example, in the expression $\texttt{a} \xleftarrow{\$} \{\texttt{0}, \ldots, \texttt{255}\}$, a random byte value is assigned to `a`.

$\mathbf{spk}_{\mathbf{p}}^{\mathbf{X}}, \mathbf{ssk}_{\mathbf{p}}^{\mathbf{X}}, \mathbf{epk}_{\mathbf{p}}^{\mathbf{X}}, \mathbf{esk}_{\mathbf{p}}^{\mathbf{X}}$ are static public, static private, ephemeral public, and ephemeral private keys for public key cryptosystem/KEM **X** and peer **p**.

X.keygen() Generates a random public, private keypair for the public key cryptosystem **X**. For `mceliece8192128`, `sntrup4591761`, and `x25519` these have lengths (1357824, 14080), (1218, 1600), (32, 32) in bytes, respectively.

X.Encap(pk) Generates a random 32-byte key **k** and its encapsulation **c** under the public key **pk**. For `mceliece8192128` and `sntrup4591761` the lengths of **c** are 240 and 1047 bytes, respectively.

X.Decap(sk, c) Takes a private key **sk** and ciphertext **c** and outputs the encapsulated key **k** if **c** is a valid encapsulation. Otherwise outputs $\perp$.

ChaCha20(k, n, m) Generates the `ChaCha20` encryption of message **m** under 32-byte key **k** and 32-bit nonce **n**, represented as a little-endian integer.

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 described in [NL18]. A ciphertext variable $\mathbf{c}^*$ containing a * indicates that the ciphertext **c** contains a 16 byte long authentication tag.

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 $\perp$.

BLAKE3(m) Generates a 32-byte BLAKE3 hash digest of the message **m**.

$\mathbf{KDF_i}(\mathbf{k}, \mathbf{d})$ Generates **i** 32-byte keys from key **k** and optional input **d**. If **d** is the empty string ϵ, then the output is simply the output of $\mathbf{ChaCha20}(\mathbf{k}, 0, 0^{\mathbf{i}\cdot 32})$ divided into 32-byte chunks. If **d** is non-empty, then it must be a 32-byte string. First the KDF call generates a new key $\mathbf{k_1} \leftarrow \mathbf{ChaCha20}(\mathbf{k}, 0, \mathbf{d})$ and then computes $\mathbf{KDF_i}(\mathbf{k_1}, \epsilon)$.

$\mathbf{C_p}, \mathbf{H_p}$ are the CipherState and HandshakeState objects for peer **p**. During the handshake each peer `p` maintains these two state variables. The first accumulates all secrets generated by the public key cryptographic operations using the KDF and is used as the symmetric key in all AEAD operations during the handshake. The HandshakeState object contains the hashed transcript of the handshake conversation itself. It is used as associated data for all AEAD operations, ensuring that both participants only establish a tunnel if they see the same transcript.

$\mathbf{T_p}$ is the root sending key for peer **p**. Thus $\mathtt{T_I}$ is the root sending key for the Initiator (and the root receiving key for the Responder).

tunnelID This is a 32-byte pseudo-random value that uniquely identifies a tunnel. This value must be unique for all tunnels currently in use by each peer.

3.3.1 Nesting Cryptographic Algorithms

The `PQConnect` handshake derives a shared secret from the accumulated outputs of public key operations using `mceliece8192128` and `x25519` long term keys and `sntrup4591761` and `x25519` ephemeral keys. As stated earlier in this chapter, `PQConnect` nests the computations of these schemes inside of one another to both force an attacker to attack each system sequentially, and also to provide a layer of confidentiality to the inner public key and ciphertext values. Forcing the attacker to peel off each layer in sequence not only prevents parallel computation against each public key cryptosystem, but also forces a specific ordering to the attack, limiting the approaches an attacker can choose.

There are several options one has for this ordering, however, 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 requires around 147,000 cycles on a 2020 Intel Core i5 CPU, while decapsulating a `mceliece8192128` key requires 1.19 million cycles on the same machine [Dana], so using `x25519` as the first layer may also waste fewer cycles if an error is encountered later in the handshake. However, this has the drawback of immediately exposing a pre-quantum algorithm to a potential attacker with a quantum computer.

It is also important to consider potential attacker models. A quantum attacker will be able to solve the discrete log problem for a server's public `x25519` key. Therefore if we put `x25519` in the outer layer, we are giving away this layer for free to a quantum attacker.

If we assume a multi-target pre-quantum attacker who wants to break the confidentiality of many servers, then putting `x25519` in the outer layer may allow the attacker to peel back this

layer on more handshakes than if we used `mceliece8192128` keys instead. This is because once an attacker has broken a single `x25519` key, the cost of breaking t more keys is an additional multiplicative factor of only $(O\sqrt{t})$ [KS01]. So once the attacker has successfully broken one server's long-term `x25519` key, for example, breaking the next 10,000 `x25519` keys it sees is only 100 times as much work. Furthermore, solving the discrete log problem once means that *all* connections to that server are now compromised. In contrast, the best attacks against code-based cryptography recover the message but not the private key. This forces an attacker to spend effort breaking each connection, even in the best case. An attacker has a speed advantage in breaking a single `mceliece8192128` ciphertext out of N observations. However, they do not gain a further speed-up for decoding all or subsequently observed ciphertexts[1].

In order to minimize the attack surface against attacks on ECDH like the ones just mentioned, I chose to use Classic McEliece with the `mceliece8192128` parameter set in the outer nest layer, despite its higher relative computation cost when compared to `x25519`. The `x25519` ECDH key agreement takes place in the middle layer of the nesting, first by computing a shared key between an Initiator's ephemeral `x25519` key and the Responder's long term `x25519` key, and then again between the Initiator's key and the Responder's ephemeral `x25519` key. Finally, the `PQConnect` handshake uses an ephemeral `sntrup4591761` public key at the innermost layer to add a final layer of defense against a quantum attacker, in the case that the long term `mceliece8192128` key is later compromised.

3.3.2 A 0-RTT Handshake

`PQConnect` encourages, but does not require, the use of public key directories where servers can regularly upload short-lived public keys. If a server uses such a directory to broadcast its ephemeral keys, then a client can first establish a tunnel with this directory, obtain the server's encrypted ephemeral `x25519` and `sntrup4591761` keys, and then use them to establish a tunnel with the server in 0-RTT, sending encrypted packets in the first message. During the handshake a secret is created using each of the Responder'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 can be seen in detail in Figure 3.2.

The Initiator begins the handshake by generating a 32-byte ephemeral `x25519` public key and its 32-byte private key. They then encapsulate a random key $\mathtt{k}_0$ to the Responder's long term `mceliece8192128` public key as $\mathtt{c}_0$. The Initiator then encrypts their ephemeral `x25519` public key $\mathtt{epk}_\mathtt{I}^{\mathtt{x25519}}$ under $\mathtt{C}_\mathtt{I} = \mathtt{k}_0$, nonce 0, and associated data $\mathtt{H}_\mathtt{I}$, producing $\mathtt{c}_1^*$. Next, they derive the ECDH keys $\mathtt{k}_1$ and $\mathtt{k}_2$ using the Responder's static and ephemeral `x25519` public keys, respectively. Finally, the Initiator encapsulates $\mathtt{k}_3$ under the Responder's `sntrup4591761` public key, generating $\mathtt{c}_2$, and then encrypts this under the updated $\mathtt{C}_\mathtt{I}$ and $\mathtt{H}_\mathtt{I}$ values and nonce 0 to produce $\mathtt{c}_3^*$. The $\mathtt{tunnelID}, \mathtt{T}_\mathtt{I}$ and $\mathtt{T}_\mathtt{R}$ shared secrets are then computed using the CipherState and HandshakeState variables as inputs to the KDF. The Initiator sends the appropriate message type, $\mathtt{c}_0, \mathtt{c}_1^*$, and $\mathtt{c}_3^*$ to the Responder. The total length of this message is 1353 bytes: 2 bytes for message type (`0x01` as a 16-bit little-endian integer), 240 bytes for the `mceliece8192128` ciphertext, 48 bytes for $\mathtt{c}_1^*$ (32 bytes for the key and 16 bytes for the authentication tag), and 1063 bytes for $\mathtt{c}_3^*$ (1047 byte `sntrup4591761` ciphertext + 16

[1]For more on DOOM (decoding one out of many) attacks on McEliece, see [Sen11]

byte authentication tag).

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

Note that for each AEAD operation, the HandshakeState variable is used as authenticated data, ensuring that the handshake only succeeds if both participants have an identical view of the handshake transcript. Except for the first encapsulated secret key, all message fields containing data are authenticated and encrypted. At the conclusion of the handshake, both parties erase all remaining non-public values aside from the `tunnelID`, $\mathtt{T}_\mathtt{I}$, and $\mathtt{T}_\mathtt{R}$.

3.3.3 A 1-RTT Handshake: First Attempt

If the Responder has not published ephemeral keys to a public directory, or the directory is unreachable by the Initiator, then the ephemeral keys need to be communicated as part of the handshake. A simple way to accomplish this would be for the Initiator to first send $\mathtt{c}_0$ and $\mathtt{c}_1^*$ to the Responder, wait for the Responder to send their ephemeral keys—encrypted under the same $\mathtt{k}_0$ and $\mathtt{k}_1$ as before—and then complete the remaining steps of the handshake. This procedure turns the 0-RTT handshake into a 1-RTT handshake, as the Initiator can only establish a shared secret with the Responder after receiving the response message. A modified version of the 0-RTT handshake is shown in Figure 3.3, with differences from the 0-RTT version highlighted in blue.

We can see that the same nesting of cryptographic primitives is present in the 1RTT version. However, this modified version has several drawbacks. The first is that the length of $\mathtt{msg.type}||\mathtt{c}_0||\mathtt{c}_1^*$ (290 bytes) is much shorter than the length of the response message (1284 bytes), making the handshake protocol vulnerable to amplification attacks by an adversary starting many handshakes with a server and directing the responses to a victim IP address.. To prevent this, we must add 994 bytes of padding to the `init` message, so that the Initiator must send at least as much data as the Responder. Including bytes that serve no computational purpose is an obvious inefficiency.

Another major drawback to this version is that the Responder must be able to link the two messages from the Initiator to each other in order to complete the handshake. This could be done by sending an encrypted nonce with the response message (not shown), for instance, that the Initiator would need to include in their final `fin` message. But in any case, the Responder now is required to keep track of the state of *incomplete* handshakes. This wastes the Responder's resources storing and tracking messages from many peers. Ideally we want the Responder to receive a valid `init` message, compute the secrets $\mathtt{T}_\mathtt{I}, \mathtt{T}_\mathtt{R}$ from this message, and send a response that allows the Initiator to compute the same values.

3.3.4 A 1-RTT Handshake: Improved Version

The problem with our first attempt at a 1-RTT handshake stems from the directionality of establishing a shared secret with a KEM: the owner of the KEM public key must wait for another peer to send them an encapsulated secret key. It now becomes clear that the reason the Responder in the previous section cannot finish the handshake immediately after receiving an `init` message is that they must first send an `sntrup4591761` public key to the Initiator and wait for them to return an encapsulated key in the next message. Meanwhile, sending

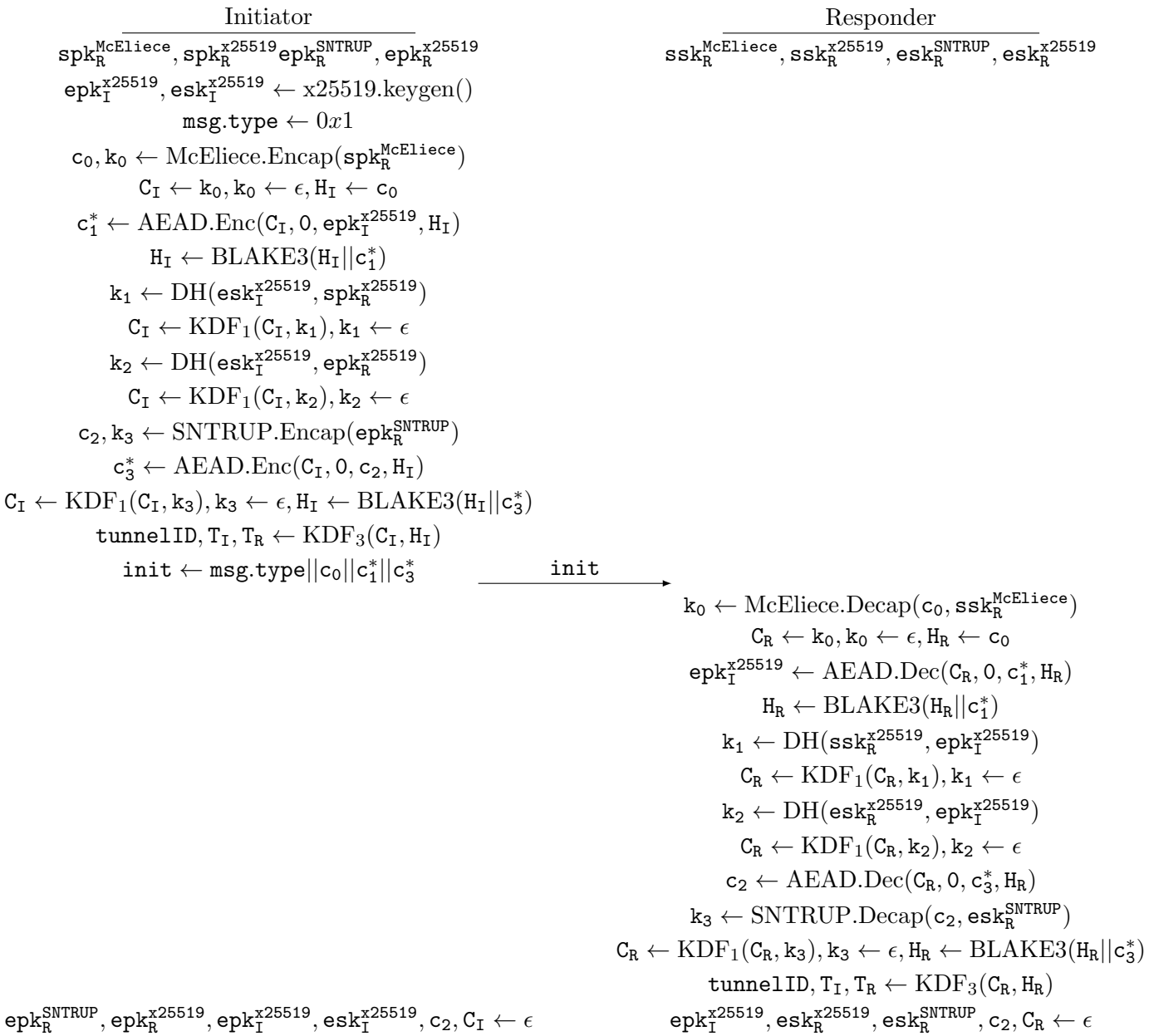

Figure 3.2: 0-RTT `PQConnect` Handshake

Figure 3.3: A first attempt at a 1-RTT PQConnect Handshake if the Initiator does not know the Responder's ephemeral keys. Differences from Figure 3.2 are highlighted in blue.

this 1218-byte key in the first place was the main reason why the Initiator needed to add 994 bytes of padding to the `init` message.

We solve both these problems by instead having the Initiator generate an ephemeral `sntrup4591761` public key and include this in the `init` message to the Responder. This automatically makes the first handshake message longer than the response, and allows the Responder to complete the handshake without keeping state by generating $\mathtt{k}_3$ instead of the Initiator. This improved version is depicted in Figure 3.4

The `init` message

The Initiator first generates a pre-quantum ephemeral `x25519` keypair and a post-quantum ephemeral `sntrup4591761` keypair. Using the Responder's long-term `mceliece8192128` public key, the Initiator encapsulates a random secret $\mathtt{k}_0$ as the ciphertext $\mathtt{c}_0$ and initializes the CipherState and HandshakeState variables with each value, respectively. They then encrypt their `x25519` public key as before to generate $\mathtt{c}_1^*$, effectively nesting the ephemeral `x25519` public key inside of the long-term post-quantum scheme. After deriving the ECDH shared key $\mathtt{k}_2$ using the Responder's long term `x25519` key and incorporating this into $\mathtt{C}_\mathtt{I}$, the Initiator then encrypts their `sntrup4591761` public key as $\mathtt{c}_2^*$ and sends $\mathtt{c}_0, \mathtt{c}_1^*$, and $\mathtt{c}_2^*$ to the Responder.

The total length of the first message is 1524 bytes: two bytes for the message type (`0x02` as a 16-bit little-endian integer), 240 bytes for the McEliece-encapsulated symmetric key, 32 bytes for the `x25519` public key, 16 bytes for the authentication tag on $\mathtt{c}_2^*$, 1218 bytes for the `sntrup4591761` public key, and 16 bytes for the authentication tag on $\mathtt{c}_3^*$.

The `resp` message

When the Responder receives a new initiation message, they decapsulate $\mathtt{k}_0$ from $\mathtt{c}_0$, initializing their CipherState and HandshakeState variables appropriately.. This allows them to decrypt $\mathtt{c}_1^*$ and obtain the Initiator's ephemeral `x25519` key. After performing an ephemeral-static ECDH computation, they compute $\mathtt{k}_1$, which they incorporate into their CipherState to decrypt $\mathtt{c}_2^*$, obtaining the `sntrup4591761` key $\mathtt{epk}_\mathtt{I}^{\mathtt{SNTRUP}}$.

To construct the response message, the Responder first generates an ephemeral `x25519` keypair and encrypts their public key under $\mathtt{C}_\mathtt{R}$ and nonce 1 to generate $\mathtt{c}_3^*$. Finally, with the Initiator's ephemeral keys, the Responder performs an ephemeral-ephemeral ECDH key agreement to obtain $\mathtt{k}_2$, and encapsulates a random key $\mathtt{k}_3$ against the Initiator's $\mathtt{epk}_\mathtt{I}^{\mathtt{SNTRUP}}$ key. The encapsulated key $\mathtt{c}_4$ is once again encrypted and authenticated under $\mathtt{C}_\mathtt{R}$, creating ciphertext $\mathtt{c}_5^*$. Finally, the Responder generates the values `tunnelID`, $\mathtt{T}_\mathtt{I}$, and $\mathtt{T}_\mathtt{R}$ as before, and sends $\mathtt{c}_3^*$ and $\mathtt{c}_5^*$ to the Initiator. The total length of the response message is 1113 bytes: two bytes for the message type, 32 bytes for the `x25519` public key, 16 bytes for the authentication tag on $\mathtt{c}_3^*$, 1047 bytes for the `sntrup4591761` ciphertext, and another 16 bytes for the authentication tag on $\mathtt{c}_5^*$. Note that the response message is indeed shorter than the initiating message, thus providing protection against amplification attacks.

Important Remark. Because no new keys have been incorporated into the CipherState variable before the Responder encrypts $\mathtt{c}_3^*$, the nonce **must** be incremented to 1 to prevent reusing the same (key, nonce) pair.

The conclusion

To conclude the handshake, the Initiator decrypts $\mathtt{c}_3^*$ under $\mathtt{C}_\mathtt{I}$, nonce 1, and associated data $\mathtt{H}_\mathtt{I}$ to obtain the Responder's `x25519` key $\mathtt{epk}_\mathtt{R}^{\mathtt{x25519}}$. They incorporate the ephemeral-ephemeral ECDH key into $\mathtt{C}_\mathtt{I}$ (and $\mathtt{c}_3^*$ into $\mathtt{H}_\mathtt{I}$), which they then use to decrypt $\mathtt{c}_5^*$ to obtain $\mathtt{c}_4$. After decapsulating $\mathtt{c}_4$, they obtain the final key $\mathtt{k}_3$, which they use to complete the handshake, exactly as the Responder did above.

3.4 Transport Data

Once the Initiator and Responder have successfully completed either the 0-RTT or 1-RTT handshake, they each share a unique `tunnelID` and secret keys $\mathtt{T}_\mathtt{I}$ and $\mathtt{T}_\mathtt{R}$. They can now begin sending encrypted, encapsulated IP packets over UDP.

3.4.1 The `PQConnect` Key Ratchet

To achieve a strong notion of forward secrecy, a key used to encrypt or decrypt a packet must be erased as soon as it is no longer needed, with no way to recover it. Otherwise an attacker can store sent messages, later steal or otherwise compromise a device, and retrieve (or recompute) its keys to decrypt the messages.

Note that while it is always obvious to the sender of a packet when a key is no longer needed, receiving a packet depends on the network, and thus the difficulty of ensuring fast key erasure is predominantly a problem only on the receiving end. Therefore we focus on the receiver's side of the key ratchet for the rest of this section.

A simple way to quickly forget keys would be to use a one-way KDF to derive a unique key for each packet. Every time a key is used, we apply the KDF to that key and "ratchet" forward to the next one. That is, if the client C wished to send packet P_i, they would encrypt it with key c_i. As soon as P_i is sent, the client would then apply the KDF to c_i and obtain $c_{i+1} = \mathrm{KDF}(c_i)$. Finally, they would delete c_i. The server S would then mirror this ratcheting to receive packets from the client. The keys then form a single chain that ratchets forward and are consumed upon use.

While this ratchet construction would work in an ideal scenario where incoming packets are never dropped and always arrive in the order they are sent, it does not work well in practice. Unlike TCP, UDP gives no guarantee of correct packet-ordering. If a packet is dropped en route to its destination, how will the recipient know if a packet is simply delayed or if it will never arrive? If the packet will eventually arrive, deleting its key would be wasteful. If the packet will never arrive, keeping the key in memory risks allowing an attacker to recompute all the keys subsequent to it, even ones that have already been used for decryption.

We can improve our one-time key scheme by not directly using keys from the chain to encrypt and authenticate packets, but instead deriving a subkey from each chain key and exclusively using these subkeys for encryption. The keys that form the chain then generate both packet keys as well as new chain keys. In this way, the computational link between successive packet keys is broken, so keeping old packet keys in memory does not allow one to recompute later packet keys.

For example, let c_0 be the start of the client's sending ratchet and the server's receiving ratchet. To encrypt packet P_0, the client applies the KDF to c_0 to generate two keys: chain key c_1, and subkey c_0'. That is, $c_1, c_0' = \mathrm{KDF}_2(c_0)$. C immediately overwrites c_0 with c_1, and

Figure 3.4: 1-RTT PQConnect Handshake

c'_0 is used to encrypt P_0. Suppose C sends two packets, and S receives packet P_1 before P_0. S can ratchet once from c_0 to obtain c_1 and c'_0, and then again from c_1 to obtain c_2 and c'_1. S uses c'_1 to decrypt P_1 and immediately throws away the key. There is now no way for S to recover c'_1, because the main chain of the ratchet has advanced to c_2. Meanwhile, S can keep c'_0 in memory while it waits for P_0 to arrive without risking that c_1 or c'_1 be recomputed.

Expanding the ratchet

The idea of using a KDF to re-key encrypted communication has been around for a long time (see, e.g. [AB00]) and is widely-used in private messaging protocols. The construction just described is nearly identical to the symmetric key ratchet used in the Double Ratchet algorithm of the Signal Protocol, for example [MP16]. However, unlike in a private messaging conversation where rate of key use is limited by the rate of messages being sent, a network tunnel can receive many thousands of packets per minute, each encrypted with its own key. An attacker with the ability to drop traffic between two parties may take advantage of the large number of packets sent over a tunnel by dropping all the packets from one peer to another, storing them, and later stealing the recipient's machine to decrypt the stored traffic. Because the attacker prevented the dropped packets from arriving, the receiver's chain keys will not have been ratcheted forward, and none of the existing decryption keys for the dropped packets will have been deleted from memory.

D2.5 [Ber18] addresses this problem by adding a time dimension to the key ratchet that forces keys to expire after a fixed period. We divide time into 30-second epochs. For each epoch there is an epoch key that forms the root of a ratchet as described above. One can visualize this as a kind of "tree", with epochs advancing along the trunk, the ratchet chain keys forming the branches, and the packet keys forming the leaves. After 30 seconds have elapsed, the epoch chain is ratcheted down the trunk, generating the initial key of the next branch. Each packet $P_{i,j}$ is sent with authenticated index values i, j that indicate the packet should be decrypted with the j^{th} packet key from epoch i (for the full message format see section 3.4.2). In order to protect against the above attack, however, all the keys from a given epoch are deleted at the latest after 120 seconds. So if epoch E starts at time t, the sender will switch to epoch $E + 1$ at time $t + 30$, and the receiver will delete all keys from the ratchet of E by $t + 120$ at the latest. Waiting 120 seconds allows for some delay in delivery as well as for small variations in local clock values. The full `PQConnect` ratchet is depicted in Figure 3.5

Synchronizing local clocks

It is possible that because of local clock variation, the two peers will not be completely in sync for the duration of a tunnel. D2.5 [Ber18] specifies that peers should advance their epoch counters in lockstep as much as possible. If one peer receives a packet from epoch n at time $t < 30n$, they can 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 should not slow down their clock. Instead, they should 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 again from delaying the erasure of keys by delaying the arrival of packets.

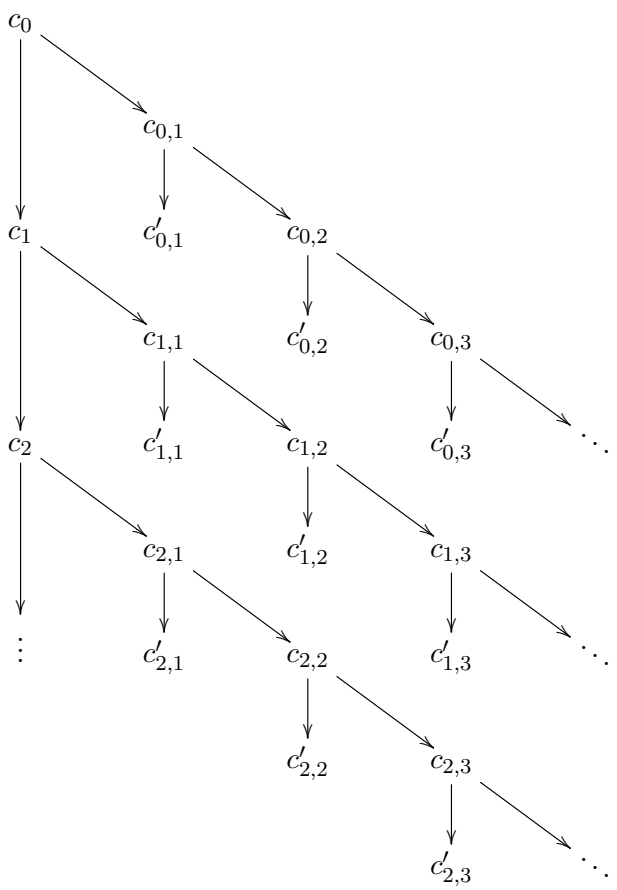

Figure 3.5: The `PQConnect` key ratchet. Keys are erased as soon as they are used, and in any case within two minutes. Key $c'_{0,i}$ is used for the ith client packet between time 0 and time 30, and is erased by the server as soon as it is used, or at the latest at time 120. Key $c'_{1,i}$ is used for the ith client packet between time 30 and time 60, and is erased by the server as soon as it is used, or at the latest at time 150. Figure taken from [Ber18]

Additionally, a peer's view of time should be consistent for both sending and receiving packets. If a peer makes a forward adjustment to a new epoch on their receiving ratchet, they should make the same adjustment on their sending ratchet. This way a peer is never sending a message in an older epoch than the most recent one in which they have received a packet.

3.4.2 Message Format

When C wishes to send packet j in epoch i of a `PQConnect` tunnel, they encrypt the packet using key $c'_{i,j}$, then prepend the 32-byte `tunnelID`, 2-byte epoch number i, and 4-byte packet number j, and authenticate the encrypted data along with these fields. The epoch and index values are little-endian. This is then encapsulated as the payload of a UDP datagram and sent to the remote host. The packet arrives on the receiver's `PQConnect` UDP port. The receiver identifies that this packet is for tunnel `tunnelID` and then retrieves or computes $c'_{i,j}$ from their receiving ratchet to decrypt and verify the packet. The packet is then routed to

Figure 3.6: Structure of a `PQConnect` packet: Light gray areas are authenticated but not encrypted. Dark gray is authenticated and encrypted

the appropriate destination IP (if not the host machine) and port, and $c'_{i,j}$ is deleted. The structure of a `PQConnect` message is shown in Figure 3.6.

Chapter 4

Improvements to PQConnect

This chapter consists of two parts. I first discuss several improvements to the PQConnect protocol that go farther beyond the description of PQConnect given in [Ber18] and the previous chapter. Afterward I present a potential resource exhaustion attack against PQConnect and a potential solution for it.

4.1 Optimization

Because the post-quantum schemes used in PQConnect have large public keys, handshakes are a potential performance bottleneck for busy servers. Additionally, PQConnect performs some operations many times throughout the life cycle of a tunnel, such as key derivation. In this section I present several additional features and efficiency improvements to the PQConnect protocol that aim to reduce resource consumption for both peers.

4.1.1 Batch Key Derivation

Because we use a ChaCha20-based KDF, we can improve the efficiency of our computation by taking advantage of the reduced number of clock cycles per output byte as the output length of the stream cipher increases [Dana]. For example, to generate 64 bytes—the equivalent of one chain key and one packet key in the PQConnect ratchet—ChaCha20 requires roughly 9 cycles per byte of output. This decreases to 1 cycle per byte when we instead generate 608 bytes (equivalent of 19 keys) of output per invocation of the stream cipher. For longer streams the number of cycles/byte decreases still further, albeit less dramatically.

Using this information, we can improve the advancing step of the PQConnect ratchet as follows. Every time we apply the KDF to a chain key $c_{i,j}$ in our ratchet, we generate the keys $c_{i,j+18}, c'_{i,j}, \ldots, c'_{i,j+17} = \mathrm{KDF}_{19}(c_{i,j})$. We then overwrite $c_{i,j}$ with $c_{i,j+18}$ and use $c'_{i,j} \ldots c'_{i,j+17}$ to encrypt and authenticate packets j through $j + 17$ of epoch i.

Although ratcheting to a new epoch—moving down the vertical chain in Figure 3.5—only occurs roughly every 30 seconds, we can still take advantage of batch key derivation to generate the next epoch chain value, the first 18 packet keys of the new epoch, and the next chain key. At the moment we wish to ratchet from epoch $i - 1$ to epoch i, we have already erased c_{i-1} and replaced it with c_i. To ratchet to epoch i, we then compute $c_{i+1}, c_{i,18}, c'_{i,0}, \ldots, c'_{i,17} = \mathrm{KDF}_{20}(c_i)$, first overwriting c_i with c_{i+1} and then storing the next chain key $c_{i,18}$. Finally we store the 18 packet keys as before.

4.1.2 Tunnel Resumption

Because each peer in a `PQConnect` tunnel erases all their remaining receiving keys for each epoch after two minutes, tunnels will automatically time out after two minutes if the receiver's epoch chain is not ratcheted in the absence of traffic. Meanwhile the sketch of `PQConnect` in D2.5 [Ber18] explicitly stated that the protocol does not support resumption, since this conflicts with the goal of fast key erasure. However, there might be circumstances where it would be nice to keep tunnels alive longer, even if they are only being used sporadically. For example, it might be useful to keep tunnels to DNS name servers and ephemeral key directory servers open for a longer period, so that clients who make DNS queries more than two minutes apart are not required to perform a new tunnel handshake each time.

One option to keep tunnels open would be for one peer to send a keepalive message to the other every $t < 120$ seconds so that in the absence of other traffic, the recipient will always have a key available to decrypt incoming packets. This approach is taken in WireGuard and OpenVPN, for example. It is worth noting also that because `PQConnect` tunnels encapsulate higher layer protocols, this might already happen anyway if the encapsulated protocol implements its own keepalive feature like in TCP. The downside of this approach is that it adds extra traffic to an existing connection whose only purpose is to reduce future traffic in establishing a new connection.

A more silent option would be for the client to use the existing tunnel to request a batch of ephemeral public keys in advance from the server. This way they could subsequently establish a new tunnel with a 0-RTT handshake, as long as the server still has the associated private keys. If the server generates several ephemeral keys for future epochs at once they have the added benefit of being able to directly use the batch key generation algorithm for `sntrup` given in [Ber+21]. This algorithm significantly reduces the amortized cost in CPU cycles of generating `sntrup` keypairs. However, this also has the downside that the client will potentially not use most of the ephemeral keys it receives, resulting in unnecessary storage and data transfer.

We can further improve on the previous approach by avoiding `sntrup4591761` key generation altogether, and instead using an existing tunnel to agree on new set of tunnel initiation values $\texttt{tunnelID}_{\texttt{new}}$, $\texttt{T}_{\texttt{I}_{\texttt{new}}}$, and $\texttt{T}_{\texttt{R}_{\texttt{new}}}$. As soon as an Initiator and Responder have established a tunnel, the Initiator sends a special `nexttunnelID` message to the Responder. The Initiator first computes $\texttt{tunnelID}_{\texttt{new}}, \texttt{T}_{\texttt{I}_{\texttt{new}}}, \texttt{T}_{\texttt{R}_{\texttt{new}}}$ as $\texttt{tunnelID}_{\texttt{new}}, \texttt{T}_{\texttt{I}_{\texttt{new}}}, \texttt{T}_{\texttt{R}_{\texttt{new}}} = \mathrm{KDF}_3(\texttt{T}_{\texttt{I}_{\texttt{old}}}, \texttt{T}_{\texttt{R}_{\texttt{old}}})$. They then send the first 16 bytes of a new value $\texttt{tunnelID}_{\texttt{new}}$ to the Responder. The Responder then responds with an acknowledgment that includes the second 16-bytes of $\texttt{tunnelID}_{\texttt{new}}$ and a timestamp value indicating until when $\texttt{tunnelID}_{\texttt{new}}$ can be used. The Initiator stores the new values in a dictionary indexed the next `tunnelID`, so that if the current tunnel expires and the next `tunnelID` has not, the Initiator can simply start sending packets in the new tunnel encrypted under keys in the ratchet derived from $\texttt{T}_{\texttt{I}_{\texttt{new}}}$.

Establishing a new `tunnelID` for tunnel resumption is used in the MinimaLT protocol predominantly for re-keying the symmetric transport key, but also for IP-portability: A client can communicate with a peer from one IP, establish a new `tunnelID`, and then resume the connection under that new `tunnelID` from a new IP address with little interruption [Pet+13]. The `PQConnect` ratchet makes the re-keying aspect of establishing a new `tunnelID` redundant, but the IP-portability benefits are still applicable in this situation as well.

4.1.3 Ephemeral Key Directory

The existence of a reliable directory for servers' ephemeral keys makes it possible to perform a 0-RTT handshake. As mentioned in section 4.1.2, servers can batch generate `sntrup4591761` keypairs and upload the public keys to the directory at regular intervals. For example, a server can generate 720 `sntrup4591761` and `x25519` keys four times per day, each with a lifespan of 120 seconds that staggers every 30 seconds (the same validity time as each receiving epoch). The server uploads these batches every six hours to the directory. This would require less than a megabyte of storage at the time of each upload, and of course as the keys expire they are automatically deleted from the directory.

When a user wishes to establish a tunnel with a new server, it queries the key directory (with which it has already established a tunnel) for the currently valid `sntrup4591761` key, and then uses this to establish a tunnel with the server of interest. The `PQConnect`-protected DNS response for the server of interest can be modified to include not only the hash of the server's long-term public keys, but also a URL to the key directory used by the server. However, as mentioned in section 3.3.3, `PQConnect` benefits from, but does not require an ephemeral key directory to function. If a server does not use a key directory, the directory is unreachable, or even if directories are coerced into boycotting a particular server by an external authority, clients can always perform a 1-RTT handshake with a server without foreknowledge of the server's ephemeral keys.

It is also important to note that while epochs within a tunnel and a server's ephemeral keys have the same lifespan and pattern of overlapping, they are not synchronized with each other. The clock adjustments made in section 3.4.1 do not affect the validity period for ephemeral keys uploaded to a directory.

4.2 New Attacks Facilitated by PQConnect

We now turn our attention to the new attack surface added by the protocol. We consider resource exhaustion due to computation and amplification attacks.

4.2.1 Ratchet DoS

If a peer receives a packet with an epoch or packet number higher than what they have already computed, they will need to advance their receiving ratchet forward in order to obtain the decryption key for the incoming packet. An active attacker who can observe packets from an existing tunnel can learn the `tunnelID` as well as the epoch and packet numbers from the header fields. If an attacker injects a garbage packet with a very high epoch or packet number in its `PQConnect` header, the receiver will fail to verify the packet and thus discard it, but not before they spend CPU cycles computing the indicated packet key. Allowing unbounded advances along the ratchet would result in the recipient performing many KDF operations before observing that the packet fails to authenticate and decrypt. We want to ensure that any computation that an attacker can instigate is limited.

Implementing a ratchet window

To mitigate the above attack, we limit the forward distance along the ratchet that a peer can advance, based on the last valid packet that was received. Because the ratchet moves independently in two directions, we need to limit the forward distance in both.

In the epoch direction (represented by the "trunk" of the tree in in Figure 3.5), it is unlikely that an honest peer will be more than one or two epochs away due to local clock differences, so if the most recently sent packet that we received was from epoch E, we will allow ourselves to try decrypting packets up to and including epoch $E + 2$.

Along each "branch" of the tree, we limit ourselves to a distance of 504 keys (28 advances of the chain key in an epoch using batch key derivation of 18 keys) from the last authenticated packet we received on that branch. So if the last authentic received packet was $P_{i,j}$, we will try to decrypt packets $P_{k,l}$ for $i \leq k \leq i + 2$ and $l \leq j + 504$ if $k = i$, else $l \leq 504$.

Intermezzo: A tree-based ratchet window

A more complex solution to this problem, but one that vastly increases the receiving window size is to arrange the "branch" within each epoch not as a linear chain, but as a full binary tree of fixed size. This allows a receiver to accept and verify packets far ahead in the chain without needing to compute all intermediate chain keys. This solution is not used in `PQConnect` for reasons explained at the end of the section and in the next, but I include it here because it may be of interest to the reader and potentially useful in other applications.

To generate the tree, each chain key becomes a node in the tree. We apply the KDF to the first node in the epoch and derive three keys: the left daughter chain key, right daughter chain key, and the packet key for the node. The tree is traversed in depth-first order. Thus for a tree of size $2^{32} - 1$ in epoch i, the daughters of node $c_{i,0}$ are node $c_{i,1}$ and node $c_{i,2^{31}}$. The daughters of node $c_{i,2^{31}}$ are nodes $c_{i,2^{31}+1}$ and $c_{i,2^{31}+2^{30}}$, etc. When a packet arrives, the receiver traverses the tree to the node indicated by the packet number. If it encounters a node with a chain key before arriving at the destination node, it uses the chain key as input to the KDF, computing the chain keys of the daughters and the node's packet key. The chain key of the node is erased, and this process recurses until the target node is reached, at which point the packet key and daughter chain keys (if not at a leaf node) are computed, the packet is decrypted, and the node's packet and chain key erased.

While this approach allows for a much larger window, it fixes the maximum number of packets that can be sent in an epoch to the size of the tree. Additionally, the state of the chain grows logarithmically in the number of received packets in the worst case, whereas in the sliding window model the receiver's state never exceeds the window size plus the number of delayed packets.

Hardening the ratchet window

Typically in an AEAD scheme, the encrypting party must communicate with each encrypted message the nonce and associated data used to generate the ciphertext, if these values are not already known to the decrypting party. This is the purpose of the `tunnelID`, Epoch No., and Packet No. header fields in `PQConnect`. Because this metadata is needed by the decrypting party for verification and decryption, they are frequently not communicated confidentially, which potentially leaks valuable information to an attacker. In [CR19], Chan and Rogaway propose a scheme called NonceWrap to hide the nonce value and associated data of an AEAD scheme, as long as the decrypting peer can predict these values. At a high level, NonceWrap works by feeding the metadata through a combination of a hash function and block cipher to create a header value for the ciphertext that reveals no information to an attacker about the metadata for that message. If the decrypting party knows in advance what values this

Figure 4.1: Structure of a `PQConnect` packet with the `tunnelID`, Epoch No., and Packet No. replaced by the secure `pID` header: Light gray areas are authenticated but not encrypted. Dark gray is authenticated and encrypted

metadata can take, they can pre-compute these headers and maintain a series of associative arrays that point from a header value to the metadata needed to decrypt the message.

The `PQConnect` ratchet window described in section 4.2.1 naturally induces a predictable set of (`tunnelID`, Epoch No., Packet No.) triples at any moment in time for the receiving peer. Taking inspiration from NonceWrap, we can reduce the attack surface for an attacker trying to advance a peer's receiving ratchet if we hide these values when sending packets. Recall also that because `PQConnect` uses unique keys for each sent packet, we only need to hide a packet's associated data (the nonce is always 0).

One of the steps of NonceWrap is to feed the associated data of each ciphertext through a hash function (though not necessarily a cryptographic hash). Using a cryptographic hash for each (`tunnelID`, Epoch No., Packet No.) triple would indeed be computationally expensive, especially given that all we need is an associated value to each triple that an attacker cannot compute without also knowing the secret key. We can generate such a value the same way we generate the keys themselves, during the ratchet advance step using the KDF. We can straightforwardly extend the ratchet to generate not only packet keys, but also pseudo-random packet IDs (`pID`s) associated with each key.

The final component of this scheme is efficient storage and lookup of `pID`s and metadata. We use two associative arrays implemented as hash tables to perform bookkeeping. The first is `pIDtoAD` that points from `pID`s to (`tunnelID`, Epoch No, Packet No.) triples. The second stores the packet keys themselves, indexed by `tunnelID`, Epoch No., and Packet No. When a packet key is created, an entry is inserted in `pIDtoAD` linking the `pID` for that packet to the `tunnelID` and index for its packet key. The packet key itself is then inserted into `ADtoPacketKeys`. The full algorithm for initializing a receiving ratchet with associated `pID`s is depicted in Algorithm 1.

Replacing the `tunnelID`, Epoch No, header Packet No. fields by the `pID` also shortens the length of the header, saving many bytes of transmitted data over the lifetime of the tunnel. We update the packet format depicted in Figure 3.6 to the one shown in Figure 4.1.

Algorithm 1: Initialize New Receive Ratchet

Input: $\texttt{tunnelID}$, $\texttt{T}_\texttt{I}, \texttt{T}_\texttt{R}$, Associative array $\texttt{pIDtoAD}$, Associative array $\texttt{ADtoPacketKeys}$

branchLimit $\leftarrow 504$;
epochLimit $\leftarrow 3$;
if Self = Initiator **then**
  $c_0 \leftarrow T_R$;
else
  $c_0 \leftarrow T_I$;
end
for $i \in \{0, \ldots, \text{epochLimit} - 1\}$ **do**
  $c_{i+1}, c_{i,18}, c'_{i,0}, \texttt{pID}_{i,0}, \ldots, c'_{i,17}, \texttt{pID}_{i,17} \leftarrow \text{KDF}_{38}(c_i)$;
  $c_i \leftarrow c_{i+1}$;
  for $j \in \{0, \ldots, 17\}$ **do**
    $\texttt{pIDtoAD}[\texttt{pID}_{i,j}] \leftarrow (\texttt{tunnelID}, i, j)$;
    $\texttt{ADtoPacketKeys}[\texttt{tunnelID}][i][j] \leftarrow c'_{i,j}$;
  end
  for $j \in \{1, \ldots, \frac{\text{branchLimit}}{18} - 1\}$ **do**
    $c_{i,(j+1)\cdot 18}, c'_{i,(j\cdot 18)+0}, \texttt{pID}_{i,(j\cdot 18)+0}, \ldots, c'_{i,(j\cdot 18)+17}, \texttt{pID}_{i,(j\cdot 18)+17} \leftarrow \text{KDF}_{37}(c_{i,j\cdot 18})$;
    $c_{i,j\cdot 18} \leftarrow c_{i,(j+1)\cdot 18}$;
    for $k \in \{0, \ldots, 17\}$ **do**
      $\texttt{pIDtoAD}[\texttt{pID}_{i,j\cdot 18+k}] \leftarrow (\texttt{tunnelID}, i, j \cdot 18 + k)$;
      $\texttt{ADtoPacketKeys}[\texttt{tunnelID}][i][j \cdot 18 + k] \leftarrow c'_{i,j\cdot 18+k}$;
    end
  end
end

Chapter 5

Formal Verification of the `PQConnect` Handshake

In this chapter I use the symbolic verification tool TAMARIN prover to formally verify security properties of the `PQConnect` handshake. First I give a brief introduction to TAMARIN and its syntax. I then describe the security properties that the 0-RTT and 1-RTT `PQConnect` handshakes should satisfy. Finally I describe the model I constructed and the verification of these properties. See Appendix A for the full model and lemmas.

5.1 TAMARIN Prover

TAMARIN prover (TAMARIN) is a formal verification tool for proving properties of cryptographic protocols, such as confidentiality, peer-to-peer authentication, and forward secrecy [Sch+12]. It has been used to analyze security properties of widely deployed cryptographic network protocols, such as TLS 1.3 [Cre+17] and the WireGuard protocol handshake [DM18]. Protocol properties in TAMARIN are proven (or disproven) in the Dolev-Yao model—that is, where the adversary has complete control to eavesdrop, intercept, modify, and insert messages into the channel [DY83]. 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.

Models in TAMARIN can define their own functions and equational theories, for example to represent AEAD operations and one-way functions. Cryptographic operations are assumed by default to be ideal; random values such as keys, nonces, and outputs of one-way functions are treated as perfectly random, unique, and unguessable. Thus TAMARIN has no notion of attacks against a particular cryptosystem unless they are explicitly included in the model. The automated prover reasons about protocol properties over an unbounded number concurrent protocol instances, and thus can uncover unexpected interactions between different parts of the protocol state machine.

5.2 Syntax

In this section I will give a brief overview of the main components of a TAMARIN model and their syntax.

5.2.1 Functions and equations

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

5.2.2 Facts and rules

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

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

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

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

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

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

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

```
[ Out(h(c)]
```

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

5.2.3 Lemmas

In TAMARIN, properties about models are analyzed by writing and proving (or disproving) lemmas. Lemmas are guarded first-order logical statements about labeled actions over all possible rule executions (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 then one where a rule that produced this action fact was executed. Actions can be used to both reason about and restrict the executions of rules in a lemma.

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

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

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

The TAMARIN prover uses a constraint solving algorithm to find counterexamples to the provided lemmas [Sch+12]. When a lemma is intended to prove a universally quantified statement, like $\forall_x P(x)$ for some predicate P, TAMARIN converts this to the equivalent existentially quantified statement $\neg\exists_x(\neg P(x))$, and then tries to find a contradiction. The decidability of statements in first-order logic is undecidable in general, so there exist lemmas for which the algorithm will not terminate. However, when the prover does terminate, then it either proves the statement to be true over unbounded executions or derives a contradiction.

5.3 Security Properties

This section enumerates the security properties that the `PQConnect` handshakes provide. First the handshakes are treated separately, and then properties are verified where both handshakes can be performed concurrently.

Executability and correctness: 0-RTT (resp. 1-RTT)

Two parties are able to complete the 0-RTT (resp. 1-RTT) handshake, and as a result derive the same `tunnelID`, $\texttt{T}_\texttt{I}$, and $\texttt{T}_\texttt{R}$ values.

Key confidentiality: 0-RTT (resp. 1-RTT)

In the absence of any weakness in the cryptosystems, two parties who perform the 0-RTT (resp. 1-RTT) handshake derive transport keys $\mathtt{T_I}$ and $\mathtt{T_R}$, and these keys are unknown to the attacker.

Quantum confidentiality: 0-RTT (resp. 1-RTT)

An attacker who obtains the long-term and ephemeral `x25519` private keys of the Responder cannot recover the transport keys.

Forward secrecy: 0-RTT (resp. 1-RTT)

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

Quantum forward secrecy: 0-RTT (resp. 1-RTT)

A FS attacker who obtains the Responder's long-term secret keys and and ephemeral `x25519` secret key does not break the confidentiality of the transport keys.

Quantum forward secrecy: combined

A quantum FS attacker as above who can concurrently execute arbitrary 0-RTT and 1-RTT handshakes cannot recover the transport keys.

Responder to initiator authentication

If I and R complete either the 0-RTT handshake or the 1-RTT, then I knows they are communicating with R.

5.4 Verified Lemmas in TAMARIN

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

Protocol executability and correctness

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

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

```
lemma 0_RTT_executable:
  /* There exists a trace, such that */
```

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

lemma 1_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 1-RTT handshake finished for id at tme #i */
      One_RTT(id) @ #i
      /* The initiator established a tunnel with R at time #i*/
      & InitiatorTunnel(R,id,ti,tr) @ #i
      /* and the Responder estalished a tunnel with the same */
      /* tunnelID ad transport keys at time #j*/
      & ResponderTunnel(R,id,tr,ti) @ #j
  "
```

These lemmas are identical except for the actions `Zero_RTT(id)` in the first and `One_RTT` in the second. These actions are both produced by the final step in each handshake, and allow us to select which handshake we want the lemmas to reason about. For example, the predicates `Zero_RTT(id), InitiatorTunnel(R,id,ti,tr)` and `ResponderTunnel(R,id,tr,ti)` can only all be true if the 0-RTT handshake succeeded and produced `id, ti`, and `tr` as its output.

Key confidentiality and forward secrecy

Any quantum FS attacker trying to break the confidentiality of the `PQConnect` handshake is of course free to perform an attack that does not use compromised pre-quantum and long-term public keys. Thus showing quantum forward secrecy implies classical forward secrecy, quantum confidentiality, and classical confidentiality. We therefore prove for each handshake individually a single lemma that satisfies all four confidentiality properties.

To model a quantum attacker for the 1-RTT handshake, we modify the handshake protocol rules slightly, so that after completing the handshake, each party sends their ephemeral `x25519` private key out onto the public channel. We then show that as long as the long-term

mceliece8192128 key is not compromised before the completion of the handshake, the attacker does not know either of the transport keys. Note that because the ephemeral keys in the 1-RTT version are not public, a quantum attacker should not be able to recover either party's ephemeral x25519 private key. By giving the attacker these keys, we are proving the property against an even stronger attacker. The full lemma is given below:

```
lemma 1_RTT_FS_confidential:
  /* It cannot occur that */
  "
    not(
      Ex R id ti tr #i.
        /* an initiator has performed a 1-RTT handshake with a responder R */
        (
          InitiatorTunnel(R,id,ti,tr) @ #i
          & One_RTT(id) @ #i
          /* and the adversary knows ti or tr */
          & ((Ex #j. K(ti) @ #j)
          |(Ex #k. K(tr) @ #k))
        )
        /* Without there having been a prior reveal */
        /* of the long term McEliece secret key */
        & not(Ex #r. PqSskReveal(R) @ #r & r < i)
    )
  "
```

For the 0-RTT handshake, the Responder's ephemeral public keys are already public, so a quantum FS attacker is an attacker who can later learn the long-term private keys and the ephemeral x25519 private key of the Responder. We therefore simply require that the private sntrup4591761 key is never known to anyone besides I or R:

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

    /* It cannot be that a  */
  "
    not(
      Ex S id ti tr #i.
        /* client has performed a 0-RTT handshake with a server'S' */
        InitiatorTunnel(S,id,ti,tr) @ #i
        & Zero_RTT(id) @ #i
        /* and the adversary knows ti or tr */
        & (
            (Ex #j. K(ti) @ #j)
```

```
            |(Ex #k. K(tr) @ #k)
          )
        /* without there having been a reveal of the ephemeral sntrup key */
        & not(Ex #r. PqEskReveal(S) @ #r )
    )
  "
```

It is worth noting that the lemma actually does not put a constraint on when the attacker learns R's private keys, so this lemma actually proves a stronger notion of confidentiality than forward secrecy.

We now remove the restrictions on which handshake is performed and show that the execution of both handshakes concurrently does not produce a state machine that introduces vulnerabilities in confidentiality.

```
lemma Combined_FS_confidential:
  /* For all traces */
  all-traces
  /* For all responders R, */
  /* tunnel values id, ti, tr */
  /* and time values #i #j, it holds that */
  "
    All R id ti tr #i #j.
      /* If an initiator has established a tunnel with */
      /* R and values id, ti, tr at time #i */
      (
        InitiatorTunnel(R,id,ti,tr) @ #i
        /* And R has established the same tunnel at time #j */
        & ResponderTunnel(R,id,ti,tr) @ #j
        /* And R's long term McEliece key was not revealed before time #j or #i */
        & not(Ex #k. PqSskReveal(R) @ #k & #k < #j & #k < #i)
      )
      /* Then an attacker never learns ti or tr */
      ==> (not(Ex #l. K(ti) @ #l) & not(Ex #m. K(tr) @ #m))
  "
```

Responder to initiator authentication

Finally we show that for both the 0-RTT and 1-RTT handshake, if an initiator has created a tunnel and a Responder has created the same tunnel, then the initiator must have created the tunnel with that particular Responder.

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

  "
    All R S id ti tr #i #j.
```

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

Chapter 6

Realization of PQConnect

As a proof of concept (PoC), I implemented the PQConnect protocol in Python. In total this comprises about one thousand lines of code, excluding test files. I first discuss the design of the PoC and then suggest appropriate modifications for a production implementation of PQConnect.

6.1 PQConnect PoC Architecture

In order to motivate the claim that PQConnect is actually useful in practice, I wrote a Python 3 implementation of the protocol described in chapter 3. Python is a widely used, general-purpose programming language with decent native networking libraries and a syntax similar to other widely used languages like Java and C. There are also Python wrappers for the C-language libpqcrypto (https://libpqcrypto.org/) post-quantum cryptographic library and a rust implementation of BLAKE3 (https://pypi.org/project/blake3/), which together contain all the cryptographic primitives that I use in PQConnect. Altogether this means that there was very little configuration needed to translate the PQConnect specification into software, and very few dependencies required to use the software.

6.1.1 Client-Side Proxy

The client-side software of PQConnect consists of a SOCKS5 proxy running locally on the client's machine, into which the cryptographic operations hook. SOCKS5 is a transport/session layer proxy. The client first configures an application such as a browser to route its traffic through the proxy instead of connecting directly to the internet. For each new website the client visits in their browser, the browser creates a new socket with the SOCKS5 proxy, the proxy creates a second socket with the remote server that hosts the website, and the traffic is passed from one connection to the other. One way to visualize how this works is to imagine a person holding two telephones, each with one person on the other end of the line, and simply repeating the messages from one person to the other in both directions.

PQConnect DNS discovery

To illustrate how passive PQConnect discovery works using DNS, I will briefly describe how I configured DNS on my own web server to broadcast its PQConnect public key hash. I then

illustrate how the Python `socket` library can be used to obtain this hash from within the SOCKS5 proxy.

Add DNS entry

Name	Type	Value	Prio	TTL
	A	94.130.73.234		1800
pq11w2gtc51cynf00wh6munqq8ggphzts9976bc6n40b7fxr1bl2bd1	A	94.130.73.234		1800
	AAAA	2a01:4f8:c0c:2fd7::2		1800
pq11w2gtc51cynf00wh6munqq8ggphzts9976bc6n40b7fxr1bl2bd1	AAAA	2a01:4f8:c0c:2fd7::2		1800
*	CNAME	pq11w2gtc51cynf00wh6munqq8ggphzts9976bc6n40b7fxr1bl2bd1.jonatha.nl		300
	NS	ns.inwx.de		86400
	NS	ns2.inwx.de		86400
	NS	ns3.inwx.eu		86400
	SOA	ns.inwx.de hostmaster@inwx.de 2021081008		86400

Figure 6.1: DNS configuration for the 'jonatha.nl' domain. The CNAME record maps all subdomains (*) to the canonical name `pq11w2gtc51cynf00wh6munqq8ggphzts9976bc6n40b7fxr1bl2bd1.jonatha.nl`. This canonical name, which contains the characters `pq1` followed by the Base32-encoded hash of the long term public keys, then maps to IPv4/IPv6 addresses via respective A/AAAA records.

Figure 6.1 shows the DNS records for the domain 'jonatha.nl'. The subdomain wildcard '*' points to a CNAME record containing the public key hash for the domain. This canonical name is then associated with the server's IPv4 and IPv6 addresses. Using the `dig` DNS lookup utility on Linux, we can see that the CNAME is returned along with the resolved IP for the domain 'www.jonatha.nl' during a normal A record query. The CNAME appears twice at the bottom, but the first is shortened for readability.

```
---:) dig www.jonatha.nl

; <<>> DiG 9.16.18 <<>> www.jonatha.nl
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 4015
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: e35f1ccf1583659fe4563b6d61125e790a65b3c9acd6bbe6 (good)
;; QUESTION SECTION:
;www.jonatha.nl.                        IN      A

;; ANSWER SECTION:
www.jonatha.nl.   300  IN  CNAME  pq11w2gtc51[...]l2bd1.jonatha.nl.
pq11w2gtc51cynf00wh6munqq8ggphzts9976bc6n40b7fxr1bl2bd1.jonatha.nl.
     1800 IN A 94.130.73.234
```

In the Python `socket` library, the method `socket.getaddrinfo()` performs a DNS query when the first parameter is a string containing the queried domain name. It also takes an optional "flags" parameter where the programmer can specify that they want the canonical name returned with the response if it is different from the query name. The following output is from an interactive Python 3 shell. The output is a list of tuples. In the first list item, the fourth element is the canonical name containing the public key hash.

```
In [1]: import socket
In [2]: socket.getaddrinfo('www.jonatha.nl',443,flags=socket.AI_CANONNAME)
Out[2]:
[(<AddressFamily.AF_INET: 2>,
  <SocketKind.SOCK_STREAM: 1>,
  6,
  'pq11w2gtc51cynf00wh6munqq8ggphzts9976bc6n40b7fxr1bl2bd1.jonatha.nl',
  ('94.130.73.234', 443)),...]
```

Hooking `PQConnect` into the SOCKS5 proxy

Knowing that we can intercept DNS queries and check for the correctly formatted public key hash allows us to hook the `PQConnect` protocol into the existing SOCKS5 proxy. When a correctly formatted CNAME is returned, we try to perform a handshake and establish a tunnel with the server over a UDP socket. The tunnel then sits at the interface of the local-facing connection and the server-facing one, encrypting and decrypting traffic transparently to the user.

The proxy also maintains several global dictionaries: one that maps public key hashes to active `tunnelID`s, one that maps public key hashes to long-term keys, and one that maps `tunnelID`s to active tunnels. This way if the proxy attempts to reconnect to a known remote peer, it can simply retrieve an existing tunnel object or immediately perform a handshake.

6.1.2 Server-side architecture

Implementing the server's side using SOCKS is complicated, because it requires interfacing the SOCKS proxy with the particular service you `PQConnect` to support. For the PoC the goal was to simply implement a basic server to provide a static HTTP web page on a fixed port. The server only maintains a dictionary mapping `tunnelID`s to active tunnels. For a fully featured release there are more elegant ways to proxy network packets than in a SOCKS5 proxy for both client and server, but especially for the server. I will discuss these in more detail in section 6.2.1.

6.1.3 The Core Protocol

In this section I describe the general organization of the PoC code and how components of the protocol are implemented.

The Initiator

The Initiator key distribution and handshake functionality is implemented as a standalone class PQCClientConnection. Each PQCClientConnection object exists for a single handshake

with a single Responder. If the Responder's public keys are already in the SOCKS5 key cache, it can be optionally initialized with these public keys. Whenever the SOCKS5 DNS query returns a CNAME with a correctly formatted public key hash, the proxy creates a new PQCClientConnection object to retrieve the Responder's keys, compute the 0-RTT or 1-RTT handshake messages, and return a new tunnel from the resulting shared secrets.

The Responder

The Responder functionality is implemented in two classes. The first is PQConnectServer, which initializes the Responder's key values and listens for incoming packets on a fixed UDP port. For each incoming packet it creates an object from the second class PQCServerConnection. PQCServerConnection extends the Python thread class and handles the complementary functionality of the PQCClientConnection class: key distribution, handshake response/processing, and tunnel creation.

Tunnel objects and key handling

When the handshake protocol has successfully completed, both the Initiator and Responder generate an object of the Tunnel class. I capitalize "Tunnel" here to indicate this refers specifically to the software implementation. The Tunnel class encapsulates all the symmetric cryptography, key erasure, and epoch synchronization. The two most important methods of the Tunnel class are Tunnel.tunnel_send, and Tunnel.tunnel_recv, which implement the actual encryption and decryption operations. The Tunnel class uses timer threads to implement the 120 second epoch erasure. When a new epoch begins in the decrypting peer's receiving ratchet, a timer thread sleeps for 120 seconds and then erases all remaining keys from the epoch that spawned it.

The last important class in `PQConnect` is the McEliecePKTree class, which implements the public key Merkle tree data structure, its verification methods, and several helper methods for hash encoding and node retrieval for client requests. Finally, I implemented the `ChaCha20-Poly1305` AEAD scheme and `ChaCha20`-based KDF.

6.2 Production Implementation

The POC that I wrote for this thesis showed that this protocol can indeed be implemented and deployed. Anyone who has been involved with the development of cryptographic or networking software will realize immediately that there are important shortcomings that are inherent to the POC design.

First among these is that Python, while a relatively easy programming language to use for modeling a protocol and "safe" from common C-flavored programming bugs like buffer overflows, does not allow the programmer to directly control the values assigned to memory and the representation of data in memory. For a feature like fast key erasure, where we want to guarantee that keys are gone after at most 120 seconds, that they have not been moved to swap, etc., Python does not give us sufficient control over memory usage. As an interpreted language, instead of a compiled language, Python is also much slower than compiled languages like C or Rust.

6.2.1 Implementing `PQConnect` at the Network Layer

From an architectural perspective, Python is inappropriate for a reason that actually has very little to do with Python. The choice of using a SOCKS5 proxy for hooking `PQConnect` functionality into was one of convenience for me as a programmer, but not as a user. User applications need to manually be configured to route their traffic through SOCKS, and there are many ways this can go wrong. In Firefox, for example, you have the option of routing DNS queries through SOCKS or making DNS resolution library calls directly from the browser. If a user does not toggle this option, then `PQConnect` connections will not be made even with servers that support it.

Fundamentally, the goal of this project is to provide post-quantum security to *all* Internet traffic, not just traffic coming from a user's browser. This requires `PQConnect` to work in kernel space, where the network layer protocols are implemented by the operating system. Software that works in kernel space has to be implemented in Linux as a kernel module. Kernel modules must be written in C[1]. Ideally `PQConnect` would implement a virtual network interface that can grab packets intended for known `PQConnect` destinations directly from the kernel's network buffer. This approach is similar to the architecture of the WireGuard VPN, and it has the benefit that it not only can encrypt and encapsulate all applicable packets directly where they are managed by the kernel, but also that the behavior of `PQConnect` is far more transparent to the user than a SOCKS5 proxy is. Implementing tunnel creation, packet encryption/decryption, and encapsulation in the kernel network layer is an elegant way to tunnel-ify as much traffic as possible on a user's machine.

6.3 Road to deployment

In order for `PQConnect` to create post-quantum tunnels spanning the internet, it needs to be supported by part of the existing network infrastructure. `PQConnect`'s security depends not only on the software running on a single client and single server's machines, but also on DNS servers and key directories that support the protocol. It is vital that the DNS queries used to discover `PQConnect` Responders be sent to name servers that also implement `PQConnect`, so that a chain of trust from the name server to the Responder's public key can be established. Establishing a tunnel with a DNS server is identical to establishing a tunnel with any other server, however, so there is no extra technical challenge in the software that needs to be overcome for name servers to support the protocol, and baking the long-term keys for DNS name servers supporting `PQConnect` into the software roots the authentication chain from DNS server to Responder, as said before.

Additionally, in order to support 0-RTT handshakes, a network of ephemeral key servers should be created to distribute keys as described in section 4.1.3. While this is not strictly required due to the 1-RTT handshake, the ability to pre-fetch keys from a directory would add an attractive performance benefit that would entice more people to use the protocol.

[1] There is an effort to expand support for the Rust language in the Linux kernel, as Rust implements several safety measures that C does not, such as bounds checking in arrays. See, e.g., https://github.com/Rust-for-Linux/linux

Appendices

Appendix A

The PQConnect TAMARIN Model

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

Authors:  Jonathan Levin
Date:          July 2021
*/

theory PQCHandshake
begin

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

/* PKI */
rule Register_static_pq_pk:
    [ Fr(~ssk) ]
  -->
    [ !PQ_Ssk($S, ~ssk), !PQ_Spk($S, pk(~ssk)) ]

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


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

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

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

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

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

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


/* 0-RTT Handshake */
rule 0RTT_PQConnectI:
  let
    c0 = aenc(~k0, spkRmceliece)
    CI = ~k0
    HI = c0
    c1 = aeadenc(CI,'0','g'^~eskIx25519,HI)
    HI = h(<HI,c1>)
    k1 = spkRx25519^~eskIx25519
    CI = kdf(<CI,k1>)
    k2 = epkRx25519^~eskIx25519
    CI = kdf(<CI,k2>)
    c2 = aenc(~k3, epkRsntrup)
    c3 = aeadenc(CI,'0',c2,HI)
    CI = kdf(<CI,~k3>)
    HI = h(<HI,c3>)
    tid = kdf(<CI,HI>)
    TI = kdf2(<CI,HI>)
    TR = kdf3(<CI,HI>)
```

```
  in
    [ !PQ_Spk(R,spkRmceliece),
      !NPQ_Spk(R,spkRx25519),
      !PQ_Epk(R,epkRsntrup),
      !NPQ_Epk(R,epkRx25519),
      Fr(~eskIx25519),
      Fr(~k0),
      Fr(~k3)]
    --[Zero_RTT(tid), InitiatorTunnel(R,tid,TI,TR)]->
    [ Out(<'1',c0,c1,c3>) ]

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

/* 1-RTT Handshake */
rule 1RTT_PQConnectI_1:
  let msgtype = '1'
      c0 = aenc(~k0, spkRmceliece)
      CI = ~k0
      HI = c0
      c1 = aeadenc(CI,'0','g'^~eskIx25519,HI)
      k1 = spkRx25519^~eskIx25519
      CI = kdf(<CI,k1>)
```

```
    HI = h(<HI,c1>)
    c2 = aeadenc(CI,'0',pk(~eskIsntrup),HI)
    HI = h(<HI,c2>)
  in
    [
      !PQ_Spk($R, spkRmceliece),
      !NPQ_Spk($R, spkRx25519),
      Fr(~eskIx25519),
      Fr(~eskIsntrup),
      Fr(~k0),
      Fr(~hid)
    ]
    -->
    [
      Init_1(~hid, $R, CI, HI, ~eskIx25519, ~eskIsntrup),
      Out(<'2',c0,c1,c2>)
    ]

rule 1RTT_PQConnectR_1:
  let
    k0 = adec(c0,~sskRmceliece)
    CR = k0
    HR = c0
    epkIx25519 = aeaddec(CR,'0',c1,HR)
    k1 = epkIx25519^~sskRx25519
    CR = kdf(<CR,k1>)
    HR = h(<HR,c1>)
    epkIsntrup = aeaddec(CR,'0',c2,HR)
    HR = h(<HR,c2>)
    /* Everything up until here is correct \o/ */

    c3 = aeadenc(CR,'1','g'^~eskRx25519,HR)
    k2 = epkIx25519^~eskRx25519
    CR = kdf(<CR,k2>)
    HR = h(<HR,c3>)
    c4 = aenc(~k3, epkIsntrup)
    c5 = aeadenc(CR,'0',c4,HR)
    CR = kdf(<CR,~k3>)
    HR = h(<HR,c5>)
    iC = CR
    iH = HR
    tid = kdf(<CR,HR>)
    TI = kdf2(<CR,HR>)
    TR = kdf3(<CR,HR>)

  in
    [
```

```
        !PQ_Ssk($R, ~sskRmceliece),
        !NPQ_Ssk($R, ~sskRx25519),
        Fr(~eskRx25519),
        Fr(~k3),
        In(<'2',c0,c1,c2>)
    ]
    --[ResponderTunnel($R,tid,TR,TI)]->
    [ Out(<'3',c3,c5>),
      Out(~eskRx25519) ]

rule 1RTT_PQConnectI_2:
  let
    CI = oldCI
    HI = oldHI
    epkRx25519 = aeaddec(CI,'1',c3, HI)
    k2 = epkRx25519^~eskIx25519
    CI = kdf(<CI,k2>)
    HI = h(<HI,c3>)
    c4 = aeaddec(CI,'0',c5,HI)
    k3 = adec(c4,~eskIsntrup)
    CI = kdf(<CI,k3>)
    HI = h(<HI,c5>)
    tid = kdf(<CI,HI>)
    TI = kdf2(<CI,HI>)
    TR = kdf3(<CI,HI>)

  in
    [
        Init_1(~hid, $R, oldCI, oldHI, ~eskIx25519, ~eskIsntrup),
        In(<'3',c3,c5>)
    ]
    --[One_RTT(tid),
       Secret(tid),
       Secret(TI),
       Secret(TR),
       InitiatorTunnel($R,tid,TI,TR)]->
    [ Out(~eskIx25519)]


/* lemmas */

lemma 0_RTT_executable:
  /* There exists a trace, such that */
  exists-trace
  /* There exists a responder R, tunnelID id, transport keys ti */
  /* and tr, and times #i and #j*/
  "
```

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

lemma 1_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 1-RTT handshake finished for id at tme #i */
      One_RTT(id) @ #i
      /* The initiator established a tunnel with R at time #i*/
      & InitiatorTunnel(R,id,ti,tr) @ #i
      /* and the Responder estalished a tunnel with the same */
      /* tunnelID ad transport keys at time #j*/
      & ResponderTunnel(R,id,tr,ti) @ #j
  "

lemma 1_RTT_FS_confidential:
  /* It cannot occur that */
  "
    not(
      Ex R id ti tr #i.
        /* an initiator has performed a 1-RTT handshake with a responder R */
        (
          InitiatorTunnel(R,id,ti,tr) @ #i
          & One_RTT(id) @ #i
          /* and the adversary knows ti or tr */
          & ((Ex #j. K(ti) @ #j)
          |(Ex #k. K(tr) @ #k))
        )
        /* Without there having been a prior reveal */
        /* of the long term McEliece secret key */
        & not(Ex #r. PqSskReveal(R) @ #r & r < i)
    )
  "

lemma 0_RTT_FS_confidential:
    /* Because ephemeral keys are public in this case, it is
```

```
    sufficient to show that if the ephemeral sntrup secret key is
    never revealed, the adversary cannot learn ti or tr. That is,
    even if the long term keys have been compromised (even before
    the handshake!), we achieve confidentiality */

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

lemma Combined_FS_confidential:
  /* For all traces */
  all-traces
  /* For all responders R, */
  /* tunnel values id, ti, tr */
  /* and time values #i #j, it holds that */
  "
    All R id ti tr #i #j.
      /* If an initiator has established a tunnel with */
      /* R and values id, ti, tr at time #i */
      (
        InitiatorTunnel(R,id,ti,tr) @ #i
        /* And R has established the same tunnel at time #j */
        & ResponderTunnel(R,id,ti,tr) @ #j
        /* And R's long term McEliece key was not revealed before time #j or #i */
        & not(Ex #k. PqSskReveal(R) @ #k & #k < #j & #k < #i)
      )
      /* Then an attacker never learns ti or tr */
      ==> (not(Ex #l. K(ti) @ #l) & not(Ex #m. K(tr) @ #m))
  "

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

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

end
```

Bibliography

[AB00] Michel Abdalla and Mihir Bellare. "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques". In: *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*. Ed. by Tatsuaki Okamoto. Vol. 1976. Lecture Notes in Computer Science. Springer, 2000, pp. 546–559. DOI: 10.1007/3-540-44448-3_42. URL: https://doi.org/10.1007/3-540-44448-3%5C_42.

[Alb+20] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. *Classic McEliece: conservative code-based cryptography*. Round 3 submission to NIST post-quantum call for proposals. https://classic.mceliece.org. 2020.

[Ber+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. *NTRU Prime: round 3*. Round 3 submission to NIST post-quantum call for proposals. https://ntruprime.cr.yp.to. 2020.

[Ber+21] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, and Nicola Tuveri. "OpenSSLNTRU: Faster post-quantum TLS key exchange". In: *CoRR* abs/2106.08759 (2021). arXiv: 2106.08759. URL: https://arxiv.org/abs/2106.08759.

[Ber06] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *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*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. Lecture Notes in Computer Science. Springer, 2006, pp. 207–228. DOI: 10.1007/11745853_14. URL: https://doi.org/10.1007/11745853%5C_14.

[Ber17] Daniel J Bernstein. *2017.07.23: Fast-key-erasure random-number generators*. http://blog.cr.yp.to/20170723-random.html. July 2017.

[Ber18] Daniel J. Bernstein. *D 2.5 Internet: Integration*. PQCRYPTO project Deliverable D 2.5 http://pqcrypto.eu.org/deliverables/d2.5.pdf. 2018.

[Ber20] Daniel J. Bernstein. *ROUND 3 OFFICIAL COMMENT: CRYSTALS-KYBER*. https://web.archive.org/web/20210705065827/https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/NSe0wAzKJtA/m/dLP05gv7BgAJ. Accessed: 2021-07-05. 2020.

[Bin+17] Nina Bindel, Udyani Herath, Matthew McKague, and Douglas Stebila. "Transitioning to a Quantum-Resistant Public Key Infrastructure". In: *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*. Ed. by Tanja Lange and Tsuyoshi Takagi. Vol. 10346. Lecture Notes in Computer Science. Springer, 2017, pp. 384–405. DOI: 10.1007/978-3-319-59879-6_22. URL: https://doi.org/10.1007/978-3-319-59879-6%5C_22.

[Bos+15] 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*. IEEE Computer Society, 2015, pp. 553–570. DOI: 10.1109/SP.2015.40. URL: https://doi.org/10.1109/SP.2015.40.

[Cas+18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. "CSIDH: An Efficient Post-Quantum Commutative Group Action". In: *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*. Ed. by Thomas Peyrin and Steven D. Galbraith. Vol. 11274. Lecture Notes in Computer Science. Springer, 2018, pp. 395–427. DOI: 10.1007/978-3-030-03332-3_15. URL: https://doi.org/10.1007/978-3-030-03332-3%5C_15.

[CR19] John Chan and Phillip Rogaway. "Anonymous AE". In: *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part II*. Ed. by Steven D. Galbraith and Shiho Moriai. Vol. 11922. Lecture Notes in Computer Science. Springer, 2019, pp. 183–208. DOI: 10.1007/978-3-030-34621-8_7. URL: https://doi.org/10.1007/978-3-030-34621-8%5C_7.

[Cre+17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1773–1788. ISBN: 9781450349468. DOI: 10.1145/3133956.3134063. URL: https://doi.org/10.1145/3133956.3134063.

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

[Danb] Daniel J. Bernstein and Tanja Lange (editors). *eBASH (ECRYPT Benchmarking of All Submitted Hashes)*. https://bench.cr.yp.to/results-hash.html. Accessed: 2021-07-05.

[DM18] Jason A. Donenfeld and Kevin Milner. *Formal Verification of the WireGuard Protocol*. https://www.wireguard.com/papers/wireguard-formal-verification.pdf. Draft Revision b956944. 2018.

[Don17] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Soci-

ety, 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/.

[DY83] Danny Dolev and Andrew Chi-Chih Yao. "On the security of public key protocols". In: *IEEE Trans. Inf. Theory* 29.2 (1983), pp. 198–207. DOI: 10.1109/TIT.1983.1056650. URL: https://doi.org/10.1109/TIT.1983.1056650.

[FMS01] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. "Weaknesses in the Key Scheduling Algorithm of RC4". In: *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers*. Ed. by Serge Vaudenay and Amr M. Youssef. Vol. 2259. Lecture Notes in Computer Science. Springer, 2001, pp. 1–24. DOI: 10.1007/3-540-45537-X_1. URL: https://doi.org/10.1007/3-540-45537-X%5C_1.

[Hül+21] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 304–321. DOI: 10.1109/SP40001.2021.00030. URL: https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00030.

[KS01] Fabian Kuhn and René Struik. "Random Walks Revisited: Extensions of Pollard's Rho Algorithm for Computing Multiple Discrete Logarithms". In: *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers*. Ed. by Serge Vaudenay and Amr M. Youssef. Vol. 2259. Lecture Notes in Computer Science. Springer, 2001, pp. 212–229. DOI: 10.1007/3-540-45537-X_17. URL: https://doi.org/10.1007/3-540-45537-X%5C_17.

[McE78] Robert J. McEliece. "A Public-Key Cryptosystem Based On Algebraic Coding Theory". In: *Deep Space Network Progress Report* 44 (Jan. 1978), pp. 114–116.

[Mos18] Michele Mosca. "Cybersecurity in an Era with Quantum Computers: Will We Be Ready?" In: *IEEE Secur. Priv.* 16.5 (2018), pp. 38–41. DOI: 10.1109/MSP.2018.3761723. URL: https://doi.org/10.1109/MSP.2018.3761723.

[MP16] Moxie Marlinspike and Trevor Perrin. *The Double Ratchet Algorithm*. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf. 2016.

[MS01] Itsik Mantin and Adi Shamir. "A Practical Attack on Broadcast RC4". In: *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2-4, 2001, Revised Papers*. Ed. by Mitsuru Matsui. Vol. 2355. Lecture Notes in Computer Science. Springer, 2001, pp. 152–164. DOI: 10.1007/3-540-45473-X_13. URL: https://doi.org/10.1007/3-540-45473-X%5C_13.

[NL18] Yoav Nir and Adam Langley. "ChaCha20 and Poly1305 for IETF Protocols". In: *RFC* 8439 (2018), pp. 1–46. DOI: 10.17487/RFC8439. URL: https://doi.org/10.17487/RFC8439.

[OCo+] Jack O'Connor, Jean Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. *BLAKE3 one function, fast everywhere*. https://raw.githubusercontent.com/BLAKE3-team/BLAKE3-specs/master/blake3.pdf. Version 20200221164500; commit f3a9c29.

[Per18] Trevor Perrin. *The Noise Protocol Framework*. https://noiseprotocol.org/noise.pdf. 2018.

[Pet+13] W. Michael Petullo, Xu Zhang, Jon A. Solworth, Daniel J. Bernstein, and Tanja Lange. "MinimaLT: minimal-latency networking through better security". In: *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. Ed. by Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung. ACM, 2013, pp. 425–438. DOI: 10.1145/2508859.2516737. URL: https://doi.org/10.1145/2508859.2516737.

[Res18] Eric Rescorla. "The Transport Layer Security (TLS) Protocol Version 1.3". In: *RFC* 8446 (2018), pp. 1–160. DOI: 10.17487/RFC8446. URL: https://doi.org/10.17487/RFC8446.

[Sch+12] Benedikt Schmidt, Simon Meier, Cas J. F. Cremers, and David A. Basin. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties". In: *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*. Ed. by Stephen Chong. IEEE Computer Society, 2012, pp. 78–94. DOI: 10.1109/CSF.2012.25. URL: https://doi.org/10.1109/CSF.2012.25.

[Sen11] Nicolas Sendrier. "Decoding One Out of Many". In: *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*. Ed. by Bo-Yin Yang. Vol. 7071. Lecture Notes in Computer Science. Springer, 2011, pp. 51–67. DOI: 10.1007/978-3-642-25405-5_4. URL: https://doi.org/10.1007/978-3-642-25405-5%5C_4.

[Sho94] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700. URL: https://doi.org/10.1109/SFCS.1994.365700.

[SSW20] Peter Schwabe, Douglas Stebila, and Thom Wiggers. "Post-Quantum TLS Without Handshake Signatures". In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM, 2020, pp. 1461–1480. DOI: 10.1145/3372297.3423350. URL: https://doi.org/10.1145/3372297.3423350.